Rapid Prototyping for Hardware Accelerators in the Medical Imaging Domain

Schnelle Prototyperstellung von Hardware-Beschleunigern für die medizinische Bildverarbeitung

Der Technischen Fakultät
der Friedrich-Alexander-Universität
Erlangen-Nürnberg
zur
Erlangung des Doktorgrades Dr.-Ing.

vorgelegt von

Moritz Gabriel Schmid

aus Nürnberg

Als Dissertation genehmigt
von der Technischen Fakultät
der Friedrich-Alexander-Universität Erlangen-Nürnberg
Tag der mündlichen Prüfung: 24.07.2015

Vorsitzende des Promotionsorgans: Prof. Dr.-Ing. habil. Marion Merklein

Gutachter: Prof. Dr.-Ing. Jürgen Teich
Prof. Dr. Marco Platzner

Acknowledgments

I would first of all like to express my sincere gratitude towards my doctoral advisor, Professor Dr.-Ing. Jürgen Teich, for all the support, patience and guidance over the course of the last years. Also, I would like to thank Professor Dr. Marco Platzner for being the co-examiner of my thesis.

I would like to thank Dr.-Ing. Frank Hannig for interest, support and the numerous hours of discussions and advice about this thesis and my doctoral study in general.

I would like to thank all my fantastic colleagues with whom I have collaborated on many scientific studies during my time in Erlangen. In no particular order they are Oliver Reiche, Christian Schmitt, Alex Tanase, Daniel Ziener, Markus Blocherer, Vahid Lari, Srinivas Boppu, and Shravan Muddasani.

In particular, I would like to thank my previous office mate, co-commuter, and coffee-companion, Moritz Mühlenthaler, for the countless hours of scientific and off-topic discussions, reading over my thesis and providing valuable feedback, and of course for the many trips to the *automaton*.

Last but not least, I would like to thank my family and friends for continuous support over the last years. In particular, I want to thank my cousin Corinne who always had an open ear for all my troubles and gave me a lot of valuable advice during the writing process.

Moritz Schmid
Erlangen, August 2015

Contents

Abstract xi

1. Introduction 1
1.1. FPGA Hardware Acceleration . . . 2
1.1.1. Benefits of FPGAs for the Implementation of Hardware Accelerators . . . 3
1.1.2. FPGA Architecture . . . 3
1.1.3. FPGA System Architectures . . . 5
1.1.4. Electronic System-Level Design . . . 6
1.1.5. Legacy FPGA Design Flow . . . 7
1.1.6. High-Level Synthesis for the Implementation of FPGA-based Hardware Accelerators . . . 9
1.1.7. Board-Level Execution and Simplified Deployment . . . 12
1.2. Rapid Prototyping of FPGA Accelerators . . . 13
1.3. Contributions . . . 14
1.4. Organization . . . 17

I. Domain-Specific High-Level Synthesis for Image Processing IP Cores 19

2. Medical Image Processing 23
2.1. Digital Image Processing . . . 23
2.2. Operations in Medical Image Processing . . . 24
2.3. Medical Image Enhancement and Denoising Techniques . . . 25
2.3.1. Spatial Operators . . . 26
2.3.2. Image Processing Algorithms . . . 28
2.4. Accelerators in Medical Image Processing . . . 31
2.5. Summary . . . 31

3. Automatic Hardware Generation through High-Level Synthesis 33
3.1. Early efforts on the Path to High-Level Synthesis 34
3.2. Current Tools 38
3.2.1. General Purpose Language Support 40
3.2.2. Restricted Language Support 41
3.2.3. Miscellaneous Language Support 43
3.2.4. Domain-Specific Languages for HDL Generation 44
3.2.5. Code Generation 46
3.2.6. Execution Environments 47
3.3. Vivado HLS as an Example for C-based High-Level Synthesis . . 48
3.3.1. High-Level Synthesis 48
3.3.2. The Vivado HLS Design Flow 50
3.3.3. Unsupported C Language Constructs 51
3.3.4. Arbitrary Precision Data Types 51
3.3.5. Interfaces 52
3.3.6. Optimization and Synthesis Guidance 52
3.4. Conclusion 55

4. Domain-Specific High-Level Synthesis for Image Processing on FPGAs 57
4.1. Preliminaries 58
4.2. Importance of Hardware-specific Coding Styles for C-based HLS 60
4.3. Stream-based Image Processing on FPGAs 63
4.3.1. Motivational Example 63
4.3.2. Memory Architecture 64
4.3.3. Causality 65
4.3.4. Border Handling 66
4.3.5. Filter Assembly 67
4.3.6. Library Integration 68
4.4. Specification of Point and Local Operators 69
4.4.1. Point Operators 69
4.4.2. Local Operators 70
4.5. Parallelization and Design Optimization 72
4.5.1. Vivado Synthesis Directives and Coding Considerations . 72
4.5.2. Optimizing Loop Counter Variables 73
4.5.3. Mapping Vector Types 74
4.5.4. Optimizations for Streaming Pipelines 75
4.6. Evaluation 75
4.6.1. Evaluation Environment 76
4.6.2. Algorithms 77
4.6.3. Comparison with OpenCV 78
4.6.4. Comparison to Other Accelerator Technologies 81

4.7. Image Pyramids for Multiresolution Analysis 84
4.7.1. Up- and Downsampling 84
4.7.2. Library Integration 85
4.7.3. Pyramid Construction 87
4.7.4. Optimization of Buffer Requirements 89
4.8. Evaluation of Pyramid Algorithms 92
4.8.1. Pyramid Optimizations 92
4.9. Summary 94

5. Beyond Instruction-Level Parallelism 97
5.1. Spatial and Data-Level Parallelism 97
5.2. Loop Tiling 99
5.2.1. Overlap Area for Local Operators 100
5.2.2. Implementation Structure 101
5.2.3. Asymmetric Image Regions 105
5.2.4. Loop Tiling for Streaming Pipelines 106
5.2.5. Library Integration 108
5.2.6. Evaluation 111
5.3. Loop Coarsening 118
5.3.1. Differences to Vector Operations 119
5.3.2. Superpixel and Superwindow Concept 119
5.3.3. Data Separation and Border Treatment 119
5.3.4. Library Integration 121
5.3.5. Evaluation 121
5.4. Comparison and Discussion 123
5.4.1. Resource Requirements and Performance 124
5.4.2. Speedup 125
5.4.3. Comparison to Software-based Accelerators 125
5.5. Summary 127

II. Interface Synthesis for FPGA-based SoC Integration of Image Processing IP Cores 129

6. An On-Chip Interconnect for System Integration 133
6.1. Generic Interfaces in Vivado HLS 134
6.1.1. Block-Level Interfaces. 135
6.1.2. Function Argument Interfaces 136
6.1.3. Creating Custom Protocol Interfaces in Vivado HLS . . . 139
6.2. AXI4 Interfaces 143
6.2.1. AXI4 Memory-Mapped Interfaces 143
6.2.2. AXI4-Stream Intefaces 144

6.2.3. Support for AXI4-Interfaces in Vivado HLS 145
6.2.4. AXI4S Interconnect 147
6.3. DDR3 Memory Abstraction Layer 148
6.4. Case Study: Intermediate Stream Buffering for Multiresolution Analysis 152
6.5. Summary 157

7. Board to Host Communication **159**
7.1. The PCI Express Bus Architecture 160
7.1.1. Introduction to PCI Express 160
7.1.2. FPGA Support for PCI Express 161
7.2. Host Coupling of Field Programmable Gate Array (FPGA)-based Accelerators 163
7.2.1. PCI Express and DMA 164
7.2.2. AXI4S Interconnect and DDR3 Abstraction 164
7.2.3. Auxiliary Components 166
7.2.4. Clock and Reset Infrastructure 167
7.3. Software 167
7.4. Summary 169

8. Board to Board Communication **171**
8.1. The RapidIO Interconnect Standard 172
8.1.1. Technological Overview 172
8.1.2. The Logical Layer 176
8.1.3. The Transport Layer 178
8.1.4. The Physical Layer 179
8.1.5. Serial RapidIO on FPGAs 185
8.2. Power Management Strategies for Serial RapidIO (SRIO) 186
8.2.1. Methodology 188
8.2.2. Experimental results 194
8.3. SRIO User Application 199
8.3.1. Autonomous Packet Transfer 199
8.3.2. Transaction Configuration 200
8.3.3. Stream Multiplexing 200
8.4. Integration of SRIO into the AXI4S-based FPGA Interconnect . 200
8.5. Summary 201

9. Rapid Prototyping of FPGA Accelerators **203**
9.1. FPGA Support Systems 204
9.1.1. Incremental Compilation 205
9.1.2. Out-of-Context System Design 206

9.2. A Fully Automated Rapid Prototyping Design Flow 207
9.2.1. Develop 207
9.2.2. Evaluate 208
9.2.3. Deploy 209
9.3. Summary 209

10. Conclusions and Future Work **211**
10.1. Summary 211
10.2. Future Work Directions 212

German Part **215**

Bibliography **223**

Author's Own Publications **239**

List of Acronyms **243**

Abstract

The tremendous technological progress in recent years has enabled Field Programmable Gate Arrays (FPGAs) to become very complex devices. In comparison to other popular accelerator technologies, such as Graphics Processing Units (GPUs), they may often offer *competitive performance* at a *higher energy efficiency*. Despite of many achievements in *electronic design automation*, a considerable part of the FPGA design process remains cumbersome and may elude inexperienced developers from choosing FPGAs over other accelerator technologies. In order to allow engineers with little or no prior hardware design experience to develop hardware accelerators for FPGA-based System-on-a-Chip (SoC) designs, this work proposes relevant contributions to *system-level design* primarily in the fields of

(a) domain-specific High-Level Synthesis (HLS), and

(b) interface synthesis for FPGA-based SoC integration

of image processing Intellectual Property (IP) cores.

Most of the currently available HLS frameworks allow design entry using high-level programming languages, such as C++. Nevertheless, achieving efficient and high-performance synthesis results still demands an in-depth understanding of FPGA design principles. In many cases it is even required to be familiar with parallelization techniques to guide the hardware synthesis process. For these reasons, many of the available tools are not yet suitable for developers without the relevant experience.

To mitigate this situation, this work proposes a lightweight library of basic code templates for medical image processing that allows the generation of hardware accelerator IP cores using HLS without requiring expertise in how to design the actual hardware implementation. Algorithms in this application domain are often a sequence of basic image processing operations, such as point, local, and global operators. Instead of providing complete algorithms, the library contains building blocks for these basic operators, as well as design elements for stream-based interconnection to facilitate the assembly of filter pipelines, as required for the implementation of complex algorithms. Directives to guide the

synthesis process are already contained in the source code of the library and can be accessed easily from the top level without having to search a vast body of code. This modular structure allows designers to easily extend the library with new features, but also to perform design space exploration and tune the synthesis to achieve specific design goals, such as a high performance or low resource requirements. Since the library is written in C++, initial development and testing of new specifications can be carried out entirely in a familiar, software-based environment and requires neither knowledge of hardware design nor the necessary Computer Aided Design (CAD) tools. Another advantage of this design approach is a reduction of the development time, since testing and error localization is achieved significantly faster by running software than at the register-transfer-level using logic simulation.

We have assessed the quality of the hardware synthesis results by using the library to generate accelerators for several typical algorithms for medical image processing. We show that in comparison to other libraries, our approach can achieve a higher throughput and a more efficient exploitation of the resources available on an FPGA. Furthermore, we have also compared the generated FPGA accelerators to highly optimized implementations for server-grade and embedded GPUs, where we can achieve a higher throughput, while at the same time, provide a substantially more energy-efficient solution.

As an FPGA is initially unprogrammed, actually employing a HLS-generated image processing IP core requires to integrate it into its on- and off-chip environment. In order to prepare the automation of on- and off-chip integration, this work is also concerned with *interface synthesis* for FPGA-based SoC integration of accelerator IP cores.

For on-chip integration, we propose an On-Chip Interconnect (OCI) that is built on a standardized bus specification and provides support for data streaming, as well as memory mapped components. Data type conversion and clock domain crossing between connected components is hereby handled inside of the OCI. In this way, a synthesized accelerator can be integrated into an SoC design without having to comply with any constraints imposed by other modules. This ensures interoperability across a wide range of vendor-supplied design modules and also supports the specification of custom protocols to communicate with user-specific components.

Furthermore, we propose solutions for the integration of the SoC design into its off-chip environment, which might include a host computer or a distributed embedded system. For coupling an FPGA-based accelerator to a host computer, we establish connectivity between an image processing IP core and a Peripheral Component Interconnect Express (PCIe) communication controller via the OCI of an FPGA-based SoC design. For this purpose, we have developed a Linux-based software interface and driver, in order to exchange data between a software program and an accelerator in a *hardware/software co-design*, using

PCIe for communication. To integrate an image processing accelerator in a distributed embedded system, we facilitate data exchange in the absence of software control using the Serial RapidIO (SRIO) protocol as an example for high-speed Serializer/Deserializer (SerDes) communication. Here we develop an SRIO communication controller that is augmented with a custom user application to *autonomously generate packets* for streaming-based application data. The developed communication endpoint can be seamlessly integrated into an OCI for FPGA-based SoC designs.

Based on the presented preparations for automating system-level synthesis, we propose a holistic, fully automated prototyping procedure for the development of FPGA-based image processing accelerators comprising the following three major steps:

1. A functional specification of the accelerator is developed in a high-level programming language, using the proposed library as a high-level abstraction. The specification can be executed as part of a software program in oder to evaluate and optimize it with respect to the design goals. Once the specification fulfills the objectives, a hardware description in form of an IP core is automatically generated using HLS.

2. For board-level evaluation, the generated IP core is integrated into an FPGA-based SoC design and implemented on an FPGA development board that is connected to a host-computer via PCIe. As part of a hardware/software co-design, the generated IP core is evaluated and verified by a software-program running on the host-computer.

3. Finally, the synthesized IP core is evaluated under real-world conditions, for example as part of a product. For this, the IP core is integrated into an FPGA-based SoC for system-level evaluation that provides connectivity to the target system, for example, via the SRIO protocol.

The presented approaches for domain-specific HLS and interface synthesis considerably contribute to the automation of system-level design and therefore make the FPGA-platform approachable also for software developers who might lack any hardware design experience. The proposed high-level abstractions in form of a library allow the intuitive development of functional specifications for HLS in a high-level programming language. This facilitates (a) software-based localization and elimination of programming errors, (b) rapid design space exploration, (c) optimization with respect to specific design goals, and may thus remarkably expedite the design cycle. The contributions for interface synthesis allow the rapid integration of a generated IP core into FPGA-based SoC designs in order to perform (a) software-based board-level evaluation and verification as part of a hardware/software co-design, and (b) system-level evaluation for

validation of system-level design decisions. In this way, we contribute a holistic and fully automized approach for rapid prototyping of hardware-accelerators that may increase productivity significantly. Although we concentrate on medical image processing in this work, the contributions are also applicable to many other application areas.

1 Introduction

Until around the year 2002, computer architects could rely on Moore's law promising that the performance of single core general-purpose processors would double every eighteen to twenty-four months [Moo65]. Unfortunately, the unlimited increase in performance was brought to a sudden halt by two painful discoveries:

1. Power is *not* for free.
2. Even very tiny transistors might need a lot of it.

This phenomenon is commonly known as the *power wall* and essentially meant that the old and very comfortable paradigm of *smaller and faster* did not hold anymore, since raising the clock frequency any higher would cause a dramatic surge in power demands. It was also envisioned that the tremendous increase in computational efficiency would allow compilers to extract more and more Instruction-Level Parallelism (ILP) that could be exploited by using deeper instruction pipelines and putting more effort on speculative execution in future Central Processing Units (CPUs). Sadly, also these endeavors where stopped, as the returns on finding new ILP could only be described as diminishing [HP14], which came to be known as the *ILP wall*. The new paradigm of *smaller and more parallel* took over, but soon, it had to be admitted that the *memory wall*, describing the increasing gap between processor performance and memory access time, is a limiting factor that would not allow multi-core parallel computing to achieve the tremendous increase in performance that was initially anticipated. As the well-known computer scientist David Patterson describes in [Asa+06], the combination of these three walls closely resembles a *brick wall* for computing and trying to break through one of the walls will only worsen the effects of the remaining walls.

In consequence, it became clear that the only way to further increase performance would ultimately have to involve concepts that would allow to *exploit parallelism more efficiently*. One of these concepts is *hardware acceleration*, which describes the use of computer hardware to perform a specific function faster than is possible in software running on a general-purpose CPU. The

hardware that is used to perform the function instead of the CPU is called the *hardware accelerator*. The use of hardware accelerators in computing dates back until to the early 1980s, when Intel proposed the 8087 chip in addition to its 8086 processor as a way to boost the performance of floating-point math operations. Meanwhile, a plethora of different variations exist, ranging from full-custom Application-Specific Integrated Circuits (ASICs) to fine-grained reconfigurable devices (Field Programmable Gate Arrays (FPGAs) [BR96]) and also architectures incorporating arrays of coarse-grained reconfigurable and programmable elements [MD96; Mei+03; Bau+03; Han+14].

Undoubtedly, the highest performance and energy efficiency can be achieved if a certain functionality can be accelerated by a full-custom ASIC. ASIC development involves complex Very Large Scale Integration (VLSI) design [Cha11] and must also take many practical impairments into account, including crosstalk, leakage, and voltage drops, amongst others [JG93]. These circumstances make ASICs only profitable if a very high amount of produced chips can amortize the high design cost. Consequently, a prime domain for implementing hardware accelerators in the form of ASICs are high-volume consumer products, such as mobile phones. A disadvantage is, however, that an ASIC can only perform a specific, fixed functionality. Other hardware accelerator technologies are more attractive if only a low volume of products is needed or programmability is required. In addition to wide-spread software-based accelerators, such as, Graphics Processing Units (GPUs) and Digital Signal Processors (DSPs), FPGAs are one programmable alternative to ASICs. Having advanced significantly since their initial introduction, they meanwhile carry enough resources to be able to implement complex embedded systems on a single device and are even migrating into the mobile accelerator market [Wan+14]. The major advantage of FPGAs over ASICs is their reconfigurability, which, due to developments of the last decade, is also feasible during runtime. A drawback in comparison to custom logic is, however, that they carry overhead in cost, speed, and power consumption. However, due to the increasing Non-Recurring Engineering (NRE) costs of ASIC design, FPGAs are applied in an ever increasing number of applications.

1.1. FPGA Hardware Acceleration

Out of the abovementioned programmable hardware accelerators, FPGAs can deliver high processing speeds and high energy efficiency. Unfortunately, they are also the least attractive platform for software programmers, since the design takes place at a very low abstraction level and requires the use of special Computer Aided Design (CAD) tools for logic synthesis and subsequent hardware implementation. The design process might therefore become quite lengthy which drives up development costs. In the presence of high-performance hardware

accelerators that can be employed with only little difference to general-purpose computing on a CPU, FPGAs provide distinct benefits in comparison to software-based accelerators.

1.1.1. Benefits of FPGAs for the Implementation of Hardware Accelerators

The world of software-based accelerators mostly owes its success and widespread use to consumer driven markets. For example, the rapid development of GPU performance mostly stems from the high demands of the computer gaming industry and the currently available highly sophisticated embedded accelerators are often the result of the ever growing consumer demands for mobile gadgets. A downside for industrial applications is that this rapid development is entailed by a relatively short life cycle of the available accelerators. In contrast, the development of FPGAs is mostly driven by industrial applications and therefore also provides a very *long life cycle* of the products. Another often mentioned benefit is that FPGAs can provide *deterministic throughput and latency*, if the application allows it. Whereas software implementations must rely on complicated protocol stacks and are often subject to scheduling of resources carried out by an underlying operating system, the latency of computations on FPGAs can be *determined precisely*, since it is not obstructed by a scheduler. Moreover, this applies to the *massive parallelism* of the architecture, as well. FPGAs may furthermore achieve a significantly better energy efficiency than software-based accelerator technologies [KC05; Sch+15a*; Rei+14*] and provide a high degree of flexibility when it comes to the support of interconnect technology. Especially for medical image processing equipment, which is subject to very strict government regulations (for example, refer to the US Food And Drug Administration regulations, Section 510(k)[1]), the FPGA platform brings the benefit of an easier system validation for approval by government agencies. Software implementations require a high amount of code reviews, verification of the individual components, as well as verification of the assembled application executing on the targeted hardware for approval. Conversely, FPGA implementations are considered as highly reliable hardware and can thus be validated as a whole, however, they require a much shorter development time than ASICs.

1.1.2. FPGA Architecture

As shown in Figure 1.1, modern FPGAs, such as the 7-series from Xilinx [Xil14b], comprise a mixture of different resources, including Configurable Logic Blocks (CLBs) (containing Static Random Access Memory (SRAM)-based Lookup

[1] http://www.fda.gov/MedicalDevices/DeviceRegulationandGuidance/Overview

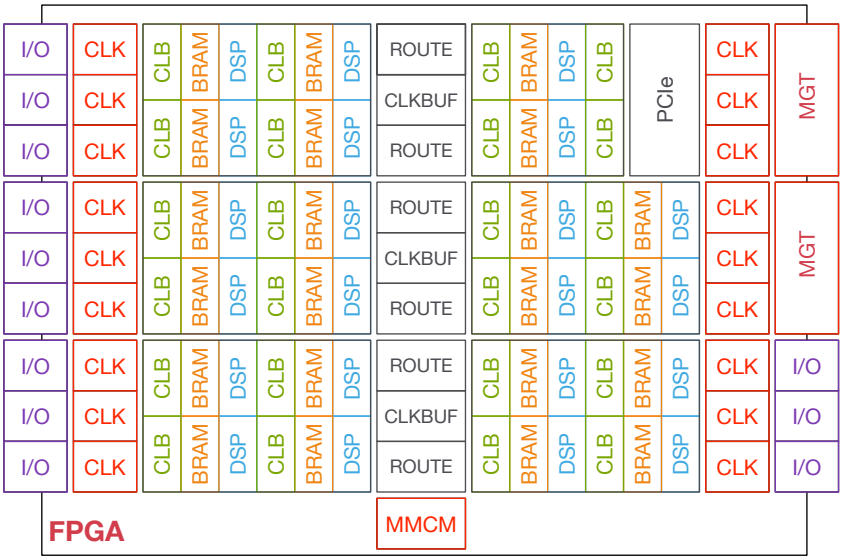


Figure 1.1.: Column-like architecture of a modern FPGA, consisting of different resources, such as CLBs, BRAMs, and DSPs. In addition to parallel I/O, modern architectures also support high-speed serial I/O via MGTs. Routing resources (ROUTE), as well as clock managers (MMCM) and buffers (CLKBUF) are typically located at the central column.

Tables (LUTs)), a constantly increasing amount of Dual-Ported Random Access Memory (DPRAM) (organized in blocks as Block Random Access Memory (BRAM)), and arithmetic blocks (DSP). The resources are arranged in columns and distributed across the chip in a striped pattern, which is especially intended to facilitate high-frequency pipelined designs. So-called clock regions may contain up to 50 CLBs, which are organized in *slices*. Slices contain several multi-input LUTs, Flipflops (FFs), and multiplexers for signal routing. In addition to implementing reconfigurable logic, slices might may also be used as shift registers or distributed Random Access Memory (RAM). BRAMs are typically 32kb in size and contain error correction and sometimes also logic to be used as a First In, First Out (FIFO) memory. DSPs slices, consisting of a multiplier and an adder, are specialized Multiplier-Accumulator (MAC) circuits. Most recent DSP slices are optimized for pipelining and also contain circuits for pattern matching. In addition to parallel I/O pins, FPGAs typically also contain several Multi-Gigabit Transceivers for high-speed serial I/O, which can support a wide range of Serializer/Deserializer (SerDes)-based protocols through hard blocks, for example for Peripheral Component Interconnect Express (PCIe) and Ethernet, or through soft Intellectual Property (IP) cores, such as, Serial RapidIO (SRIO) [Ful05] and

Interlaken [GOW07]. Clocking resources are typically located closely to the I/Os to minimize clock jitter and allow high frequencies. The functional blocks and the I/Os are interconnected by a tightly interwoven fabric of routing and signaling resources. The central column typically includes routing resource (ROUTE) between the clock regions, the Mixed-Mode Clock Managers (MMCMs) for clock generation, as well as global and local clock buffers (CLKBUF). FPGAs are programmed using a bitfile that configures the contained resources, as well as the switching points of the interconnect.

1.1.3. FPGA System Architectures

Modern FPGAs have become highly flexible and incorporate a very large amount of logic cells, which allows them to implement complex functionality. They are also often combined with embedded processors in the form of hard blocks, such as the PowerPC [Xil11c] or, more recently, ARM Cortex cores [Xil14c; Alt15a], but also allow to implement soft IP based processors [Xil15b; Alt15b]. Despite of these capabilities, they are initially unprogrammed, meaning that the designer does not only have to design a circuit to implement the actual accelerator, but also the system architecture. Such architectures are quite similar to so-called System-on-a-Chip (SoC) designs and typically consist of processing elements, for example embedded processors or accelerators, memory, and I/O interfaces, as illustrated in Figure 1.2. The elements are connected via an On-Chip Interconnect (OCI) that could be a simple bus architecture or an advanced Network-on-Chip (NoC) [Pan+05]. Due to the amount of functional units to

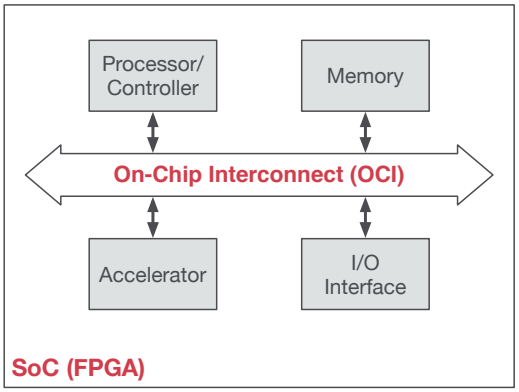


Figure 1.2.: FPGA system architectures are very similar to System-on-a-Chip designs and typically consist of embedded processing elements (processors, accelerators, etc.), coupled with memory and I/O interfaces via an On-Chip Interconnect.

integrate as well as the heterogeneity of the components, these designs constantly increase in complexity. To handle such high complexity, the preferred design strategy is to start specifying the system at the *Electronic System-Level* (ESL).

1.1.4. Electronic System-Level Design

ESL design is often mentioned as the key design strategy to be able to handle the *crisis of complexity*, meaning the *increasing gap between silicon technology and SoC design complexities* [Hen03]. Bailey, et al., define it as *the use of appropriate abstractions to increase comprehension of a system and enhance the probability of successfully implementing its functionality in a cost-effective manner, while meeting necessary constraints* [BMP07]. The design methodology hereby follows a top-down approach, where more and more detail is added to a specification on a high abstraction level on its way to a lower abstraction level. Closely related to Electronic System-Level (ESL) design is the so-called *double roof* model, proposed by Teich in [Tei97], which is shown in Figure 1.3. The two *roofs* denote the *behavioral* and the *structural* view. The structural representation is obtained from the behavioral description through *synthesis*, which is specified by Teich in [Tei00] as follows: *Synthesis is the refinement of a behavioral specification into a structural specification at a certain abstraction level. The main synthesis tasks are independent from the level of abstraction and may be classified as [Tei97]:*

- allocation *of resources*
- binding *behavioral objects to allocated structural objects, and*
- scheduling *of behavioral objects on the resources they are bound to. A schedule may be a function that specifies the absolute or relative time interval, a behavioral object is executed, or just an order relation for the execution of several objects (complete order or partial order, priorities, etc.).*

According to the double roof model, designing for FPGAs is situated on the right side of the model. Below the electronic system-level, the general levels of abstraction are the *architecture* and the *logic level*. At the ESL, the complete system is described by interconnected subsystems, which represent, for example, algorithms, or system tasks, such as communication. The architecture level comprises communicating functional blocks that carry out complex arithmetic or logic operations. At the logic-level, the components possess a rather high amount of detail, as they are described by interconnected gates and registers to compute boolean equations. The Electronic Design Automation (EDA) tool *SystemCoDesigner* [Hau+07; TH07], for example, realizes the double roof approach to system-level design. It performs automatic design space exploration and generates platform-based prototypes of (mixed) hardware/software systems.

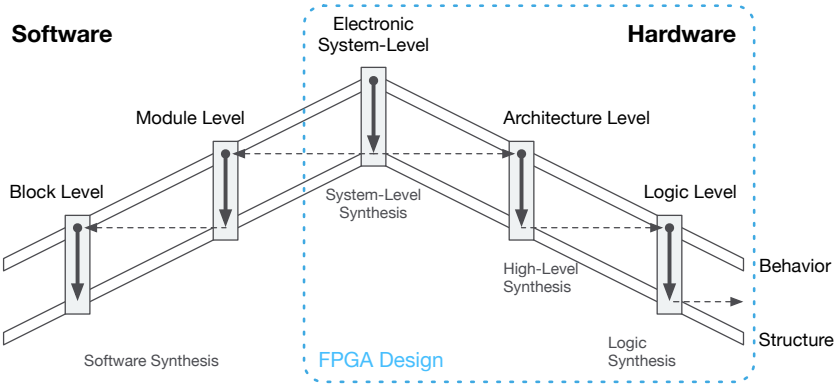


Figure 1.3.: Abstraction levels for embedded system design, adapted from Teich's *double roof* model [Tei97]. The two roof-like layers describe the behavioral and the structural view. In this work, we mostly consider the right part of the model, which describes hardware design.

In general, the lower the abstraction level, the more intensively it has so far been studied and sophisticated tools exist to automate the synthesis process (refer to Chapter 3). An alternative approach to synthesis is *platform-based design*, which is a rather integration-oriented approach for system design that focusses on systematic reuse of previously designed components, so-called IP cores [BMP07]. Most major Original Equipment Manufacturers (OEMs) meanwhile accompany their products with IP core libraries that include optimized implementations of buildings blocks for a wide range of application areas. However, integrating a complex IP core into a custom system architecture might still require a lot of design experience. In order to reduce development costs, it has nevertheless become an often used strategy in EDA to combine both approaches for system design, that is, to make use of IP cores for commonly used components of the system architecture, whereas new and custom components are developed according to the process described by the double roof model.

1.1.5. Legacy FPGA Design Flow

A very important factor for the high complexity of the design task is the very cumbersome design flow for developing FPGA hardware. Although the process is significantly faster than an ASIC tape-out, it still contains enough discouraging elements to draw non-experts away from FPGAs. Figure 1.4 depicts the individual development stages a design has to traverse from an initial *functional specification* to the final FPGA design in form of a bitfile. A more detailed treatment of the design flow is, for example, provided by

Chu in [Chu06]. If we omit the initial phase of transforming an algorithm

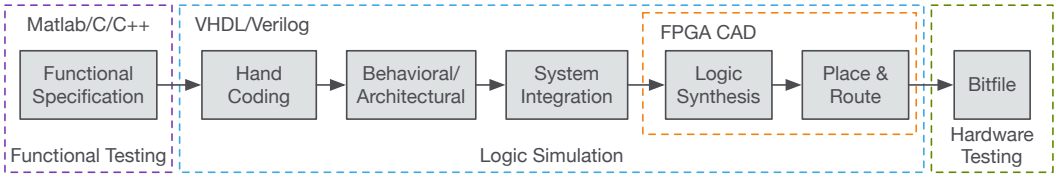


Figure 1.4.: Individual stages of the legacy FPGA design flow from the functional specification to the implementation in form of a bitfile.

into a software implementation, we can start with a *functional specification*, given in a programming language, such as C [ISO11], or C++ [ISO14], or as a Matlab program [The15]. The functional specification captures all aspects of the functionality of the design at a high abstraction-level that does not include timing information [BMP07]. The functional specification moreover has the benefit that it can be validated and verified using very fast *functional simulation*.

The first step on the design's path to become a hardware implementation is that the functional specification must be transformed into an *architectural specification*. This means converting *what the design does* into *how the design looks like to achieves it*. The de facto standard for developing hardware designs still involves hand coding using Hardware Description Languages (HDLs), such as VHDL [IEE09] or Verilog [IEE04]. As the languages are intended also for validation and verification of a described design, they both also contain a behavioral part to ease test bench specification. Whether the behavioral part of the language can be handled during behavioral synthesis depends on the synthesis tool. Moreover, it is also often a tradeoff between how much detail the designer wants to be able to preserve for system verification after synthesis, since every part of a design that should be observed has to be made available explicitly at one of the system interfaces. Hand coding using an HDL requires a completely different skill set than high-level programming, may take a very long time, and might be error prone. Furthermore, the resulting Register Transfer Level (RTL)-description of the system can only be tested and validated using *logic simulation*, which is significantly slower than testing at a high abstraction level.

Once the functional aspects of the design have been captured as an architectural specification, the next problem in line is that *FPGAs are initially unprogrammed* and only developing the solution to a problem usually does not suffice to actually employ the FPGA in a system. Instead, an SoC-like system architecture, as described in Section 1.1.3, must be designed and the previously developed structural description of the algorithm must be *integrated into the system architecture*. Furthermore, the interaction between the developed component and the system

architecture must be tested and verified using logic simulation. Validation and verification of communication controllers and peripheral components require complex simulation models and might therefore cause a long simulation time.

After the entire design has been developed and tested, it has to undergo the typical *FPGA implementation procedure*. The individual steps include *logic synthesis*, which transforms the hardware description typically into a netlist of logic elements and flipflops with optimization of either logic depth or number of gates, which impact the maximum achievable clock frequency or required amount of hardware resources, respectively. Subsequently, the phase of *technology mapping* maps the optimized netlist to the programmable logic blocks of the target FPGA device. Finally, during *placement and routing*, the logic blocks are actually placed on the target device and the wires to interconnect the logic blocks are routed using the available resources on the target FPGA. Several problems may emerge during this process, for example, the device resources, such as CLBs, BRAMs, DSPs, or signal routes may be insufficient to fit the design on the FPGA or the current architecture may not be able to achieve a certain clock frequency[2]. Coping with these issues requires *in-depth knowledge of FPGA CAD tools* and may even require design alterations. It is moreover considered good practice to simulate the design after every step of the design flow to verify its correctness. As the abstraction level decreases, the necessary time for logic simulation also increases, since the simulation must perform more operations at a more fine-grained wall-clock resolution, which also adds to the development time. After the completion of the FPGA CAD tools, the resulting bitfile can be used to configure the FPGA.

It becomes evident that the task of developing an FPGA solution, from the initial specification until it is ready for deployment, can become very complex and might take substantially longer than developing a software implementation. Furthermore, developing for FPGAs requires proficiency in high-speed RTL design and the associated tools. Tasks, such as, clock-domain transition, floor planning, and creating timing constraints are very different from the necessary knowledge for software programming and software engineers might not be familiar with them.

1.1.6. High-Level Synthesis for the Implementation of FPGA-based Hardware Accelerators

A first step towards making an FPGA more approachable for inexperienced designers is to provide a familiar development concept. High-Level Synthesis

[2]Satisfying all the timing constraints imposed on a digital circuit in order to meet a specific clock frequency is often referred to as *timing closure*. A circuit for which timing closure has been achieved is also referred to as being *timing closed*.

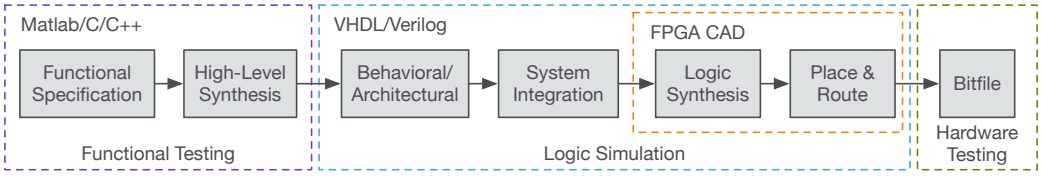


Figure 1.5.: Using HLS to synthesize an architectural description from a functional specification, hand coding may become obsolete for certain components of the SoC architecture, such as accelerators, and therefore might simplify the legacy FPGA design flow.

(HLS) [TH07; De 94] describes the automatic synthesis of a hardware architecture from a high-level specification and aims at making hand coding at the Register Transfer (RT) level obsolete. Figure 1.5 illustrates that HLS can replace hand coding in the legacy design flow for FPGA designs. Whereas initial approaches only allowed behavioral design specifications, for example in behavioral VHDL, recent HLS tools allow synthesis from a functional specification, for example in C-based programming languages, such as C and C++. Hereby, the HLS tools must perform allocation, binding, and scheduling for the specified algorithm but also for the interface. To briefly give and idea of how HLS achieves this, we show a very simple example of a function to be synthesized in Listing 1.1.

Listing 1.1: Simple function for architecture synthesis using HLS.

```
1  void foo(float &a, float &b, float &c)
2  {
3    c = (5.0*a + 7.0*b)/3.0;
4  }
```

According to the representation of the function `foo` as a Data Flow Graph (DFG) (refer to the left of Figure 1.6), we must allocate one or multiple multipliers, an adder, and a divider that support single-precision floating-point arithmetic, as well as several registers to hold the constant values for the implementation of the data path. A possible schedule and binding for an allocation of one multiplier (FPMul), one adder (FPAdd), and one divider (FPDiv) is shown in Figure 1.6. A possible accelerator for the function `foo` is shown in Figure 1.7, consisting of the *data path*, as well as the *control path*, which may be realized as a Finite State Machine (FSM) [TH07]. In combination with the data path, such a structure is usually referred to as a Finite State Machine with Data Path (FSMD).

It becomes obvious that even very simple specifications, such as given in Listing 1.1 can have several formally correct solutions to the synthesis problem. Therefore, most modern HLS tools require user guidance for synthesis, for instance, Vivado HLS from Xilinx [Xil15c] requires the user to specify synthesis directives to control allocation, binding, and scheduling. Moreover, despite of the

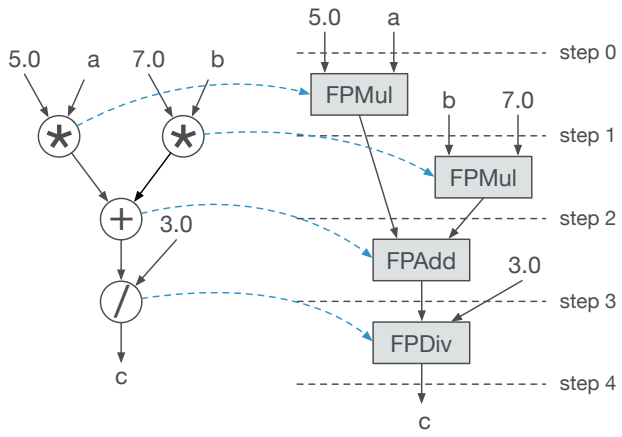


Figure 1.6.: DFG of the function `foo` of Listing 1.1, as well as a schedule and binding for an allocation of one multiplier (FPMul), one adder (FPAdd), and one divider (FPDiv).

fact that HLS can significantly shorten the development time by automatically transforming a functional into an architectural specification, a vital aspect is that although the entry point for C-based HLS looks like a software implementation, it is far from it. In order to achieve a high-quality circuit implementation that can rival highly optimized, hand-crafted RTL implementations in terms of latency,

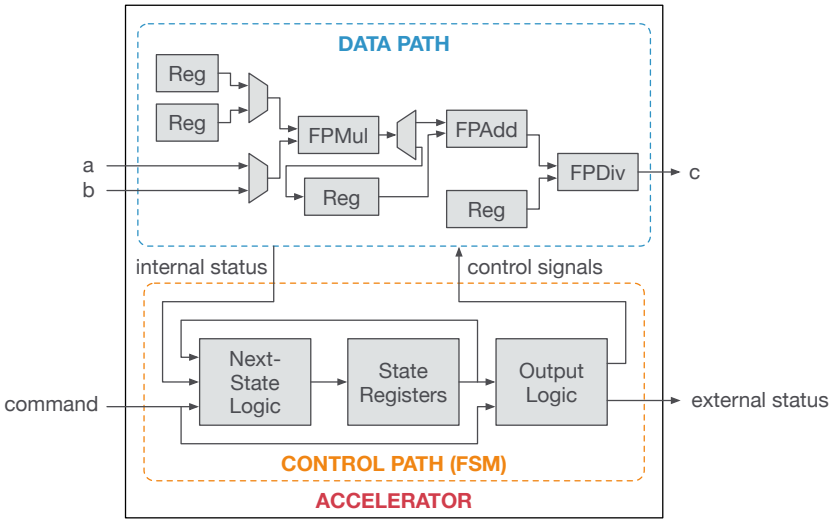


Figure 1.7.: Accelerator for the `foo`-function of Listing 1.1 implemented as a Finite State Machine with Data Path (FSMD).

maximum clock frequency, and resource requirements, the designer must describe the actual hardware architecture to be synthesized in the program. To mitigate this problem, we propose a lightweight library of generic classes, methods and functions for domain-specific HLS of image processing accelerators for FPGA targets [Sch+14a*]. Since the library is written in C++, it can be used as a high-level programming abstraction for code development on a CPU. In the context of HLS, the library yields highly efficient hardware implementations that can rival hand-optimized RTL and even outperform software-based accelerators, for example GPUs [Sch+15a*; Rei+14*].

1.1.7. Board-Level Execution and Simplified Deployment

Another key factor to make FPGAs more attractive to software programmers is to provide a familiar, software-like execution environment for evaluation and provide a means for deployment of the generated accelerator that minimizes the interaction with FPGA CAD tools. As the result of HLS is an IP core that often only implements the synthesized function, this step poses several challenges at once.

1. Since FPGAs are initially unprogrammed, evaluation on actual hardware requires the design of system components for connectivity and possibly also the corresponding software counterparts.

2. FPGA implementation with CAD tools might become very difficult. Especially off-chip communication imposes stringent timing constraints that are easily violated and difficult to fix.

To mitigate these problems, we propose the use of pre-designed FPGA *support systems* that provide the system architecture, as shown in Figure 1.8, for testing

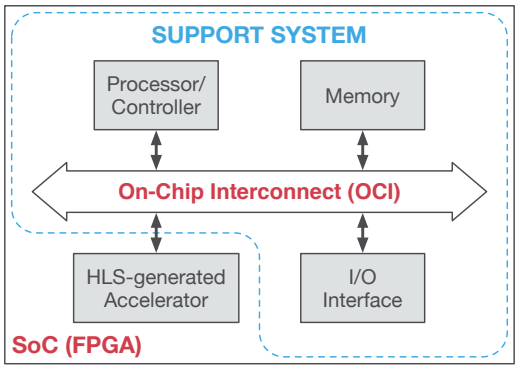


Figure 1.8.: An FPGA support system provides a system architecture to test a developed accelerator.

the developed accelerator in a host-based environment, but also to allow simplified deployment in the context of an actual embedded systems framework. The interaction with FPGA CAD tools can be minimized by using a hierarchical design flow, where the implementation and timing closure of the components of the support system can be achieved independently of the IP core produced by the HLS tools, before the design is handed over to a designer. The IP core can then be inserted at a later time and the remaining implementation task will not affect the components of the system architecture.

1.2. Rapid Prototyping of FPGA Accelerators

To encapsulate the individual steps and solutions to make an FPGA platform more attractive to developers with little to no prior FPGA experience, this thesis proposes a new and unique rapid prototyping flow, as depicted in Figure 1.9. It can generate, evaluate, and deploy FPGA accelerators for a substantial set of

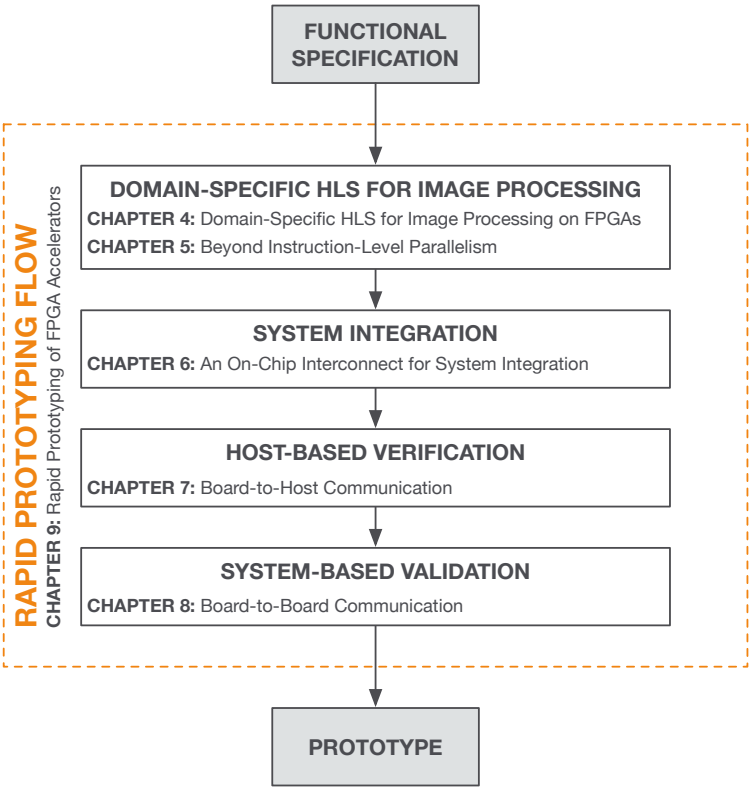


Figure 1.9.: Overview of the proposed flow for rapid prototyping of FPGA accelerators.

algorithms from the domain of medical image processing. The prototyping flow hereby consists of the following three major steps:

1. A functional specification of the accelerator is developed in a high-level programming language, using the proposed library as a high-level abstraction. The specification can be executed as part of a software program in oder to evaluate and optimize it with respect to the design goals. Once the specification fulfills the objectives, a hardware description in form of an IP core is automatically generated using HLS.

2. For board-level evaluation, the generated IP core is integrated into an FPGA-based SoC design and implemented on an FPGA development board that is connected to a host-computer via PCIe. As part of a hardware/software co-design, the generated IP core is evaluated and verified by a software-program running on the host-computer.

3. Finally, the synthesized IP core is evaluated under real-world conditions, for example as part of a product. For this, the IP core is integrated into an FPGA-based SoC for system-level evaluation that provides connectivity to the target system, for example, via the SRIO protocol.

1.3. Contributions

Providing energy-efficient solutions for the acceleration of compute-intensive tasks is gaining more and more importance. Due to their tremendous processing capabilities while having extremely low energy requirements, FPGAs and reconfigurable technology can make a significant contribution in this area. However, the currently available development methodologies require a lot of domain-specific hardware experience to achieve high-quality results, in terms of throughput and resource requirements. Moreover, despite the fact that modern HLS tools can be used to synthesize individual components of an SoC-like FPGA design, actually using these on an FPGA to accelerate a task also requires the remaining components of the architecture for communication and accessing external memory, among others. Unfortunately, the design of the system architecture and the integration of the developed accelerator into the architecture must still be done by hand. Developing new design flows and development tools to make the FPGA platform more accessible to software programmers, who lack the appropriate hardware design experience, is therefore a very active research area. A recent example for the importance of the topic is the *FPGAs for Software Programmers* Workshop, co-located with the *International Conference on Field Programmable Logic* (FPL) in 2014 [HKZ14].

The primary goal of this dissertation is to make the FPGA platform more approachable for developers with little to no prior hardware design experience, thereby neither sacrificing any optimality of the performance nor any requirements for image processing algorithms on FPGA targets. To achieve this goal, the main contributions of this thesis are situated at the ESL and represent the necessary preparations for automating the subsequent synthesis steps at the system- and architecture-level. In the fields of *Domain-Specific High-Level Synthesis for Image Processing IP Cores* as well as *Interface Synthesis for FPGA-based SoC Integration of Image Processing IP Cores*, the main contributions are summarized as follows:

Domain-Specific High-Level Synthesis for Image Processing IP cores

High-level and system-level synthesis have received a lot of attention over the past decades. However, only recent years have seen the development of tools that deliver a high enough quality of the generated hardware architectures in terms of throughput and efficient use of FPGA resources and allow non-experts a familiar approach to the many benefits of reconfigurable logic (refer to Chapter 3). Despite the fact that modern HLS tools, such as Vivado HLS [Xil15c] can produce hardware designs from almost any functional specification, the synthesized designs might not match the nature of the resources or interfaces on the FPGA and therefore only provide mediocre performance. A very high throughput can be achieved on the FPGA if the design uses pipelining in combination with data streaming instead of sequential memory accesses. Since such implementations must be explicitly specified, the HLS tools still require a lot of domain-specific knowledge to achieve high-performance implementations. In the context of *Domain-Specific High-Level Synthesis for Image Processing IP Cores*, this thesis makes the following contributions:

- Domain-specific support for HLS of a large set of image processing applications, consisting of point and local operators [Sch+14a*]. Here, a lightweight library of generic classes, methods, and functions is developed that solves the problem of specifying an appropriate hierarchical memory architecture for data-reuse in data streaming, as well as takes care of border treatment and causality in image processing with local operators. Furthermore, the library contains supporting elements for data streaming and therefore facilitates the assembly of complex image filter pipelines comprised of basic image processing operators.

- Domain-specific support for HLS of pyramidal algorithms is developed that is applicable for Multiresolution Analysis (MRA) applications from medical mage processing [Sch+14c*] as well as scientific computing [Sch+15b*].

- Finally, data structures and transformations to achieve a high degree of data-level parallelism that can be realized solely using HLS for FPGAs. In addition to providing support for well-known *loop-tiling* techniques, we propose the novel *loop coarsening* approach, which allows to process multiple pixels in parallel, whereby only the kernel operator is replicated within a single accelerator [Sch+15a*].

Using the library contributed to HLS in this thesis as a domain-specific high-level programming abstraction for medical image processing, it becomes feasible for non-expert designers to intuitively develop highly efficient dedicated FPGA accelerator IP cores using a familiar programming environment.

Interface Synthesis for FPGA-based SoC Integration of Image Processing IP Cores

As FPGAs are very versatile, not only the implementation of the functionality but the design and implementation of the communication architecture takes a lot of time. Using HLS, designers can develop highly efficient accelerator IP cores, however, the generated IP must still be integrated in an FPGA platform with given communication interfaces by hand to be of any actual use. A second desideratum is to minimize the interaction with FPGA development tools while implementing a design for evaluation and deployment. Also here, in order to enable testing of generated FPGA accelerators in a familiar software environment, as well as to allow a simplified deployment in SoC-like system architectures, including one or multiple processors together with a set of synthesized IP cores, this thesis makes important contributions:

- A highly flexible, all-hardware On-Chip Interconnect (OCI) structure as introduced in [Sch+13*] may serve for system integration of HLS generated accelerators with other components of an FPGA design. In this way, the design of complex filter pipelines, including multi-image [Sch+13*] and multiresolution algorithms [Sch+15b*] becomes feasible using solely HLS.

- Off-chip communication interfaces using industry-standard high-speed serial technology [SHT12*] to allow the integration of generated hardware accelerators in software-controlled environments [Sch+13*] and as part of FPGA-based SoC architectures.

- FPGA support designs to ease the integration of hardware accelerators into SoC-designs for board- and system-level evaluation. Hereby, hierarchical component implementation and final timing closure is automated to minimize interaction with FPGA development tools when the design with the integrated accelerator is implemented to obtain the configuration bitfile for the FPGA.

Overall, the above contributions provide new and important means to close the gap of modern HLS-based approaches to FPGA design in terms of EDA with a focus on case studies from the domain of medical image processing. Although we concentrate on medical image processing in this thesis, the contributions are also applicable to many other application domains.

1.4. Organization

We have divided this thesis into two parts, where the first part is concerned with domain-specific HLS for image processing IP cores and the second part is dedicated to interface synthesis for FPGA-based SoC integration of image processing IP cores. We begin the first part by providing an overview of the targeted application area of medical imaging in the next chapter, which will introduce the various challenges of the field and detail the domain-specific problems solved in this thesis. One fundamental aspect of our solution to close the design automation gap for FPGA accelerators even for non-expert developers is the automatic synthesis of hardware architectures to solve problems current HLS may not. Therefore, Chapter 3 will introduce the topic and provide an overview of the specific HLS environment we will concentrate on in this thesis. As HLS is still not an out-of-the-box solution, we discuss how specific problems from medical imaging can be solved efficiently using HLS in Chapter 4. The performance of the introduced hardware accelerators can be improved by exploiting data-level parallelism, such as loop-tiling and loop coarsening techniques, which are described in Chapter 5. The second part of this thesis is concerned with interface synthesis for FPGA-base SoC designs and distributed embedded systems. In Chapter 6 we introduce the concept of a highly flexible, all hardware OCI, that serves as a high-level abstraction layer for integrating a HLS design with the system architecture. Data communication is one of the most important facets of FPGA design and, moreover, one of the most challenging problems. To allow for efficient development of dedicated FPGA accelerators, we discuss appropriate means for board to host data communication in Chapter 7, which is essential to evaluate an accelerator design in a familiar software-controlled environment. Chapter 8 deals with board to board communication, which is vital also for the deployment of the developed FPGA accelerator in a distributed embedded systems architecture that might be composed out of multiple FPGAs or coupled to processors and memories, to give an example. Integration the HLS design with the proposed interconnect and peripherals is discussed in Chapter 9, where we introduce FPGA support designs that facilitate system integration, finally providing an unprecedented design automation path for rapid prototyping of FPGA accelerators.

Part I.

Domain-Specific High-Level Synthesis for Image Processing IP Cores

The first part of this work is concerned with domain-specific HLS for image processing IP cores. In the context of the *double roof* model, as shown in Figure 1.10, the work is situated at the ESL and contributes the necessary preparatory steps for those modules that must be synthesized in hardware for the FPGA target. For this, we first introduce the targeted application domain

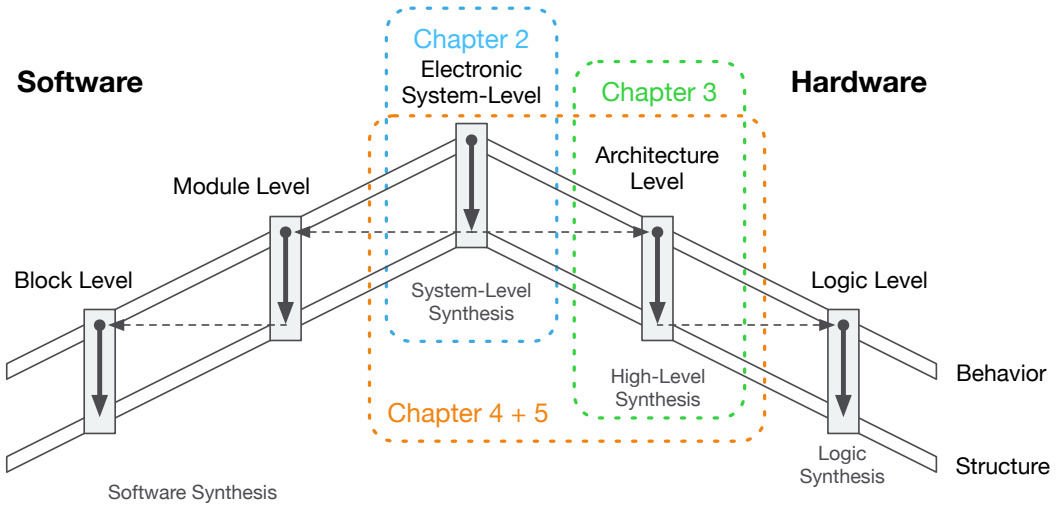


Figure 1.10.: Abstraction levels for embedded system design, adapted from Teich's *double roof* model [Tei97].

of medical imaging and explain the challenges solved in this thesis in Chapter 2. In Chapter 3, we introduce HLS and explain Vivado HLS as an example of a modern HLS tool in more detail. Chapter 4 details our approach for preparing applications from the medical image processing domain for HLS by introducing a lightweight library of generic classes, methods, and functions that allow the specification of a large set of image processing algorithms. In oder to exploit the opportunity for data-level parallelism on modern FPGAs using HLS, Chapter 5 shows how to apply well-known *loop tiling* techniques and proposes a novel approach called *loop coarsening*.

2
Medical Image Processing

The discovery of seminal physics phenomena, such as X-rays, radioactivity, and ultrasound have enabled the development of medical imaging instruments that have provided the most effective diagnostic tools in medicine up to date.

As with most technologies that prove to be magnificently useful, there are inherent up- and downsides to their use. For example, subjecting a patient to X-rays provides us with the possibility to gain insight to what is happening on the inside of the physiological system without having to open it up and the higher the dosage of the radiation, the more clear that insight will be. However, if an organism is exposed to too much radiation, the process may cause excessive cell damage. The risks may be mitigated by reducing the dosage of the radiation, which in turn dilates relative noise in the image. Hence, it is paramount to apply as little radiation as possible but still allow interpretation of the acquired information. In addition, medical images are commonly deteriorated by noise, stemming from various sources of interference. Moreover, other effects attempt to tamper with the measuring process in image and data acquisition systems, resulting in Gaussian noise and non-Gaussian errors. For the resulting images to be viable for visual and automated interpretation, medical imaging makes intensive use of *digital image processing*.

In the following, we provide a short introduction into the basics and fundamental applications as well as approaches to design. The chapter thereby serves as an introduction and motivation for the subsequent chapters on automated accelerator design for rapid prototyping of medical image processing applications.

2.1. Digital Image Processing

We denote an *image* as a two- or higher-dimensional array of numbers, which represent the real continuous intensity distribution of a spatial signal. To obtain a digital representation, the continuous spatial signal is sampled at regular intervals by some sensing modality and its intensity is quantized to a finite number of P elements. We can thus define a *digital image* of resolution $y \times x$ as a function $f : \{0, \dots, y-1\} \times \{0, \dots, x-1\} \to \mathbb{R}$. The value of $f(\mathbf{x})$, where $\mathbf{x}$ describes

any pair of spatial coordinates $(i, j), 0 \leq i \leq y-1$, and $0 \leq 1 \leq x-1$, represents the quantized intensity of the image at that point and is often referred to as a *picture element*, or short, as a *pixel*. Subjecting such images to mathematical operations in order to transform the image to a desired form is called *digital image processing*, if the computations are carried out by a computer. The results can hereby be, for example, an enhanced image or an interpretation of a scene and the contained objects. According to the purpose of the transformation, image processing can be distinguished into several categories.

Image Enhancement is aimed at improving the subjective quality or detectability of objects within an image.

Image Restoration focusses on the modeling of degradation processes and applies the inverse model to restore a degraded image.

Image Reconstruction describes the process of restructuring the available image data into a more useful form, for example, image super-resolution reconstructs a high resolution image from several low resolution images.

Image Analysis deals with the extraction of information from images.

Pattern Recognition is concerned with the identification of objects in images based on patterns.

Computer Vision describes a model-based approach to image processing, where models of the scene, as well as the imaging process are used to derive a higher-dimensional representation of a scene and the content of the images.

In the process from acquiring an image until to the visual inspection by a medical expert, medical imaging makes use of all of the above named categories.

2.2. Operations in Medical Image Processing

As the nature of deterioration in medical image processing can be manifold, a first step is to apply intensive digital signal processing during *image enhancement*, which aims at removing noise and making specific features more clearly noticeable. In a further step, *automated analysis* often applies segmentation to discriminate regions of interest from background noise. Essential diagnostic information can be acquired by applying quantification algorithms to segmented structures, which is a fundamental step for classification of tissue types. Furthermore, the registration of two or more data sources of the same part of the body is essential to applications where the correspondence between the data may deliver the desired information.

A sequence of such individual image processing steps to obtain a desired result is referred to as an *image processing algorithm*. Based on the computational complexity, Ratha and Jain categorize the operations in a three-level hierarchy [RJ99]. Tasks at the lowest level mostly involve pixel-based operations, such as filtering. Segmentation and grouping tasks are situated at the intermediate level and are characterized by more complex pixel operations. At the highest level are decision-oriented tasks, such as quantification and registration. In [DC98], Downton and Crookes have classified the level of data abstraction according to the hierarchy illustrated in Figure 2.1. Enhancement and preprocessing operations

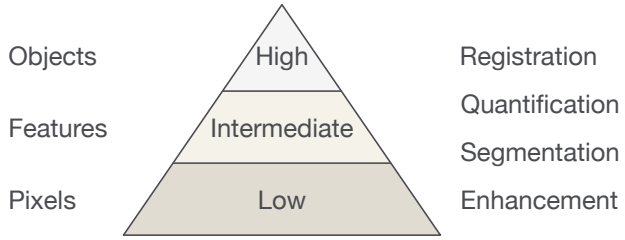


Figure 2.1.: Abstraction level pyramid.

at the lowest level are typically image-to-image transformations that operate on the pixel information of an image. Segmentation operates on intensity or texture variations and involves techniques such as thresholding, region growing, and deformable templates, but also techniques from pattern recognition. The transformation usually starts at the pixel level and yields features to describe objects or region information. Quantification at the border between the intermediate and the highest level makes use of the features detected by segmentation and involves these for the classification of objects. Registration at the highest level operates on the objects identified by quantification.

2.3. Medical Image Enhancement and Denoising Techniques

Enhancement and denoising methods for medical imaging are mostly mathematical methods to derive an image from an original source that exhibits certain details more explicitly than in the original and may thus aid during diagnostics and interpretation. Methodologies range from simple static calculations up to complex methods which are dynamically adapted to the image source or a specific feature to enhance. Inappropriate application of denoising techniques may make it harder to enhance certain structures of interest and, in turn, enhancement procedures might also amplify the noise present in the image.

2.3.1. Spatial Operators

Digital images, conforming to the definition presented in Section 2.1, can be processed through digital signal processing in the spatial domain by applying a spatial operator T to an image $f(\mathbf{x})$, also called the *input image*, in order to derive a *processed image* $g(\mathbf{x})$, often called the *output image*. We denote processing in the spatial domain by the expression $g(\mathbf{x}) = T[f(\mathbf{x})]$. In its simplest form, T is of size 1×1 and only operates on a single pixel. Note that the input image $f(\mathbf{x})$ and the processed image $g(\mathbf{x})$ need not necessarily be of the same image resolution. The operators of the spatial domain of image processing are often distinguished into *point*, *local*, and *global operator* classes, according to how much information of the input image is necessary to calculate a single pixel of the output image.

Point Operators

The class of point operators comprises image enhancement and manipulation techniques that only operate on actual pixel values without requiring information from the pixels in the local neighborhood. An example is given in Figure 2.2. A

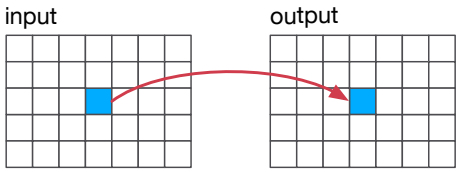


Figure 2.2.: Point operators only take single pixels into account for the calculations.

common operation for point operators is, for example, intensity scaling of an image, where the intensity of every pixel $f(\mathbf{x})$ is multiplied by a certain factor e. The basic operation is defined as

$$g(\mathbf{x}) = f(\mathbf{x}) \cdot e.$$

Another example is **RGB!** (**RGB!**) to grayscale conversion. Point operators are often a part of image processing algorithms, for example to square an intermediate result for further computations.

Local Operators

Local operators derive the output pixel by not only looking at the intensity of the pixel at the current position, but take a certain neighborhood surrounding the current image location into account, as illustrated in Figure 2.3. A prime

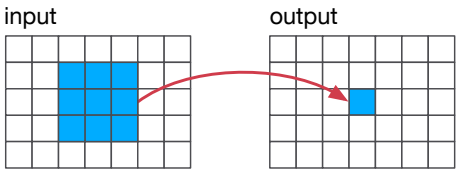


Figure 2.3.: Local operators take a local neighborhood of pixels into account for the calculations.

example is *convolution*, where the local operator is also often referred to as the *kernel*. We can consider a kernel $k(\mathbf{x}, \xi)$ of radius ξ as a two-dimensional array of coefficients, also called *mask*, and define convolution of an image with the kernel as

$$g(\mathbf{x}) = \sum_{\xi} f(\mathbf{x})k(\mathbf{x}, \xi). \tag{2.1}$$

Local operators are often used for noise suppression. For example, using *Gaussian convolution*, the smoothed image $g(\mathbf{x})$ can be obtained by convolving the image with the Gaussian kernel, defined as

$$k(\mathbf{x}, \xi) = \frac{1}{2\pi\sigma^2} e^{-\frac{1}{2}\left(\frac{||\xi - \mathbf{x}||}{\sigma}\right)^2},$$

which measures the *geometric distance* between the center pixel $\mathbf{x}$ and a pixel located at a nearby point ξ within a neighborhood centered around $\mathbf{x}$. For hardware implementation, filtering an image with the Gaussian kernel can be approximated efficiently using the *binomial* kernel as coefficient mask [AL96]. Edge preserving or enhancing filters are also a common application of local operators. The Sobel operator, for example, can be used to detect edges in images and guide noise reduction filtering. Another example is a simple isotropic derivative for edge detection in images, known as the *Laplacian operator*. The application of the kernel $l_{hv}(\mathbf{x}, 1)$ as a local operator to an image $f(\mathbf{x})$, defined by

$$\begin{aligned} g(i,j) &= f(i,j)l_{hv}((i,j),1) \\ &= [f(i-1,j) + f(i,j-1) + f(i,j+1) + f(i+1,j)] - 4f(i,j), \end{aligned}$$

can detect horizontal and vertical lines in images. To also detect diagonal edges, the kernel $l_d(\mathbf{x})$ can be applied, which is defined as

$$g(i,j) = f(i,j)l_d((i,j),1) = \sum_{\substack{m=-1 \\ m\neq 0}}^{1} \sum_{\substack{n=-1 \\ n\neq 0}}^{1} f(i+m, j+n) - 8f(i,j). \tag{2.2}$$

In combination with the Gaussian kernel, called Laplacian-of-a-Gaussian, the filter can be used for contrast enhancement in medical imaging [Ney93].

Global Operators

Global operators are distinctively different from the above described operators, as they often do not produce an image as output, but deliver information about the image, which can then be used for further processing. For this, global operators must take the entire image into consideration, as depicted in Figure 2.4. A

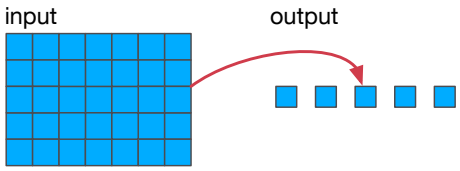


Figure 2.4.: Global operators take the complete image data into account for the calculations.

common operation for global operators are histogram computations. A simple histogram, for example, analyzes the input image and delivers a vector, containing the count of pixels in the image that represent a certain intensity value, or that fall into a certain range of intensity values. For a gray scale image with pixel intensities $f(\mathbf{x}) = i \in \{0, 1, \ldots, P-1\}$, the histogram $h(i)$ is defined as

$$h(i) = \sum_{\mathbf{x}} \delta(f(\mathbf{x}) - i),$$

where

$$\delta(w) = \begin{cases} 1 & \text{if } w = 0 \\ 0 & \text{else.} \end{cases}$$

Histograms have not only been used as global operators, but also in the context of local operators to enhance a local area in order to increase the visibility of subtle image features [Hum77]. Furthermore, histograms can assist in adaptive image enhancement [Piz+87].

2.3.2. Image Processing Algorithms

As the nature of deterioration in medical image processing can be manifold, it is most often the case that a single technique for denoising or enhancement may not be enough to provide a resulting image that is sufficient for interpretation. Image processing algorithms therefore apply several methods in sequence to derive sufficient results. Hereby, the term algorithm is used ambiguously, as the individual operations are also called algorithms. To clarify the terms, we use the differentiation between algorithms at the *operation level*, which describe the application of a single method, such as a local operator, and algorithms at the *application level*, which describe a sequence of operation level algorithms.

Application level algorithms can furthermore be distinguished into *single-* and *multi-image*, but also *multiresolution* algorithms.

Single-Image Algorithms

Algorithms that only require a single image are among the most common. A single input image $f(\mathbf{x})$ is transformed into the output image $g(\mathbf{x})$ by a sequence of image processing steps. One of the algorithms we will examine in this work is the Harris corner detector [HS88] as an example for single-image algorithms. At the operation level, the Harris corner requires the execution of several distinct steps, which are depicted in Figure 2.5. After building up the horizontal and vertical

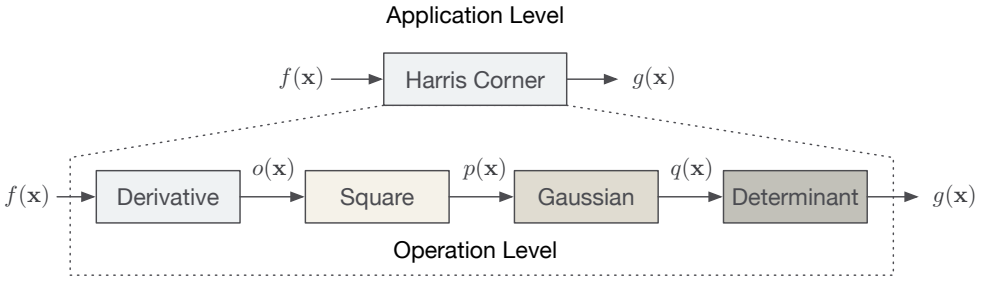


Figure 2.5.: The Harris corner detector at the application level comprises several operation level algorithms.

derivatives, the results are squared, multiplied, and processed by Gaussian kernels. The last step computes the determinant and trace, which is used to detect when a threshold is exceeded. The computation of the derivatives and the Gaussian filters are applied as local operators, whereas squaring, as well as determinant and trace computation are point operators. In medical image processing, the Harris corner detector is of interest for image registration, especially if there are rotational differences between the images [Gos05].

Multi-Image Algorithms

Multi-image algorithms, as shown in Figure 2.6, provide a further class for image enhancement in medical imaging. As a basic requirement, the images must be comparable, that is, they must possess the same dynamic range and preferably the same size. A widespread application is the suppression of temporal noise under the assumption of static scenes by *temporal averaging*. The basic principle is to average the intensity of the pixels of an image frame $f_0(\mathbf{x}$ with their predecessors in n previous images $f_1(\mathbf{x}), \ldots, f_{n-1}(\mathbf{x})$ to form the temporal average $t(\mathbf{x})$, as

$$t(\mathbf{x}) = \frac{\sum_{i=0}^{n-1} f_i(\mathbf{x})}{n}.$$

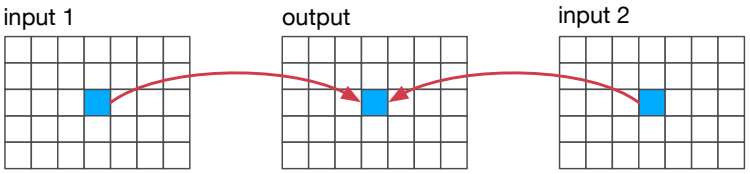


Figure 2.6.: Multi-image algorithms take multiple images into account for the calculations.

For dynamic scenes, however, the concept of bilateral temporal filtering can mitigate transient errors [Sch+13*]. The temporal average $t(\mathbf{x})$ is obtained by

$$t(\mathbf{x}) = \frac{\sum_{i=0}^{n-1} f_i(\mathbf{x}) c(f_0(\mathbf{x}), f_i(\mathbf{x})) s(f_0(\mathbf{x}), f_i(\mathbf{x}))}{c(f_0(\mathbf{x}), f_i(\mathbf{x})) s(f_0(\mathbf{x}), f_i(\mathbf{x}))}.$$

Here, $c(f_0(\mathbf{x}), f_i(\mathbf{x}))$ measures the temporal distance between the current frame $f_0(\mathbf{x})$ at time step t_0 and frame $f_i(\mathbf{x})$ at time step $t_0 - i, i = 1, \ldots, n-1$. The component $s(f_0(\mathbf{x}), f_i(\mathbf{x}))$ compares the photometric similarity between the intensity of the pixel $\mathbf{x}$ in frame $f_0(\mathbf{x})$ and that of pixel $\mathbf{x}$ in frame $f_i(\mathbf{x})$. Another application is to determine the direction of motion in an ordered sequence of images by means of the *Optical Flow* algorithm. In medical imaging, one of its uses is the *in vivo* assessment of heart performance involving tagged Magnetic Resonance (MR) images [PM92].

Multiresolution Algorithms

In Multiresolution Analysis (MRA), a certain data set is represented on different resolution levels, as shown in Figure 2.7. A well-known example is the Gaussian

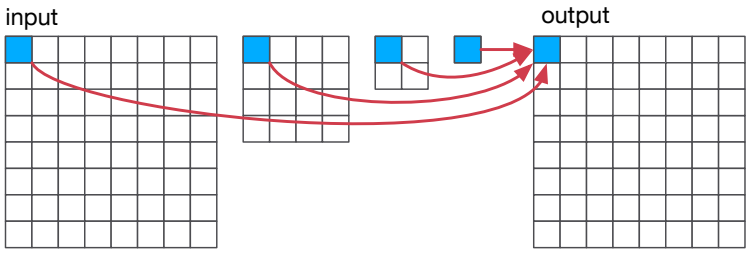


Figure 2.7.: Multiresolution image operations apply the operation to image versions at different resolutions.

pyramid, which is made up of low-pass filtered, downsampled images of the preceding stage of the pyramid, where the base stage g_0 is defined as the original image $g_0(\mathbf{x}) = f(\mathbf{x})$. Higher stages are defined by $g_s(\mathbf{x}) = \sum_\xi g_{s-1}(\mathbf{x}) w(\mathbf{x}, \xi)$,

where $w(\mathbf{x}, \xi)$ is a weighting function that is identical for all stages, termed the *generating kernel*. The weighting function closely resembles the Gaussian kernel, hence the name of the pyramid. The Laplacian pyramid uses Laplacian images, which can be obtained by up sampling the Gaussian image to its original dimension and subtracting it from the Gaussian image of the previous stage, $l_l = g'_l - g_{l-1}$. After processing the images at each stage, the output is reassembled by fusing together images of successive stages in a reconstruction step. For this, the smaller image is first increased in size to match the larger image then the two images are added together. Multiresolution processing is, for example, applied for adaptive contrast enhancement [Lai+93].

2.4. Accelerators in Medical Image Processing

The above introduced algorithms as well as many other applications in medical image enhancement and denoising may be programmed or specified as nested loop programs, which inhibit *embarrassingly parallel* operations and, given enough resources, all pixels could be processed concurrently. Such algorithms or related programs exhibit parallelization opportunities and are therefore often implemented by Graphics Processing Units (GPUs), Digital Signal Processors (DSPs), or Field Programmable Gate Arrays (FPGAs) [Keh10]. Performance, however, is not the only desired feature, as medical devices and medical image processing applications must be highly reliable and have deterministic latencies. Especially FPGAs can provide such characteristics and are moreover suitable for long-term application. They are therefore often chosen to accelerate medical imaging, for example, in systems from Intuitive Surgical[3] or Covidien[4].

2.5. Summary

In this chapter we have provided an introduction to the nature of basic medical image processing algorithms and their classification. We have explained fundamental operator types and how these are employed in several important image processing methods. Algorithms at the lower and intermediate abstraction levels, where computations can be carried out in a massively parallel manner were identified as the most attractive targets to speed up the execution using dedicated hardware accelerators.

[3] http://www.intuitivesurgical.com
[4] http://www.covidien.com

3

Automatic Hardware Generation through High-Level Synthesis

The process of designing hardware has always been cumbersome and required a specific technical skill set. In the beginning, designers had to develop systems composed of circuit-level components, such as diodes and resistors. The era of Small Scale Integration (SSI) saw designers coping with switching circuit components, which were Flipflops (FFs) and logic gates. Register Transfer Level (RTL) elements, represented by Arithmetic Logic Units (ALUs) and registers, were introduced with the move to Medium Scale Integration (MSI). The introduction of larger memories and processor primitives conducted the advent of Large Scale Integration (LSI). During this transition, it became more and more difficult to productively design hardware and the amount of logic gates that could be integrated into a single Integrated Circuit (IC) grew tremendously with the move to Very Large Scale Integration (VLSI). Despite of this development, rapid progress in computational technology was possible since the current generation of the technology could aid in the development of successor technologies. A valuable technique to cope with the ever increasing complexity of the current trend towards System-on-a-Chip (SoC) devices is Electronic System-Level (ESL) design, which is basically founded on the principles of Design Automation (DA) and the reuse of previously designed components [BMP07]. A practiced approach is to create often used modules once and then replicate and reuse these so called Intellectual Property (IP) cores. They can be classified into soft IP, where the functionality is described in a target-independent manner, firm IP, which describes the functionality in the form of a technology-dependent Hardware Description Language (HDL) on the Register Transfer (RT) level, or as hard IP, which provides a target-dependent netlist. IP cores on the RT level can only be firm or hard, since the implementation is almost always dependent on some specific features of the underlying technology. As efficient HDL descriptions vary in coding style for different target technologies, the desiderata for soft IP cores can only be achieved on the behavioral or functional level. Being able to synthesize hardware modules from a functional description in a High-Level Programming Language (HLL) does not only simplify hardware creation, but also

allows much more efficient design space exploration for embedded systems design. Although High-Level Synthesis (HLS) is an important topic for the development of both, Field Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs), in this chapter we will mostly focus on the approach for FPGAs and cover ASICs as targets only marginally.

3.1. Early efforts on the Path to High-Level Synthesis

The idea of shortening the cumbersome hardware design process is not a new one. In fact, the beginnings of HLS can be traced back until the early 1960s. However, to arrive at the level of DA that is possible today, innovations had to be made on all frontiers, ranging from appropriate input languages to algorithms for the various synthesis steps.

The earliest scheme for automatic synthesis was in the form of obtaining a description of a system on the RT level from Boolean equations by Press [Pre63] and Schorr [Sch64]. A noteworthy project of that era, where automatic hardware generation was mostly referred to as DA, is the Automatic Logic Design Generator (ALERT) [FY69]. It used the Iverson language as design input, which allowed the specification of a system in the form of individual microprograms, which would execute in concurrency with other microprograms. The system already addressed important aspects of automatic hardware generation, such as scheduling, called sequencing, and storage allocation, which was referred to as the automatic identification of FFs.
In a 1972 report of the current state of DA [Bre72], Breuer comes to the conclusion that automating the placement and interconnect process is feasible, however, synthesis and partitioning must still be made manually. Accordingly, the 1970's saw a burgeoning interest in research of the lower levels of the design hierarchy, such as automation of the layout process. Most of the developed approaches of that time were considered as research projects and did not receive much interest from the industry. Nonetheless, the decade saw great progress in the development of algorithms and techniques. Stemming from the works of Barbacci on Design Space Exploration (DSE) [BS73], Siewiorek proposed the EXPL system in [SB76], which was the first automated system to perform DSE using series-parallel tradeoffs. For EXPL, designs could be specified in the Instruction Set Processor Language (ISPL) from which it built a RTL structure, using a limited set of predesigned register-transfer modules to simplify the synthesis process. Another pioneering work of the late seventies, the Carnegie-Melon University DA system (CMU-DA) is considered as the major research effort. In CMU-DA, the design is specified at the behavioral level using ISPS, a more powerful

successor of ISPL, from where the output in either CMOS standard cell layout or Transistor-Transistor Level (TTL) could be obtained in a top-down approach. Moreover, the MIMOLA design system [Zim77] from the University of Kiel could aid in the design and user controlled optimization of complex digital processors and synthesis of microcode. The MIMOLA language could either be used as an RTL HDL, or for specification on the behavioral level, similar to todays popular HDLs, Very high speed Hardware Description Language (VHDL) and Verilog. In fact, the MIMOLA system was later acquired by Honeywell and was later used as a part in a VHDL design system [Kro98] [WC91a].
In the beginning of the 1980's, Darringer and Joyner [DJ80] report that despite the many attempts at automatic hardware generation, hardly any commercial success could be achieved. To their believes, this was mostly due to the fact that early systems, such as ALERT, required up to 160% more logic resources than handcrafted designs [FY70] and were therefore unattractive for industrial application. In conclusion, they present a system that produces an initial naive design from a functional specification, which is then refined by the designer through manual interaction with the tool. Their early prototype was able to achieve a high Quality of Results (QoR), delivering an implementation that was nearly equal to a manually designed system. Interestingly, the approach is already very similar to methodologies used today, where human interaction is still required for tasks, such as parallelization. A few years later, Shiva notes in [Shi83] that full automation of the synthesis process is not practicable and may also not be desirable. He comes to the conclusion, that the so far envisioned synthesis systems can be considered as a design aid for logic minimization, circuit analysis and simulation. Although other aspects were targeted for automation, such as placement routing and design rule checking, the synthesis systems of the 1980's have not been able to come closer to the creative aspect of the human design process, which is required to conquer the complexity of hardware design. Also in 1983, Gajski and Kuhn structure approaches into three different directions [GK83]. A first group believed in an all-manual process for digital logic design that should be approached from the bottom upwards. A second group backed the solution that human knowledge can be captured in design rules, which could then be used in expert systems for design automation. A third group was convinced that design knowledge is algorithmic and that synthesis from higher to lower abstraction levels can be achieved by translators.

Common to most approaches presented so far, was the decomposition of the design process into several steps, including *translation*, *module selection*, *data path scheduling and allocation*, as well as *synthesis* of the control path. Subsequent years saw the development of many tools following in the same footsteps, however, these were often only single research projects.

The earliest noteworthy system is HAL [PKG86]. It combined the translation approach with an expert system to achieve a structural representation of a

system in the form of functionally specified partition blocks. HAL first of all translates the functional description into a graph-based internal representation. Following are the allocation of possible resources and the partitioning of the resulted data path to control steps. After the allocation of registers, the data path is synthesized, where the allocated operators can be assigned to a library of standard cells or generated by a module builder. As a last step, HAL performs the synthesis of the controller and synthesizes the necessary interconnect. Based on an explicit timing constraint, HAL allows DSE to evaluate different options for design creation. For scheduling, referred to as partitioning, HAL uses a load balancing approach.

Another interesting approach is Carnegie-Melon's System's Architect Workbench (SAW) [Tho+88]. Starting from a behavioral description of a piece of hardware in ISPS, the behavioral synthesis system offers two approaches for synthesis. The first one is a style-specific synthesis program for microprocessors, called SUGAR, which includes a set of domain-specific design rules that allows expedited synthesis. The other approach is a general synthesis path, consisting of behavioral transformations, architectural partitioning (APARTY), control step scheduling (CSTEP), data-path synthesis (EMUCS), and a bus chooser (Busser).

Also worth mentioning is the ADAM synthesis system as introduced in [Jai+89]. ADAM builds on two parts to enable synthesis of an RTL system from a behavioral description and DSE using prediction tools, which guide the designer in DSE. The synthesis first of all requires a design style selection for either a pipelined design, or a non pipelined design. According to the desired design style, the prediction system generates individual lower-bound area-delay tradeoff curves, which are used by SLIMOS to select appropriate modules from a library. The system then chooses the MAHA tool for non-pipelined scheduling [PPM86], or Sehwa for performing pipelined scheduling. Both paths end up in the MABAL tool for allocation and binding. Over the course of the design, the prediction curves are updated with the area required for multiplexers and registers, the wiring, and the predicted area for the control logic. Finally, an overall tradeoff curve is derived to aid the designer in making design choices.

The Hercules/Hebe system [KD90a], is a part of the OLYMPUS system, developed at Stanford University, that researched high-level, logic, and physical synthesis. It is divided into behavioral and structural synthesis. Behavioral synthesis makes use of compiler transformations, enabled by a so called reference stack, whereas the structural synthesis emphasizes on iterative refinement for the generation of a data path and controller.

There were also noteworthy industry approaches, for example, Cathedral-II [De +86] from IMEC, which is specifically focused on synthesizing synchronous multi-processor chips for digital signal processing. As algorithms in Digital Signal Processor (DSP) often consist of a number sequential steps, the system

realizes each part as an optimized processor and interconnects these by specific busses. The Silage language, optimized for behavioral specification of DSP can be considered as a first work in the direction of HLS from a Domain-Specific Language (DSL). Silage uses signals as basic objects to describe an algorithm as a signal flow graph. Cathedral-II was specifically aimed at creating silicon implementations, for which it used a so-called meet-in-the-middle approach, that allowed a top-down implementation for an algorithm until the level of reusable silicon modules, which were created in a bottom-up fashion. An algorithm described on the behavioral level is simulated and optimized before it is compiled into a structural representation. Both steps are controlled by the designer and expert rules, specified in Prolog.

Another commercial tool aimed at synthesizing silicon implementations for DSP is the Bit-Serial Silicon Compiler (BSSC) [Yas+87] developed at General Electric. It used the Bit-Serial Language (BSL) for high-level specification of signal processing algorithms. Using behavioral simulation, the designer could verify and optimize the algorithm before processing it through the behavioral compiler, which performs the synthesis to a structural representation. The system also included a gate-level simulator to inspect the outcome of the synthesis. Furthermore, a layout system was used to perform placement and routing to yield a description of the finished design consisting of cells, pads, and routing.

IBM's Yorktown Silicon Compiler [WC91b] also proposed a completely top-down compilation approach. Starting from a program description in YYL (an extension of APL), the system carries out a sequence of synthesis steps, to first perform target independent, programmable logic optimization, before the target dependent synthesis adapted the structure to a specific technology. The input language could be executed as a normal APL program and could also include code for verification that would not be synthesized.

The time period also saw the development of many important algorithms used in HLS. The popular *list-scheduling algorithm*, for example, was introduced by Adam, et al., in [ACD74] and used in MAHA to solve non-pipelined scheduling with resource constraints [PPM86]. For pipelined scheduling in ADAM, Sehwa did not only generate implementations, but also allowed DSE of multiple solutions [PP88]. The HAL system proposed *force directed scheduling* for the optimization of resource constraints in the presence of performance constraints [PK89]. An approach to relative scheduling for supporting operations with unbounded delays in Hercules was detailed by Ku, et al., in [KD90b]. To aid synthesizing of data paths and their control in Finite State Machine with Data Path (FSMD) designs, *conflict-graph coloring techniques* for resource sharing were described [KP87] and Camposano presented the performance optimization of conditional branches [Com90].

In summary, these early projects helped to create an algorithmic basis for behavioral and logic synthesis and provided many innovative design techniques.

Since most tools were built on top of RTL synthesis, improvements in this area were able to spur success of behavioral synthesis on a higher abstraction level. En route to the new millennium, proprietary tools were developed from Philips [Lip+91], Siemens [Bie+93], IBM [Ber+95], and Motorola [Kuc+98]. Also, the vendors of Electronic Design Automation (EDA) environments introduced tools for HLS. Visual Architect from Cadence [Hem+94], Monet from Mentor Graphics [Ell99], and the Behavioral Compiler from Synopsys [Kna96] allowed the creation of RTL structures from behavioral descriptions. However, the used behavioral and untimed HDLs were not very well suited for hardware description and were thus not popular among designers. Eventually, the systems disappeared from the market, such as the Synopsys Behavioral Compiler in 2004.

3.2. Current Tools

Recent years have seen a new generation of HLS tools, which do not only allow to generate hardware architectures from hardware behavioral models, but perform synthesis starting from algorithms specified in HLLs. One of the reasons for this development is the ever growing popularity of reconfigurable logic, which aims at providing the performance and energy efficiency of integrated circuits at a flexibility that is very close to software. Despite being a very attractive hardware target for the acceleration of computations, the programming still often requires knowledge of hardware structures and the associated Computer Aided Design (CAD) tools for implementation. Being able to use a familiar language for algorithm specification makes developing for an FPGA platform as target more approachable to algorithm designers.

A very important aspect is a significantly easier development and accelerated verification of hardware/software systems. With mobile systems having to provide more and more computational power, a current trend in system design is to develop heterogeneous systems, comprising not only general purpose processors, but also highly energy-efficient hardware accelerators for computationally intensive tasks, such as video encoding or wireless communication. Being able to specify all parts of such a system, i.e., software and hardware, on a high abstraction level, preferably using the same programming language, greatly simplifies system development and allows verification at a very early stage of the design cycle.

There has been a lively discussion going on, about whether programming languages, such as C or C++ are suitable languages for the specification of hardware implementations [Edw06]. Often mentioned is the fact that these languages are intended to specify programs that are executed in a sequential manner on a microprocessor and lack proper mechanisms to describe the behavior of parallel hardware. Also, arbitrary precision arithmetic, one of the key advantages of

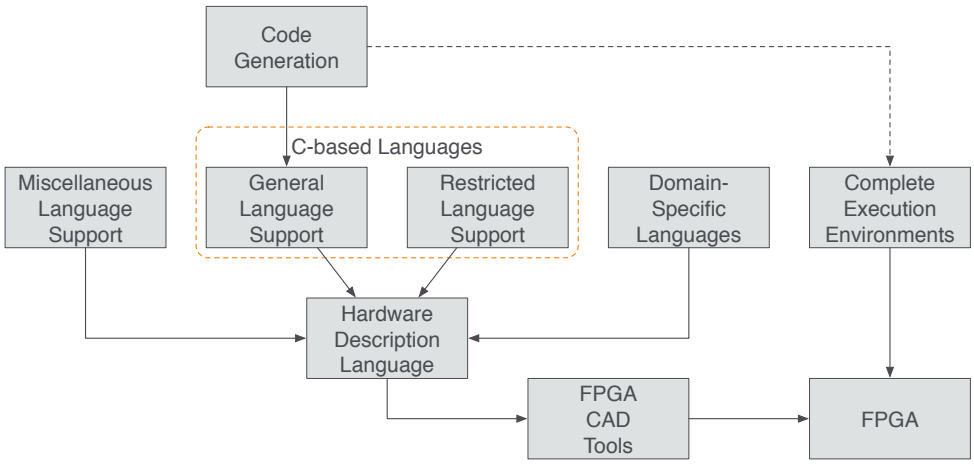


Figure 3.1.: Classification of HLS frameworks according to the support for C-based input languages.

hardware, as well as data streaming and pipelining are not supported by such languages. Moreover, these languages feature capabilities that are difficult to realize in hardware, such as dynamic memory allocation, recursion, and polymorphism.

To provide an overview of the many facets of available HLS frameworks, Figure 3.1 shows a classification based on language support and synthesis flow.

We can identify two basic concepts used by HLS tools that provide support for C-based languages to overcome these deficiencies. One concept is to provide *general language support* and require extensive user guidance for scheduling and synthesis. A second concept can be described as *restricted language support*, where the frameworks are often specific to a certain application domain, for example, image processing and attempt to perform several domain-specific optimizations automatically. Of course, there also exist frameworks that use an intermediate approach and could be classified into either the first or the second concept. Furthermore, there exist approaches that only use the syntax of C-based languages, provide domain-specific augmentations, or focus on other HLLs, such as Java or Python. We classify these approaches as *miscellaneous* language support. Recent years have also seen the advent of using DSLs as input to HLS to directly synthesize hardware architectures, but also as front ends to generate code for HLS frameworks. A completely new class of frameworks, mostly provided by Original Equipment Manufacturers (OEMs), are *execution environments*, which aim at fully abstracting from the FPGA design flow.

3.2.1. General Purpose Language Support

A very successful approach is (a) to make only a few restrictions to the C-based language or (b) provide extensions to support the modeling of hardware features. Tools following this approach can often synthesize almost any purpose into hardware. Since the compiler requires extensive user-controlled guidance, detailed knowledge of the underlying hardware target is yet essential to achieve a high quality of the synthesis results, such as high throughput, low resource requirements, or efficient use of the target platforms resources.

Catapult

Catapult is a commercial HLS product originally developed by Mentor Graphics and acquired by Calypto Design Systems in 2011 [Cal14]. Initially oriented towards ASICs, the product can meanwhile also target FPGAs. It supports C, C++, and SystemC as input language, from which it synthesizes individual functions into structural representations in VHDL and Verilog at the RT level, but also SystemC. Like many other C-based tools, the framework is very powerful, however, it requires deep understanding of the underlying hardware, since all transformations must be performed manually by the designer. Catapult can also generate simulation test benches from high-level models for the design to be verified by a cycle-accurate logic simulation tool, such as ModelSim [Men15].

C-to-Silicon

C-to-Silicon is a HLS tool from Cadence [Cad11] that can synthesize specifications given in C, C++, or SystemC into hardware descriptions in Verilog and supports the synthesis of different types of interfaces. A limitation exists for verification, where the design tool generates a non-standard SystemC wrapper, if C or C++ is used as input, which can only be simulated with the Cadence tool chain.

C to Hardware

C to Hardware is a commercial HLS tool from Altium and part of the company's Integrated Development Environment (IDE) [Alt13]. Input specified in C can either be synthesized as a set of connected components or in the context of Altium's Application-Specific Processor (ASP). As output, the tool generates IP cores in either Verilog or VHDL. Moreover, the tool can synthesize floating-point arithmetic, or optimize it to fixed point to improve area requirements.

Synphony C Compiler

Synphony C Compiler is commercial HLS system from Synopsys and the successor of the PICO tool from Synfora [Syn15]. The tool accepts input specifications in C/C++ and generates VHDL or Verilog code for FPGAs, as well as ASICs and is designed to integrate seamlessly with appropriate Synopsys tool chains. It supports the creation of test benches to simulate the compiled designs from high-level verification models and also supports a large number of common interfaces. To optimize sequential input programs, it features parallelization techniques, such as automatic single-to-multithreaded transformations.

Vivado HLS

Vivado HLS from Xilinx as part of Vivado Design Suite is the commercial successor of AutoPilot [Zha+08] and provides HLS from C, C++, and System C specifically for Xilinx FPGAs [Xil15c]. The tool is aimed at synthesizing individual functions into individual IP Cores in either VHDL, Verilog, or SystemC. Xilinx provides libraries for challenging problems and a partial port of the Open Source Computer Vision (OpenCV) [Pul+12] framework to aid designers. Moreover, the tool can make use of the extensive library of Xilinx IP cores, for example to support floating-point arithmetic. Specific focus has been laid on interface synthesis, which in addition to general purpose communication also supports ARM's AXI4 standard to integrate generated hardware modules with the FPGA fabric. Vivado HLS is used for HLS in this work and is explained in more detail below.

LegUp

LegUp is a research compiler from the University of Toronto, specifically targeted at Altera FPGAs [Can+11]. From an input specification in C, the compiler can generate Verilog for selected functions that are either synthesized as part of a hardware/software co-design to be controlled by a softcore processor, or it can generate standalone a hardware implementation. Moreover, it supports floating-point arithmetic and can automatically generate test benches.

3.2.2. Restricted Language Support

An alternative the above presented concepts is to restrict language support to a subset, but use a highly intelligent compiler that can perform many optimizations needed for a certain implementation style without requiring extensive user interaction.

Cynthesizer

Cynthesizer, originally from Forte Design Systems and recently acquired by Cadence, is a HLS compiler that enables design specification in SystemC and generates Verilog RTL [For15]. It offers verification and co-simulation, as well as formal equivalence checking, power analysis and a number of optimizations.

SPARK

SPARK is a modular and extendible academic HLS framework, currently hosted at the University of California in San Diego [Gup+03]. From an input program in ANSI-C, the system may generate VHDL IP cores at the RT level. It is particularly targeted at multimedia and image processing applications alongside control-intensive microprocessor functional blocks. The input code is partitioned into basic blocks before being translated into the Directed Acyclic Graph (DAG)-based internal representation. SPARK can carry out a number of compiler optimizations before scheduling to synthesize a structural representation.

GAUT

GAUT is an academic HLS tool for HLL input [Cou+08]. It was designed specifically for DSP applications, for which it can generate a pipelined processing unit, as well as memory and communication units. In addition, GAUT features automatic test bench generation and support for floating-point arithmetic.

Trident

Trident is a research-based HLS system aimed at synthesizing applications from scientific computing operating on floating-point data [TGP07]. As the compiler uses LLVM, it has access to all the high-level transformations and code optimizations available in the framework.

HercuLeS

HercuLeS is a commercial compiler from Ajax Compilers, that offers C to VHDL compilation of IP cores [KM12]. The generated structure follows the FSMD approach and is generated from a novel internal representation, called N-address code (NAC) that can be handled through the GNU Compiler Collection (gcc). For optimization, it provides high-level code transformations through gcc and additional hardware optimizations.

ROCCC

, the Riverside Optimizing Compiler for Configurable Circuits [Vil+10], takes a subset of C to produce hardware accelerators in VHDL without the addition of explicit, implementation controlling constructs. The original 1.0 version of the compiler was built on top of SUIF, however, since this format is no longer supported, the 2.0 version was rebuilt on top of LLVM. Like many other HLS systems, ROCCC is not intended to create hardware for complete systems, but to accelerate critical application parts, such as loop nests, in the form of IP cores. Its design goal is to maximize throughput while minimizing memory accesses and hardware resource requirements. It uses a top-down design methodology but also allows to create and import individual modules, which are stored in a database and can be called as C-functions, for example, it is possible to include hardware IP for floating-point arithmetic. Specific to ROCCC is the automatic minimization of memory accesses using the so-called smart buffer. To facilitate streaming pipelines, ROCCC allows the creation of FIFO-based channels between hardware systems in addition to explicit memory access.

3.2.3. Miscellaneous Language Support

Of course there are also several approaches that do not specifically follow one or the other concept but are situated somewhere in the middle.

Cyber Workbench

Cyber Workbench (CWB)is a set of synthesis and verification tools for system-level designers of ASICs and FPGAs, developed by NEC [WO00]. Designs can be specified in HDLs, SystemC and a superset of ANSI-C, which is extended by constructs to express hardware knowledge, called Behavioral Description Language (BDL), for example, variable bit-depths, concurrency enablers, parallelism and memory descriptions. CWB generates IP cores in either VHDL or Verilog, supports fixed scheduling, and automatically achieves resource sharing and pipelining. Emphasis is put on the design and co-verification of complete hardware/software systems, such as Multi-Processor System-on-a-Chip (MPSoC) designs, which can be verified as C programs and by cycle-accurate simulation.

Bluespec

Bluespec is a HLS tool from Bluespec, Inc.that uses Bluespec SystemVerilog as design language [AN08] and also offers emulation tools and IP cores. Modules are implemented as a set of rules, called Guarded Atomic Actions, that express behavior in the form concurrently cooperating Finite State Machines (FSMs). Bluespec is mostly targeted at designers with prior hardware knowledge, as

the verification of generated implementations is only possible in HDL using appropriate test benches.

Bambu

Bambu is an academic HLS tool from Politecnico in Milano in the context of the Panda framework [Cas+13]. It uses gcc for processing input programs in C to compile hardware architectures for FPGAs and ASICs.

3.2.4. Domain-Specific Languages for HDL Generation

Whereas language restriction often leads to limited applicability of the HLS tools, the converse approach of unlimited language support still requires in-depth knowledge of the underlying hardware to achieve a high quality of results. A very interesting concept are domain-specific languages, which focus on a specific target domain and may produce highly optimized hardware implementations in return.

SPIRAL

SPIRAL is a framework for the automatic code generation, optimization and platform adaptation of DSP algorithms for hardware and software targets. The research for this project is conducted at CMU and involves a large, interdisciplinary team of researchers. Although the complete framework is most thoroughly covered in a journal article from 2005 [Püs+05]. The domain-specific framework consists of a code creation and evaluation path, as well as an expert system that can learn from previous results to find an optimal solution. The system uses a custom DSL, called SPL, to input DSP transforms and express rules for the autonomous optimizations. The code generation path is subdivided into algorithm, implementation, and evaluation levels. Algorithms to be synthesized in SPIRAL are given as a set of rules, for which the algorithm level generates a formula and optimizes it using a database of breakdown and manipulation rules. Specified as a formula in SPL, the algorithm serves as input to the implementation level, also called the SPL compiler, which is divided into an implementation and code optimization block. The compiler translates SPL into Fortran or C and applies several implementation options, such as loop unrolling. The code optimization block performs various high-level transformations and architectural optimizations. The generated implementation is compiled and evaluated at the evaluation level, which passes performance results to the expert system, referred to as search and learning. The expert system can also control the algorithm and implementation levels to find optimal solutions autonomously, using techniques, such as dynamic programming and evolutionary search, but also techniques

from Artificial Intelligence (AI) for learning. Recent years have seen research on energy efficiency [THM06], vectorization [FVP06], retargeting a part of the framework to support automatic construction code generators using Scala and Lightweight Modular Staging (LMS) [Ofe+13], as well as the compilation of linear algebra computations [KSP15].

HDL Coder

HDL Coder is an extension to the Matlab/Simulink environment and generates synthesizable VHDL or Verilog code from Matlab functions, as well as Simulink and Stateflow models. The tool provides HDL coding with focus on the use of predesigned library components for commonly used functions in signal processing. It automatically converts floating-point arithmetic into fixed point and supports user-controlled optimization for the hardware implementation, such as loop and memory optimizations. Through integration with FPGA IDEs from Xilinx and Altera, HDL coder offers advanced verification features that allow the back annotation of the algorithm model with estimated hardware execution times, as well as support for cycle-accurate HDL simulation and hardware-based co-simulation.

PARO

PARO is a high-level synthesis framework that is capable of automatically generating highly parallel hardware accelerators for a broad variety of multi-dimensional dataflow dominant applications [Han+08]. It is based on polyhedral loop specifications and can therefore perform a wide range of polyhedral optimizations to expose an algorithm's parallelism and restructure it for implementation on or be synthesized as a massively parallel hardware target. The input language for PARO, called PAULA [Han09], is a domain-specific functional language that allows the description of an algorithm in a very close resemblance of the mathematical specification (refer to Chapter 2 for image filter applications). The internal representation is the reduced dependence graph (RDG), which separates the data flow from the static control flow. To restructure an algorithm for implementation, PARO applies a number of high-level polyhedral transformations on the internal representation, before scheduling the execution of loop iterations at specific control steps on a fully parallel multi-datapath architecture. Instead of only focusing on a specific target architecture, PARO can produce hardware descriptions for FPGAs and ASICs, as well as, produce programs for Coarse-Grained Reconfigurable Arrays (CGRAs), such as Tightly-Coupled Processor Arrays (TCPAs) [BHT14].

Darkroom

Darkroom was recently introduced as a language and compiler for image processing. The framework automatically generates fully pipelined, buffered hardware implementations for image processing kernels for FPGAs and ASICs, as well as for Central Processing Units (CPUs). For implementation, the input is first transformed into a DAG onto which the compiler performs a number of high-level transformations. It also generates an ILP to determine optimal buffering and causality preserving index shifts, from which the pipeline is generated. The optimized pipeline is then used to produce code for the individual target platforms.

LALP

LALP is a domain-specific language and compilation environment for aggressive loop pipelining (ALP) [Men+09]. It uses a C-like syntax for the DSL and is targeted at accelerating loop programs. Instead of using an FSM for controlling the loop within the iteration space, LALP uses counters and executes conditional loop statements based on counter values. The framework provides the synthesized hardware description in the form of VHDL code at the RT level.

Optimus

Optimus is an optimizing compiler for synthesis of streaming applications on FPGAs [Hor+08]. Algorithms specified in the Synchronous Data Flow (SDF)-based streaming language StreamIt [TKA02], consists of processing modules interconnected by streams. Hardware creation is split into synthesis of the stream graph and the synthesis of the processing modules. Modules are partitioned into basic blocks which are then replaced with specific templates. Optimus can perform several optimizations on the streaming infrastructure as well as used different scheduling techniques to obtain rate-matched or ASAP-like execution schedules.

3.2.5. Code Generation

The concept of domain-specific languages is also used to overcome insufficiencies of HLS frameworks by generating highly optimized intermediate code that already contains many optimizations and synthesis directives which would otherwise have to be included by hand and would make the development process more challenging. Other concepts allow parallel computing languages for algorithm specification and generate optimized C-code for HLS using one of the established tools.

HIPAcc

HIPAcc is a publicly available domain-specific framework and language for automatic code generation for multiple targets for the class of two dimensional image processing algorithms [Mem+15a]. Starting from a DSL embedded in C++, image filter descriptions can be synthesized by translation into multiple target languages such as Compute Unified Device Architecture (CUDA), Open Computing Language (OpenCL), or Renderscript as used in Android [Mem+14]. By building on a library of highly optimized code templates for image processing for C-based HLS (refer to Chapters 4, 5, and [Sch+14a*]), the framework may also generate code specific to the HLS framework Vivado HLS from Xilinx [Rei+14*].

Several other approaches employ source-to-source compilation to generate code for an established HLS framework from parallel programming models. In [CBA13], the authors describe an approach of how to compile programs using OpenMP and Pthreads to C code that can serve as input to LegUp. A very interesting approach that uses Scala for code generation is described by George, et al., in [Geo+13]. The authors use LMS to perform optimizations on internal representations on different abstraction levels in order to avoid interference. The described approach may generate C code for synthesis in LegUp.

3.2.6. Execution Environments

Many of the above presented approaches are only aimed at generating hardware descriptions for individual functions, but not to synthesize complete systems. Integrating any such generated IP core would still require a lot of knowledge of hardware design. A fairly recent development, mostly targeting FPGA platforms are execution environments, that aim at complete abstraction from the underlying hardware and the involved processes.

Altera SDK for OpenCL

Altera SDK for OpenCL is a commercial framework for HLS from Altera that allows to synthesize kernels described in OpenCL, a programming model based on ANSI C with extensions to extract parallelism, to FPGA accelerators. Instead of producing IP cores that could be integrated with custom designs, the tool is focused on providing an automated execution environment for FPGA acceleration to software developers without prior knowledge of FPGA hardware technology. The Altera solution is dependent on the availability of a host processor for execution of the kernels and specific to the company's hardware.

SDx Environments

SDx Environments are commercial products from Xilinx to enable the use of FPGA acceleration to software engineers without prior technology-specific knowledge. The environment consists of the SDNet tool, which focuses on hardware accelerated management of communication networks, and SDAccel, a tool for automatic HLS and execution of compute kernels on FPGA. Similar to the approach taken by Altera, SDAccel can handle OpenCL as input language and is aimed at complete abstraction from the underlying hardware.

Although the execution environments solve the problem of how to avoid FPGA CAD, the fact that they only produce complete system designs and rely on the availability of tool-specific embedded CPUs makes the approaches infeasible for general rapid prototyping, as proposed in this work.

3.3. Vivado HLS as an Example for C-based High-Level Synthesis

3.3.1. High-Level Synthesis

Over the past decades, the FPGA community has seen the move to ever higher abstraction levels. Each new abstraction layer aims at hiding design complexity, which offers increased productivity at the cost of less visibility of the challenges associated with the lower abstraction level. The current move from behavioral to functional design specification brings the benefit of accelerating the overall design cycle including expedited design simulation, automatic correct-by-construction RTL creation and easy design space exploration. C-based design entry is a very popular mechanism to create functional specifications and many engineers are familiar with either ANSI-C, C++, or SystemC. Most of the language constructs are supported, except for recursion and those involving dynamic allocation of memory, which is known to be challenging on hardware.

Synthesis of a functional specification in C to an RTL description for hardware implementation involves not only the synthesis of the specified algorithm but also the synthesis of an interface to the hardware module. The combination of both can lead to a plethora of different designs. The individual steps to be taken during high-level synthesis can be categorized into control and data path *extraction*, *allocation* of suitable resources, *binding* of the elements of the control and data path to the resources, and *scheduling*, as described in Section 1.1.6. Over the course of these steps, design *constraints* and *optimizations* can severely influence the available *design space* and thus the achieved design solution.

Control and Datapath Extraction

The first step in HLS is to extract control information from the functional specification by expecting the control flow of the program, i. e., conditionals and loops. Vivado HLS achieves this by extracting a control flow graph, which is then implemented as a finite state machine in hardware. For the initial extraction, it is sufficient to assume single cycle instructions. In further steps, the control logic will most likely have to be adapted according to allocated resources and interfaces. Data path extraction can be achieved rather easily by unrolling loops in the design specification and evaluating conditionals.

Scheduling and Binding

Scheduling and binding are the two most essential steps during HLS. After instructions have been bound to resources for execution, a schedule can be determined to plan when which instruction is executed. The order might also be reversed by first determining a schedule for execution, for example, by specifying very strict timing constraints, and performing allocation and binding according to the schedule. For this, Vivado HLS always requires the user to specify a basic set of constraints, consisting of the targeted hardware platform and clock period. It is up to the user to constrain the design further using so-called *directives*. Amongst others, directives can be used to specify a certain interface, how loops are to be handled, and which resources are to be allocated for certain instructions.

Optimization and Performance Metrics

Vivado HLS can perform several optimizations to produce high quality RTL to satisfy as well performance as area goals. It uses two metrics to describe design performance, namely the *latency* and the *initiation interval*. In general, latency specifies how many clock cycles it takes to complete all of the computations of a module. If the algorithm contains loops, the *loop latency* specifies the latency of a single iteration of the loop and the *Initiation Interval* (II) determines after how many cycles a new iteration can be started. If the iterations of the loop are executed in a sequential manner, meaning that an iteration has to finish all computations before the next iteration can start, the Initiation Interval (II) is equal to the loop latency. In this case, the total latency, or as named in Vivado, the loop interval, lat_{total} can be computed by multiplying the iteration latency lat_{it} by the number n of iterations to be performed, also known as the *trip count*. An additional latency must be added for entering and exiting the loop, as well as performing tasks outside of the loop, which we account for as lat_{extra}. In other words,

$$lat_{total} = (n \cdot lat_{it}) + lat_{extra}.$$

For large values of n, the extra latency may become negligible. However, loops in algorithms can often be parallelized in order to decrease the overall latency. In this case the II might become significantly smaller than the iteration latency which may decrease the overall latency significantly,

$$lat_{total} = (n \cdot II) + lat_{iter} + lat_{extra}.$$

The II can thus also be seen as a measure of throughput of the generated RTL design. The antagonist of parallelization and throughput optimization are area constraints. To be able to execute in parallel, operators must either be replicated or at least pipelined. Both increase the hardware resource requirements of the generated RTL. A brute force optimization strategy is to unroll all loops of the algorithm and provide dedicated operators and data storage for each iteration. In consequence, this will also demand a high amount of resources. Fortunately, HLS enables strategies to speed up the execution of an algorithm while keeping the resource demands at a minimum. In addition to demanding a certain clock frequency for the design, users can specify latencies across functions, loops, and regions. Moreover the amount of resources can be constrained and data dependencies can be relaxed, for example, by permitting read before write.

3.3.2. The Vivado HLS Design Flow

Figure 3.2 shows an overview of the design flow in Vivado HLS. Design entry can be made using either C, C++, or SystemC. Additionally, it is possible and recommended to specify a functional, self-checking test bench for the design. Inside Vivado HLS, the design can be verified against the test bench before the

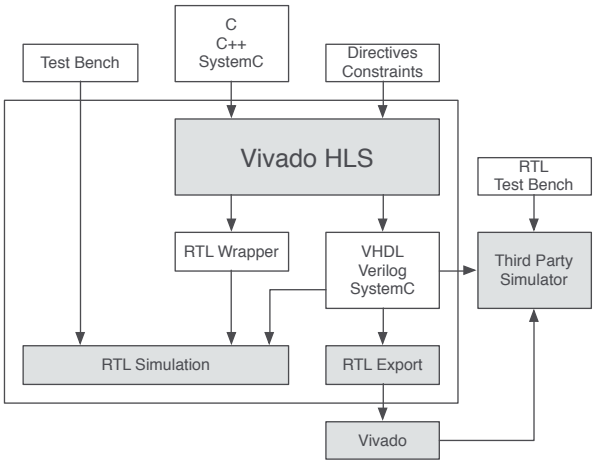


Figure 3.2.: Design flow in Vivado HLS.

actual synthesis. This is very useful, since Vivado contains several libraries to support the specification of an algorithm for hardware synthesis, which can be tested inside of HLS. Moreover, the functional test bench is used to craft the RTL simulation for the synthesized design. RTL sources are available for use outside of Vivado HLS after the synthesis, although it is also possible and often useful to use the RTL export function of the tools, which can make the synthesis results available in the Vivado IP core library, as well as generate an IP core for System Generator or Xilinx EDK. The contained RTL simulation can be used for design validation, however, not to find where errors might stem from. A third party external RTL simulator can be run on the generated output of RTL synthesis in conjunction with a custom RTL test bench. Moreover, there exists an almost one-click simulation setup. Exporting bundles the RTL source files together with a script to import these into a Vivado synthesis project. This script can be edited to also include a custom RTL test bench and setup an external simulator. In Vivado it is then only a single click to start the simulation.

3.3.3. Unsupported C Language Constructs

Vivado HLS is able to synthesize a large subset of the C modeling standard. However, there are several language constructs, that are not supported for high-level synthesis. One requirement is that the C function must contain the entire functionality and must not rely on any calls to system functions or the underlying operating system. Moreover, C constructs must use a fixed, bounded size and the implementation must be unambiguous. Therefore, Vivado HLS cannot synthesize any system calls, such as printf(), time(), sleep(), etc. Some of these functions are supported to simulate the design with the functional test bench, however, they are ignored for synthesis. As memory cannot be allocated dynamically during runtime, all memory requirements must either be outside of the scope of the function or their fixed bounds must be known at compile time. In general, pointer casting is only supported for native C types. Furthermore, the C++ STL is not supported.

3.3.4. Arbitrary Precision Data Types

C-based native data types are only on 8-bit boundaries. RTL on the other side can support arbitrary lengths. HLS includes a mechanism to support arbitrary precision data types. This is crucial for area optimization, as it is suboptimal to be forced to use a 32-bit multiplier in a design, where a 17-bit operator would be sufficient. HLS provides such data types for C and C++, as well as it supports the data types of SystemC. If arbitrary precision data types are to be used in C functions outside of Vivado, the Vivado compiler `apcc` must be used to compile the source code. Although standard compilers, such as gcc can

compile the sources, the produced executable will not be able to handle the compilation directives. These restrictions do not apply to C++, as the custom data types are implemented as a class, which can be compiled correctly by standard compilers. For C++ and SystemC, Vivado supports fixed-point data types and provides an appropriate library. Arbitrary fixed-point precision can be defined by specifying the overall word-length, the amount of bits for the integer part, as well as instructions to control rounding behavior. In addition, HLS supports the synthesis of single and double precision floating-point data types.

3.3.5. Interfaces

Interfaces for RTL designs synthesized by Vivado HLS are generated from the arguments of the target function. The `interface` directive can be added to enforce a certain interface protocol, however, supported protocols greatly depend on the type of the argument. Vivado differentiates between C function interfaces and block interfaces. C function interfaces define how data is passed to the synthesized function, whereas block level interfaces apply an interface to the generated IP core, which controls, for example, when to start execution or when the block is ready to accept new data. Function interfaces are applied to individual function arguments, block-level interfaces are applied to the function itself. A more concise coverage of interface synthesis in Vivado HLS is provided in Section 6.1.

3.3.6. Optimization and Synthesis Guidance

Vivado HLS provides the user with several options to optimize the design in terms of resource usage and performance. Optimizations can be categorized into function optimization, meaning to optimize the execution of functions in relation to each other and optimization of the elements of a function, such as loop constructs and storage elements, for example, arrays. Optimizations are specified as *directives* and can be entered either as *pragmas* directly in the source code or by using a directives script file. Both options have their reasoning, as pragmas might be used for directives which always apply, for example, loop unrolling, and directives which might change, such as pipelining with a certain initiation interval, can be easily grouped together in the script file.

Function Optimization

There are several possibilities to guide the synthesis of subfunctions within a function. An optimization also known from traditional C and C++ programming is function *inlining*, which copies the function code to the location where the function is called. The concept is ideally suited for small functions and for

those, which are shared amongst several other functions. In terms of Vivado HLS, using the `inline` directive on a function puts it within the context of its encapsulating function, thus, enabling optimization during synthesis. Another useful directive is function `instantiation`, which can be used on functions with a complex control set to split these up into smaller simple functions and instantiate these directly where they apply. Moreover, function synthesis can be guided by constraining the minimum and maximum number of cycles for the function latency, using the `latency` directive, as well as by choosing a specific interface through the `interface` directive.

Array Optimizations

As memory dimensions must be fixed, arrays are the preferred structure for data storage and may thus have great influence on area and performance. As a consequence, there are several optimizations available that can be applied to arrays. As arrays are typically implemented using a Dual-Ported Random Access Memorys (DPRAMs), most directives are concerned about how the C structure is implemented in hardware. The `resource` directive makes it possible to select a specific DPRAM implementation in RTL, such as, for example, one using asynchronous read but synchronous write, or a traditional true dual-ported implementation. Furthermore, small arrays may rather be implemented using registers or a larger multi-dimensional array might be split up and implemented using several DPRAMs. For these cases, the `partition` directive can be applied to control the implementation. Also, the contents of an array can be reorganized using the `reshape` directive. This is useful if, for example, multiple consecutive values should be read from an array. The array can be *reshaped* and the contents can be read in parallel using a larger word-width. Finally to use hardware resources more efficiently, multiple small arrays can be mapped together into a single DPRAM component using the `map` directive.

Loop and Function Optimizations for Parallelization

A crucial part of the optimization of functions is concerned about loops and sequential function calls to exploit *temporal parallelism* by overlapping their execution, also known as *pipelining*. Pipelining a loop nest on a specific level requires that all lower-lever loops are completely unrolled. Unrolling a loop can be instructed by the `unroll` directive and aids in exploiting *instruction-level*, or *logical parallelism*, by generating dedicated hardware for each operation in the loop, which results in the highest hardware requirements, but also the highest performance. Completely unrolling is advisable only for the innermost loops of a loop nest, such as a convolution kernel. Moreover, unrolling requires the loop bounds to be known at compile time. Higher-level loops can be pipelined by the

`pipeline` directive. In contrast to unrolling, the operators will be shared across the iterations of the loop, resulting in substantial speedup compared to sequential execution at a very reasonable increase in hardware cost. To ideally prepare loops for optimization, loops should be specified *perfectly* instead of *imperfectly*. Consider the following imperfectly specified loop construct to implement a shift register.

```
int A[7]; // this is the shift register
int new_data; // the image data
// implement the shifting as a loop
for(int i = 0; i < 6; i++){
  A[i] = A[i+1];
}
A[6] = new_data;
```

To *perfectly* describe the loop, it should be rewritten as follows.

```
for(i = 0; i < 7; i++){
  if(i == 6)
    A[i] = new_data;
  else
    A[i] = A[i+1];
}
```

Perfectly specified loops may be optimized better than imperfectly specified loops, especially when Vivado HLS analyzes loop-carried dependencies. However, it may also occur that Vivado HLS determines dependencies within a loop (*intra*) and between loops (*inter*) too conservatively and prevents the level of optimization that would actually be possible. For these occurrences, Vivado HLS provides the `dependency` directive to actively set dependencies to `false` if they incorrectly prevent loop optimization. Moreover, if loop bounds are dependent on a function argument, Vivado HLS is currently unable to determine the trip count, that is, the number of iterations of the loop. In such cases, HLS can be informed about the maximum number of iterations using the `tripcount` directive to specify a lower and an upper bound on a loop. In addition, separate sequential loops can be merged to reduce overall latency and improve operator sharing and nested loops can be collapsed into a single loop using the `flatten` directive.

If a top-level function contains separate sequential loops or sequentially called subfunctions, such as for the description of an algorithm on the application level as a sequence of operation level steps, the loops and calls can be pipelined and paralellized by using the `dataflow` directive. Dataflow creates a pipelined architecture, thereby synthesizing each called function and executed loop into an individual module. It also optimizes the communication between the functions and can improve the design throughput, but it leaves the specification and generation of storage elements between the pipelined modules to the designer.

3.4. Conclusion

The tremendous progress in design automation saw the development of many great tools for HLS which deliver many benefits in comparison to hand coding hardware at the RT level. In a coarse comparison, we can distinguish the proposed approaches and available frameworks into two groups. One group severely limits the scope of the application, but is therefore also capable of providing a design flow that is accessible to developers lacking detailed hardware knowledge. The other group tries to provide a versatile design solution that can handle HLS for many different application areas, but achieving a high quality of the synthesized results requires proficiency in hardware design. The most autonomous approaches to enable software-like programming of reconfigurable hardware are presented by execution environments, however, these are currently limited to specific hardware and the generated implementation cannot be used out of the scope of the environment.

Despite the potential, automatic hardware generation that can be operated by non-experts is still not completely solved. Although recent developments have produced many great approaches, they are often only focused on a specific application domain or require expert knowledge to produce acceptable results. Furthermore, the fundamental problem of providing connectivity to a generated accelerator has been targeted only recently and is far from providing a general solution.

4

Domain-Specific High-Level Synthesis for Image Processing on FPGAs

In the previous chapter, we have outlined the tremendous progress achieved in design automation starting from C-based language specifications with a focus on High-Level Synthesis (HLS) and Field Programmable Gate Array (FPGA) targets over the past decades. Meanwhile, frameworks starting from imperative languages only put few constraints on the code to be synthesizable. As we exemplify in the beginning of this chapter, although code intended for software execution can be synthesized, it will only produce mediocre results in terms of throughput and resource requirements on an FPGA. Significantly improved performance and resource utilization can only be achieved by adopting a coding style that exploits the target architecture artifacts. As an important observation and motivation of the following central results of this thesis, it is shown that in order to generate highly efficient accelerators for image processing applications on FPGA targets, such a coding style requires to (a) define an appropriate memory architecture to exploit data streaming in the hardware implementation instead of sequential global memory accesses, (b) handle causality problems that arise when data streaming is combined with local operator processing, and (c) support border treatment. Yet to abstract from these low-level details and let the designer focus on the actual algorithm design, a solution based on a lightweight C++-embedded library of code templates for C-based HLS of image processing accelerators is proposed that includes

- point, local, and vector operators for operation level algorithms,
- components to design streaming pipelines for application level algorithms, and
- components to support Gaussian and Laplacian pyramids for Multiresolution Analysis (MRA).

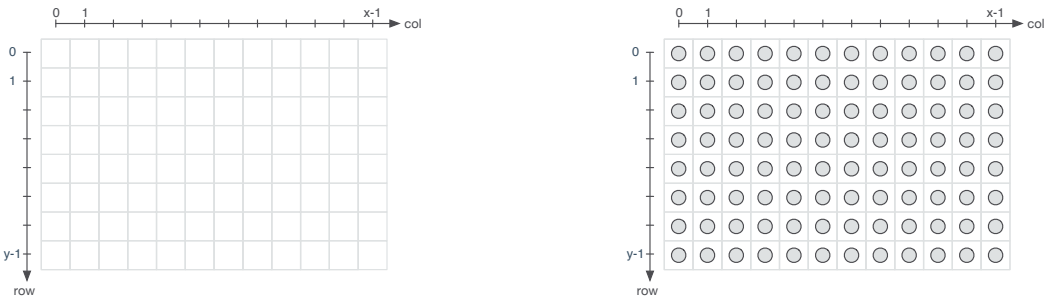
The proposed library sets appropriate synthesis directives in the source code and is designed to allow simplified customization for specific purposes and new application domains. The quality of image processing accelerators generated using the library-based approach is evaluated and compared to design implementations synthesized from a commercial image processing library as well as software-based accelerator technologies from mobile and high-performance computing. The results assert a higher throughput and more efficient resource utilization than the commercial solution, as well as a significantly higher throughput than the embedded accelerator and compellingly more efficient energy requirements than the High-Performance Computing (HPC) accelerator.

Parts of the contributions described in this chapter have been published as follows:

- A lightweight library of C++-embedded code templates supporting domain-specific HLS of image processing accelerator Intellectual Property (IP) cores was introduced in [Sch+14a*].
- Several optimizations and the application of synthesis directives to achieve higher performance and more efficient resource utilization than a comparable commercial solution in [Rei+14*].
- Finally, the implementation and optimization of accelerators for Multiresolution Analysis (MRA) in image processing and scientific computing are detailed in [Sch+14c*].

4.1. Preliminaries

For our analysis, we consider image processing at the low and intermediate levels of the abstraction hierarchy, according to Figure 2.1 in Section 2.2. The algorithms at operation level are either specified in C or C++ and coded by nested iterative loops. Such loop nests are defined by an *iteration space*, comprising a finite discrete Cartesian space with dimensionality equal to the loop nest level [Wol89]. We restrict the analysis to *rectangular* iteration spaces, which is the case if the limits of the inner loops are static and do not depend on the values of outer loops. The Cartesian space for a two-dimensional loop for image processing is shown in Figure 4.1(a) and the associated iteration space in Figure 4.1(b). The outer loop-level defines the so-called *scanline*, which is also the order in which pixels arrive in data streaming. If the specification follows Listing 4.1, where the outermost loop describes the image rows and the next inner level defines scanning of the columns of the image, the scanline is in *row-major* order (refer to Figure 4.2(a)).



(a) Image data. (b) Iteration space.

Figure 4.1.: Image data and iteration space defined by iterative nested loops.

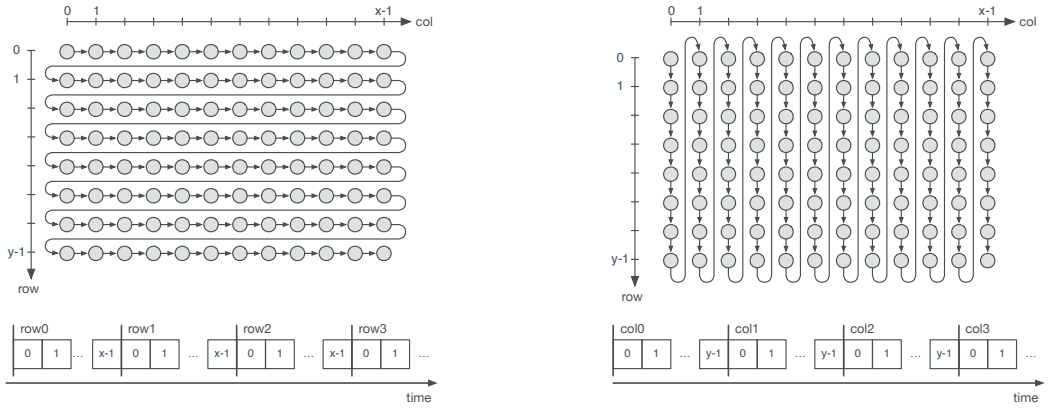
Listing 4.1: For loops to scan an image in row-major order.

```
1  for(int row = 0; row < y; row++){
2    for(int col = 0; col < x; col++){
3      ...
4  } }
```

In contrast, the scanline is in *column-major* order, if the outermost loop defines the columns and the next inner loop describes the image rows, as shown in Listing 4.2 (refer to Figure 4.2(b)).

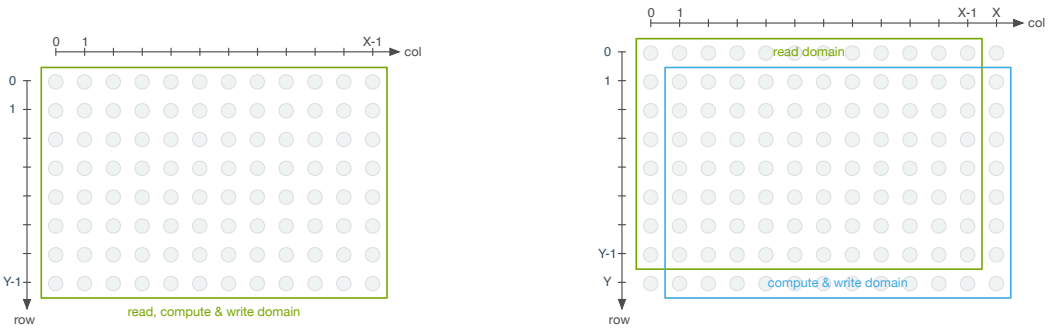
Listing 4.2: For loops to scan an image in column-major order.

```
1  for(int col = 0; col < x; col++){
2    for(int row = 0; row < y; row++){
3      ...
4  } }
```



(a) Row-major order. (b) Column-major order.

Figure 4.2.: Illustration of the scanline defined by the semantics of the outermost nested loops.



(a) Overlapping domains for point operators.

(b) Different domains for local operators.

Figure 4.3.: Point and local operators have different domains for reading, computing, and writing data.

If data is read from a stream at a specific point in the iteration space, it is not necessarily processed or written at the same point in time. Hence, we differentiate between several domains over the iteration space to clarify at which points in time data is accessed (read domain), at which points computations are performed (compute domain), and at which points data is written (write domain). For point operators, which only require the data at the current point in the iteration space, all three domains are the same, as depicted in Figure 4.3(a). For local operators, which require data from the local neighborhood for computation, we must insert an offset between the read and the compute domain, which must also be applied to the write domain and causes the iteration space to become larger, as shown in Figure 4.3(b). The reason for this is explained in Section 4.3.3.

4.2. Importance of Hardware-specific Coding Styles for C-based HLS

An often desired, but so far unfortunately unreached goal for HLS frameworks is to be able to generate high throughput hardware implementations from code that was developed by software developers for sequential execution on a microprocessor. Whereas the typically sequential software execution model greatly benefits from the high clock frequencies of general purpose Central Processing Units (CPUs), hardware execution must exploit parallelism contained in the program to allow lower clock frequencies and therefore a significant reduction in power requirement. Low and medium level image processing algorithms are often composed of loop programs, where the same instructions will be executed over and over again for

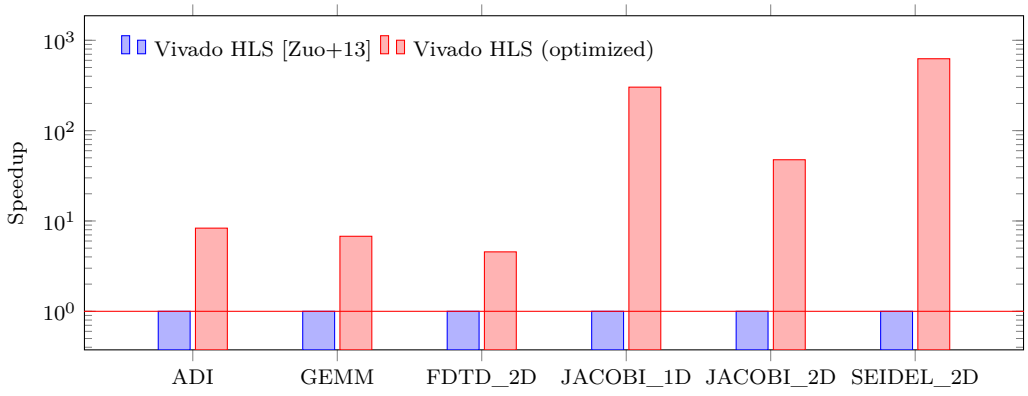


Figure 4.4.: Speedup achieved through applying several high-level compiler and architectural optimizations for selected algorithms of the PolyBench/C suite using Vivado HLS.

each pixel of the image. Pipelining and overlapping of the execution of these instructions is usually a first step to generate high throughput implementations. Furthermore, in the same way as caching can significantly reduce the effects of the memory bottleneck on CPU-based architectures, data streaming using First In, First Out (FIFO) interfaces is a means on FPGAs to considerably reduce the amount of cycles to fetch new pixels in comparison to reading from and writing to Dual-Ported Random Access Memory (DPRAM). It is sometimes also possible to increase the throughput by applying high-level compiler transformations, such as *loop interchange*, where the scanning order over the data elements of the iteration domain is adjusted to the algorithm's data dependencies, or *loop fusion*, which combines the executions from two or more loops in the same iteration domain into a single loop. Finally, parallelization may be exploited to execute independent loop iterations in parallel.

Tool-based code generation that automatically transforms C-code written for software targets into C-code for HLS was recently studied using algorithms from the polyhedral benchmark suite PolyBench/C[5], which is as a collection of benchmark codes for polyhedral code compilation. In [Zuo+13], the authors modify the C-based polyhedral code generation tool CLooG [Bas04] for HLS and apply it to several algorithms of the Polybench/C 3.2 collection. Although the authors can already reduce the latency using this approach, as detailed in [Zuo+13], the combination of data streaming and high-level transformations can still decrease the latency by up to several orders of magnitude, as it is illustrated in Figure 4.4

[5] *http://polybench.sourceforge.net*

Table 4.1 lists how the optimized algorithm implementations were achieved as well as the difference in Lines of Code (LoC) between the original and the optimized code. It can be observed that compiler optimizations, such as loop manipulations are effective techniques for performance improvement at no additional cost. The architectural optimizations that were applied to the last three algorithms are discussed in more detail in Section 4.3. It can already be seen here that these optimizations can deliver a tremendous boost in performance, although they come at the penalty of severely increasing the LoC. An increase

Table 4.1.: High-level compiler and architectural optimizations applied to selected algorithms from the PolyBench/C suite to improve the performance in HLS.

Algorithm	Optimization	Original LoC	Optimized LoC
ADI	Loop Interchange	41	41
GEMM	Loop Interchange	23	23
FDTD_2D	Loop Fusion	28	28
JACOBI_1D	Architectural Optimization	15	224
JACOBI_2D	Architectural Optimization	21	230
SEIDEL_2D	Architectural Optimization	17	226

in the LoC necessary to specify an algorithm does not only mean that it takes the developer longer to program, but it also exacerbates the chance for coding errors and requires more effort to maintain the algorithm.

A key element of our rapid prototyping approach is to make the implementation of FPGA accelerators in the image processing domain more accessible to algorithm developers without in-depth knowledge of the targeted platform. A very often used means to ease the burden of lengthy implementations is to supply libraries of highly optimized components that were designed for reuse. An example for such a library available as part of Vivado HLS is the partial port of the OpenCV library [Pul+12], which aims at delivering key architectural components and algorithm implementations for image processing on FPGAs [Xil13]. Yet, an often missed aspect of such libraries is adaptability of the components, which would make these applicable to a wide range of problems. An example is an efficient memory architecture for data streaming. Although the OpenCV library provides such an architecture for pipeline-based image processing, the implementation is very complicated and not intended for customization. In contrast, users would benefit the most from a library of highly efficient basic components, which provides basic building blocks instead of specific algorithms and may thus be essential to a wide range of applications.

4.3. Stream-based Image Processing on FPGAs

One of the key concepts of the here presented library is to provide means to productively develop accelerators for image processing using HLS and efficiently integrate these with the data streaming concept. Vivado HLS, for example, implements a specific `stream` class to represent streaming data communication using FIFO semantics. Our approach uses this streaming class as an interface for the hardware accelerators that can be created using HLS in combination with this library. In this way, we can seamlessly combine the library components to synthesize fully pipelined image processing algorithms and also prepare the synthesis of an interface to integrate the design with other key components of the FPGA fabric, such as memory controllers, and communication interfaces. Moreover, building on FIFO interfaces is a key concept to achieve a low pipelining interval, since they enable reading and writing new data elements in every clock cycle.

We introduce the essential building blocks to separate the above performance-critical artifacts of our library approach using a simple two-dimensional Gaussian convolution example as stated in Equation (2.1) in Section 2.3.1. Despite its simplicity, this example already presents many of the design challenges faced during the development of an efficient hardware accelerator using C-based HLS. These include (a) defining an appropriate memory architecture (refer to Section 4.3.2), (b) dealing with causality issues (refer to Section 4.3.3), and (c) providing a means for border treatment (refer to Section 4.3.4).

4.3.1. Motivational Example

In the C++ programming language, the Gaussian convolution for a neighborhood of size $w \times w$ can be stated rather easily using a two-dimensional array of coefficients, as shown in Listing 4.3.

Listing 4.3: Gaussian convolution using a kernel size of $w \times w$ pixels.

```
1  data_t coeff[w][w];
2  data_t gauss(data_t window[w][w])
3  {
4    data_t accu = 0;
5    for(int i = 0; i < w; i++)
6      for(int j = 0; j < w; j++)
7        accu += window[i][j]*coeff[i][j];
8    return accu;
9  }
```

In general, Listing 4.3 can be regarded as a template for two-dimensional convolution, not only for Gaussian kernels, but also, for example, for edge detection using a Sobel kernel.

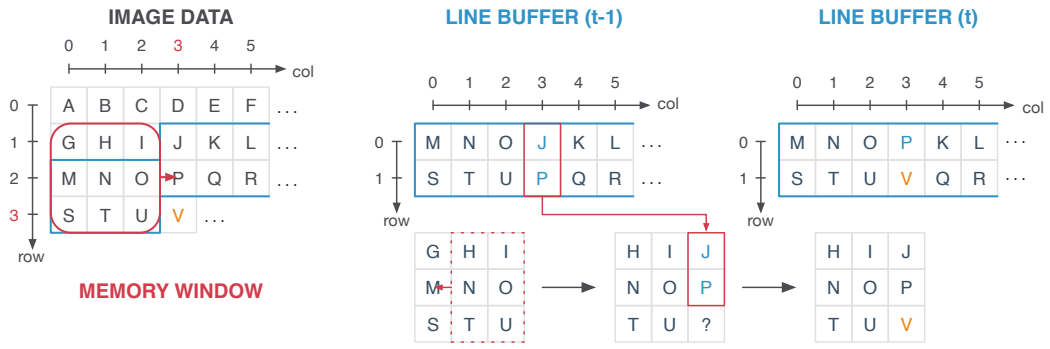


Figure 4.5.: An example of the interaction between a line buffer and a memory window. As the window traverses in a scanline from left to right over the image, the current window content is shifted to the left and the new column is inserted at the rightmost position.

4.3.2. Memory Architecture

Local operators process a local neighborhood, also referred to as the *window*, and therefore need to access a pixel repeatedly in successive iterations. Handling streaming data on an FPGA therefore requires a memory architecture to retain data for multiple accesses. Memory resources on modern FPGAs can be broadly categorized into Flipflops (FFs) and DPRAM. Whereas FFs can store only single bits and are a part of the Configurable Logic Blocks (CLBs) of the FPGA, DPRAM is organized in blocks and therefore commonly referred to as Block Random Access Memorys (BRAMs). As both types of memory resources are scarce on FPGAs, an efficient memory architecture for streaming data uses a combination of *line buffers*, implemented using BRAM, for storage of complete image lines and FF-based *memory windows* for the parallel processing of local neighborhoods.

A line buffer can typically be implemented as a two-dimensional array, however, several high-level synthesis frameworks also provide special classes for C++-based implementations. The interaction between the line buffer and the memory window is shown in Figure 4.5. New image data are stored in the line buffer, which is then used to update the memory window. In order to update the window, that is, to move it to the right, all of its contents are shifted to the left and the pixels of the rightmost column are obtained from the current column of the line buffer as well as the new input element. For pipelining, the code for the interaction between the line buffer and the memory window must be written in such a way that it can be completely unrolled and executed concurrently. Our solution for a memory window `wnd` of size $w \times w$ and a line buffer `lb` is shown in Listing 4.4.

Listing 4.4: Interaction between the memory window and the line buffer.

```
1  data_t lb[w-1][x]; // LINE BUFFER
2  #pragma HLS array_partition variable=lb dim=1 complete// PARTITION THE LB
3  data_t wnd[w][w]; // WINDOW
4  #pragma HLS array_partition variable=wnd dim=0 // PARTITION THE WINDOW
5
6  for(int row = 0; row < y; x++){
7    for(int col = 0; col < x; y++){
8
9      // UPDATE THE WINDOW
10     for(i = 0; i < w; i++)
11       #pragma HLS unroll
12       for(j = 0; j < w-1; j++)
13         wnd[i][j] = wnd[i][j+1];
14
15     // UPDATE THE LINE BUFFER
16     for(i = 0; i < w-1; i++){
17       #pragma HLS unroll
18       wnd[i][w-1] = lb[i][col];
19       if(i>0){
20         lb[i-1][col] = lb[i][col];
21       }
22     }
23
24     // INSERT NEW ELEMENT
25     wnd[w-1][w-1] = new_pix;
26     lb[w-2][col] = new_pix;
27   }
28 }
```

To enable an Initiation Interval (II) of one clock cycle, the interaction must not require more than two accesses to the BRAM-based line buffer per iteration. Our solution in Listing 4.4 achieves this by

(a) partitioning each line of the line buffer into a dedicated BRAM using the `array_partition` directive,

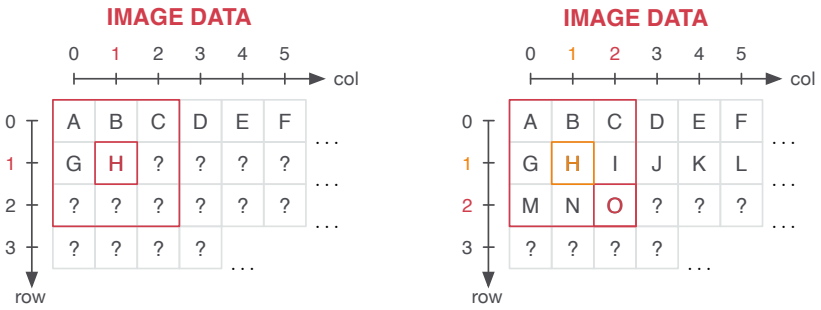
(b) completely unrolling the `for`-loops,

(c) interweaving copying data from the line buffer to the window with the line buffer update, and

(d) assigning the new pixel (`new_pix`) to the window directly instead of copying it from the line buffer.

4.3.3. Causality

Another challenge for stream-based image processing using local operators is that almost half of the data in the window is not available when processing the central pixel at index $(1, 1)$, as shown in Figure 4.6(a). This means the filter is acausal because it uses some samples from the *future*. In practice, only causal filters may be used, so the response of the filter must be shifted by $\tau_x = \tau_y = \frac{w-1}{2}$ steps into the past. A solution, given in Figure 4.6(b), is to simply wait until



(a) Missing pixels in the window at iteration $(1, 1)$.

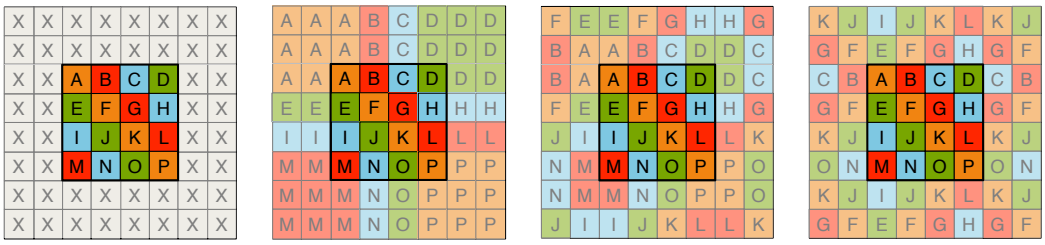
(b) Completely filled window at iteration $(2, 2)$.

Figure 4.6.: Data causality in window-based image processing for streaming data.

all the data for the window are available, which is the case at index $(2, 2)$ and increases the so-called *group delay* by $\tau_g = (\tau_y \cdot w) + \tau_x$ pixels (in the context of a linear stream of pixels). We compensate this by discarding the first τ_g output samples which means shifting the output sequence τ_g samples *back* into the future. As a consequence, the output data are delayed in order to assign the value computed at $(2, 2)$ to its correct location in the output image at $(1, 1)$.

4.3.4. Border Handling

In addition to causality issues, we also have to deal with processing the image borders. Here, the window might covers a region which lies partially outside of the image. For example, if the neighborhood around position $(0, 0)$ is processed with a 3×3 window, the values to the left and above (the negative indices) are missing. Just skipping over these positions is usually not an option since it may either decrease the output image by nearly half of the window size in each direction or leave a visible frame around the processed region, depending on whether unprocessed pixels are removed or retained in the output image. Figure 4.7 shows several *border treatment* techniques for handling obtaining the missing information from available image data. Filling up the missing pixels with a constant value, as shown in Figure 4.7(a) tends to *pull* the filtered value towards this constant. For example, choosing zero as the constant causes the image to become darker at the borders. The procedures shown in Figures 4.7(b) to (d) mostly give acceptable results, meaning that there is no perceptible degradation towards the borders of the image. Algorithms in medical image processing usually apply the *mirroring* border treatment techniques, shown in Figures 4.7(c) and 4.7(d). The first design question for border handling is whether the reordering of pixels should take place when inserting input data



(a) Constant (b) Clamp (c) Mirror (d) Mirror 101

Figure 4.7.: Border treatment variants for window-based image processing for windows of size 5×5. Algorithms in medical image processing usually apply the *mirroring* border treatment techniques, shown in Figures 4.7(c) and 4.7(d)

into the line buffer, when loading pixels from the line buffer into the register window, or as a additional processing of the window. DPRAM, which is used to implement the line buffer only allows a maximum of two memory accesses per clock cycle, one per port. Restructuring pixels for border treatment already in the line buffer is therefore not practical. Including border treatment during loading new pixels into the memory window might also require more than two memory accesses, which could also hinder achieving a small initiation interval. The best option to perform border treatment is therefore to restructure the information in the memory window.

Handcrafted code often states border treatment explicitly. Depending on the size of the window and how the missing pixels are obtained, the necessary code can become very long. An alternative to the explicit statement is to implement border treatment as a function, which is inlined for synthesis.

4.3.5. Filter Assembly

Under consideration of the aforementioned aspects, we can assemble the filter structure to obtain a hardware accelerator for the two-dimensional Gaussian convolution. We read the input image pixel by pixel from the input stream and store the data in the line buffer. The line buffer is hereby updated in every new iteration in combination with the memory window. Before we compute the 3×3 Gaussian convolution for every pixel using the neighborhood pixels as provided by the memory window, the border treatment function restructures the content of the memory window according to the chosen border treatment technique, if necessary. After the computation, the processed pixel is written to the output stream. The assembly is encapsulated in the `process`-function, shown the following Listing 4.5, which also serves as the entry point for subsequent high-level synthesis.

Listing 4.5: Filter assembly for streaming data computation of the 2D Gaussian convolution, including the proposed solutions for the memory structure, causality, and border treatment.

```
1   #define GRP_DLY 1 // group delay
2   void process(
3     hls::stream<data_t> &img_in, // input stream
4     hls::stream<data_t> &img_out) // output stream
5   {
6     data_t[2][n] lb; // line buffer
7     data_t[3][3] wnd; // memory window
8
9     data_t in_pix, out_pix;
10
11    ROW_LOOP: for(int row = 0; row < m+GRP_DLY; row++){
12      COL_LOOP: for(int col = 0; col < n+GRP_DLY; col++){
13        #pragma HLS pipeline ii=1
14        #pragma HLS inline region
15        //READ NEW PIXEL
16        if(row < m && col < n)
17        {
18          new_pix << img_in;
19          // BUFFERING AND BORDER TREATMENT
            ...
181       }
182       if(row >= GRP_DLY && col >= GRP_DLY)
183       {
184         // COMPUTE CONVOLUTION
185         out_pix = gauss(wnd);
186         // WRITE OUTPUT PIXEL
187         img_out << out_pix;
188       }
189     }
190   }
191 }
```

As Listing 4.5 shows, the complete code may easily exceed 200 lines and poses many chances for coding errors. In contrast, the next section will demonstrate how image processing operators, such as the local operator used in this motivational example can be specified in a very compact way while remaining easily readable by proposing a library.

4.3.6. Library Integration

The Gaussian convolution, as shown in Listing 4.5 can be specified using the here proposed library with the help of two of its basic components:

(a) A class for 1D and 2D convolution, called `ConstWindow`, and

(b) the function `proc_locOp` for the specification of local operators.

The library hereby encapsulates our solution for the memory architecture, as proposed in Listing 4.4, as well as handles causality performs and the chosen method for border treatment. The specification of the example in Listing 4.5 then simplifies to the code shown in Listing 4.6.

Listing 4.6: Specification of the Gaussian filter using the function `proc_locOp` and the `ConstWindow` class of the proposed library.

```
1  void gauss(
2    hls::stream &img_in,
3    hls::stream &img_out)
4  {
5    ConstWindow<data_t> coefficients(Gauss_3x3);
6    proc_locOp(img_in, img_out, BorderTreatment::BORDER_MIRROR, coefficients);
7  }
```

The `proc_locOp` function takes as arguments (a) the data streams for input and output, (b) the desired method for border treatment, and (c) the name of the local operator itself. In this example, the local operator is an instance of the `ConstWindow` class, which overloads the `()`-operator and can thus be called just like a function. The constructor takes the coefficients for a 3×3 Gaussian kernel and computes the convolution. In essence, the `proc_locOp` function can handle any local operator that is specified in the form of a class in a similar fashion as the `ConstWindow` class. In the next section, we show how to define point and local operators, as well as how they can make use of the proposed library.

4.4. Specification of Point and Local Operators

Very important basic operators for image processing at low abstraction levels involving mostly pixel operations are *point*, *local*, and *global operators*. All of these operators are implemented and customized in the proposed library, as follows.

4.4.1. Point Operators

Typically, point operators are used to perform tasks such as inversion, brightness, and contrast alterations, or color scheme transformations. Just as the library contains the `proc_locOp` method to process a local operator, it also contains a specific function for point operators. For image transformations, such as to increase the brightness of an image by multiplying the intensity of each pixel by a factor, a point operator can be specified as follows.

Listing 4.7: Point operator for brightness increase.

```
1  template<typename T> class PntOpBright{
2    const T factor;
3    PntOpBright(const T _factor) : factor(_factor) {}
4    T operator() (const T &pixel) const{ return pixel*factor; }
5  };
6
7  void top_level(hls::stream<data_t> &img_in, hls::stream<data_t> &img_out){
8    PntOpBright<data_t> pnt_operator(1.1);
9    proc_pntOp(img_in, img_out, pnt_operator);
10 }
```

Moreover, point operators are also used to in the streaming infrastructure to copy or merge different streams. Point operators can be specified as Multiple Input, Multiple Output (MIMO) systems, as shown for stream duplication in Listing 4.8.

Listing 4.8: Point operator for stream duplication.

```
1  void splitStream(
2    hls::stream<data_t> &img_in,
3    hls::stream<data_t> &img_out_1,
4    hls::stream<data_t> &img_out_2)
5  {
6    const data_t in_pix;
7    for(int i = 0; i < width*height; ++i){
8      #pragma HLS PIPELINE ii=1
9      img_in >> in_pix;
10     img_out_1 << in_pix;
11     img_out_2 << in_pix;
12   }
13 }
```

Note that point operators, if used in conjunction with local operators, must respect the increased loop bounds due to causality issues, as shown in Section 4.3.3.

4.4.2. Local Operators

Using *local operators* as in filtering an image with a constant kernel is one of the most common tasks. It is a discrete 2D convolution of the image and the kernel, basically a 2D FIR filter and is therefore a linear operation. Typical kernels are, for example, a Gaussian, rectangular, or Hann window for noise suppression, and Sobel or Laplacian operators for edge detection.

We provide two simple ways of implementing local operators in our library. One method is to specify the weights in the source code which will be synthesized and therefore cannot be modified afterwards. Another method is to pass them as interface parameters prior to the image processing, so the weights can be changed between the processing of two images. For convenience, we have predefined frequently used kernels (such as Gaussian, Sobel, and averaging kernels) to elude the process of repeatedly typing these in.

The implementation of such a kernel (here, for example, using a 5×5 coefficient mask for a Gaussian filter) can be kept rather short, as shown in Listing 4.9.

Listing 4.9: Specification of a convolution kernel using a predefined coefficient window of the `ConstWindow` class.

```
1  ConstWindow<float> coefficients(Gauss5x5);
```

Defining a custom kernel only requires to add a definition of the kernel elements. As an example, Listing 4.10 shows the specification of a two-dimensional *emboss kernel*.

Listing 4.10: Specification of a custom convolution kernel.

```
const float emboss_krnl[] = {
    -2.0f, -1.0f, 0.0f,
    -1.0f,  1.0f, 1.0f,
     0.0f,  2.0f, 2.0f
};
ConstWindow<float> emboss(emboss_krnl);
proc_locOp(img_in, img_out,
    width, height,
    BorderTreatment::BORDER_CLAMP,
    emboss);
```

Apart from two-dimensional convolution, the proposed approach is also suitable for other image processing operations that can be carried out using local operators. A prime example are *rank-order filters*, which are non-linear filters that can be used to remove salt-and-pepper noise from images. Rank-order filters sort all values (which is the non-linear part) from the image region and return the k-th element (or p-quantile). A popular rank-order filter is the *median filter*, which always takes the element at 0.5-quantile. Sorting in integrated circuits usually does not utilize *classical* software algorithms like *QuickSort* but instead makes use of concurrent sorting networks, for example, the *odd-even merge* network [KT11].

To give an example, we show the implementation of a median filter based on the *odd-even transposition* sorting network [Knu98]. Just as for the convolution, the functionality is encoded in a class, which has to provide a specific interface in form of a function that takes a window buffer as argument.

Listing 4.11: Local operator for a median filter.

```
template<typename T>
class MedianFilter{
public:
  T operator()(const T wnd[7][7]) const{
    static const int N = 7 * 7;
    data_t z[N]; data_t t[N];
    int i = 0, j = 0, k = 0, stage = 0

    //serialize the window for sorting
    for (i = 0; i < 7; ++i){
      #pragma HLS unroll
      for (j = 0; j < 7; ++j)
        z[i*7+j] = wnd[i][j];
    }

    // sorting network loop
    for (stage = 1; stage <= N/2; ++stage) {
      #pragma HLS unroll
      k = (stage % 2) == 0 ? 0 : 1;
      for (i = k; i < N-1; i += 2) {
        t[i]   = std::min(z[i], z[i+1]);
        t[i+1] = std::max(z[i], z[i+1]);
        z[i]   = t[i];
        z[i+1] = t[i+1];
    } }
    return z[N/2];
} };
```

For median filter defined in this way, we do not have to generate the full sorting network with $N = 49$ stages, but can stop after the first $\lfloor N/2 \rfloor = 24$ stages. The median filter can be included in an image processing accelerator as follows.

Listing 4.12: Execution of the median filter.

```
1  MedianFilter<float> median();
2  proc_locOp(img_in, img_out, width, height,
3    BorderTreatment::BORDER_CLAMP, median);
```

4.5. Parallelization and Design Optimization

Although the possibility to use a C-based language for design entry lowers the hurdle for acceptance of an HLS framework, algorithms stated in such a language are inherently sequential and must be parallelized and optimized in order to efficiently use the FPGAs resources. As opposed to the world of HPC, where the fastest processing speed is paramount, developing hardware accelerators may be subject to several contrasting design goals. Of course, high throughput and short clock periods are important achievements, but often it is also necessary to comply with a certain resource budget. In the following, we discuss the placement of synthesis directives (refer to Section 3.3.6), which allow to specify how the input design is to be parallelized and optimized, as well as several other optimizations to achieve a high quality of the synthesis results in terms of throughput and resource usage.

4.5.1. Vivado Synthesis Directives and Coding Considerations

In order to exploit the architecture of the FPGA target during HLS, several synthesis directives must be either inserted in the code directly as *pragmas*, or collected in a script file which are then applied during synthesis. Certain directives, such as for partitioning the line buffer and memory window, unrolling `for`-loops, or to optimize hierarchical structures by *inlining* calls to small functions, are rarely changed and are thus placed directly in the source code of the library as pragmas. Examples for the placement of these directives are shown in Listing 4.4 and Listing 4.5.

Several other directives can be used for design space exploration or to enforce a certain optimization strategy. Yet, searching through the code to find and alter these directives may become inconvenient for large designs. A very important example for such a directive to transform the sequential specification into a typical parallel hardware structure is the `pipeline` directive, which is also used to control the II of a loop, as shown in Listing 4.5. To enable pipelining in Vivado HLS, however, the kernel size, as well as the iteration space must be

known at compile time. We have chosen to use C++ templates to specify such information, since the templated information is available during compilation.

An example for a templated function to implement a local operator is depicted in Listing 4.13

Listing 4.13: Use of templates for design parameters.

```
1  template<int II_TARGET, int MAX_WIDTH, int MAX_HEIGHT, int KERNEL_SIZE,
     typename IN, typename OUT, class Filter>
2  void process(...)
```

Whereas template arguments for sizes and data types can be used directly in the code, applying the template argument for Vivado HLS pragmas requires involving the preprocessor, as shown in the following.

Listing 4.14: Defines for pragma substitution.

```
1  #define PRAGMA_SUB(x) _Pragma (#x)
2  #define PRAGMA_HLS(x) PRAGMA_SUB(x)
```

The template argument can then be used in the library code, for example, to control the pipeline rate, as follows.

Listing 4.15: Example of using a template argument to control the pipeline interval.

```
1  for(int row = 0; row < height; row++){
2    for(int col = 0; col < width; col++){
3      PRAGMA_HLS(HLS pipeline ii=II_TARGET)
```

In this way, the directives are inserted at the correct locations in the source code but can be easily altered by the designer.

4.5.2. Optimizing Loop Counter Variables

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

such variables. As shown in Listing 4.16, the `assert` statements assure that the value used during runtime never exceed the maximum value known at compile time.

Listing 4.16: Example of using `assert` statements to specify the maximum value of a variable for automatic bit-width optimization.

```
1  template<int II_TARGET, int MAX_WIDTH, int MAX_HEIGHT, int KERNEL_SIZE,
     typename IN, typename OUT, class Filter>
2  void proc_LocOp(
3    hls::stream<IN> &img_in,
4    hls::stream<OUT> &img_out,
5    const int &width,
6    const int &height,
7    Filter &filter,
8    const enum BorderPadding::values borderPadding)
9  {
10   assert(width <= MAX_WIDTH + GROUP_DELAY);
11   assert(height <= MAX_HEIGHT + GROUP_DELAY);
12   ...
13   process_main_loop:
14   for (int row = 0; row < height; row++) {
15     for (int col = 0; col < width; col++) {
16       PRAGMA_HLS(HLS pipeline ii=II_TARGET)
17       #pragma HLS INLINE region
18       ...
19     }
20   }
21 }
```

4.5.3. Mapping Vector Types

Vector types are a common way to accelerate computations on Single Instruction, Multiple Data (SIMD) architectures. Mathematical operations are not carried out by scalar operators, but by vector operations. Besides performance concerns tackled by explicit vectorization, these types are particularly well-suited for expressing computations on common formats of image processing in a natural way.

To support vector types in Vivado HLS, basically two approaches are available. All vector types can be treated separately, which would result in multiple line buffers, multiple windows, and multiple streams for each local, or point operator. An improved solution is to pack the individual components into a vector data type, resulting in only a single stream for each operator. The first approach is realized by the *HLS video library* provided with Vivado HLS[6]. The latter approach is followed by our library, which may reduce the overall consumption of memory resources.

We have implemented vector types for Vivado HLS as C structures and realize computations by operator overloading. Reads and writes to stream objects are replaced by conversion functions, either extracting the vector types from

[6] http://www.wiki.xilinx.com/HLS+Video+Library

packed stream elements or packing vector types into stream elements. Inter-kernel computations, i.e. operations on local variables, are described as vector operations that are covered by overloaded operators.

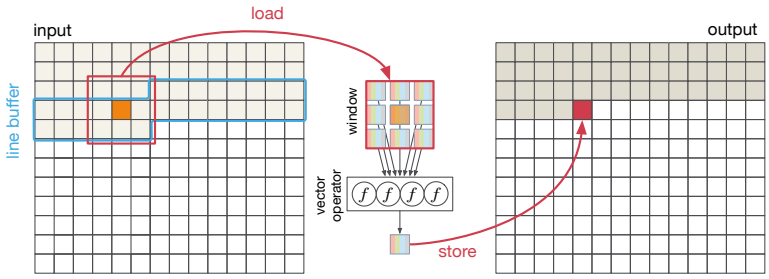


Figure 4.8.: Processing vector types with a local operator by substituting the scalar operations by vector operations.

4.5.4. Optimizations for Streaming Pipelines

As mentioned earlier, image processing algorithms usually consist of a sequence of individual steps. An optimal hardware implementation can be achieved by constructing a so-called *streaming pipeline*, which applies point and local operators to data concurrently as soon as it is available. In comparison to a sequential store and process methodology, this approach can avoid having to provide memory for complete images and severely increases the degree of parallelism.

An often evaluated example for a streaming pipeline image processing algorithm is the Harris corner detector, which was first introduced in [HS88], and consists of a pipeline of kernels, implemented using point and local operators, as depicted in Figure 4.9. After building up the horizontal and vertical derivatives (dx, dy), the results are squared and multiplied (sx, sy, sxy), and processed by Gaussian blurs (gx, gy, gxy). The last step (hc) computes the determinant and trace, which is used to detect when the threshold is exceeded. Kernels are interconnected with each other using stream objects implementing FIFO semantics. Vivado HLS can then be instructed to run all kernels in parallel using the *dataflow* directive, as shown in 4.10 (left), which can deliver a significantly shorter processing time than sequential execution (right).

4.6. Evaluation

In order to asses the performance of the image processing accelerators developed with the library, we have evaluated several algorithms from the image processing domain, spanning from simple two-dimensional convolution to a complex filter

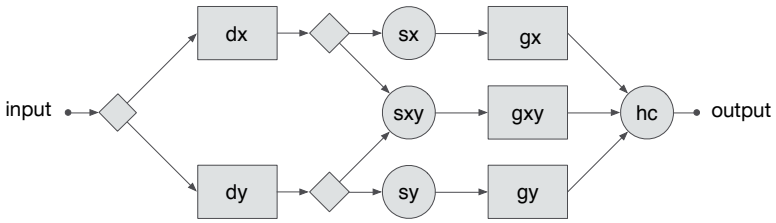


Figure 4.9.: Sequential execution of kernels for the Harris corner detector implemented as a pipeline of point operators (circles) and local operators (squares).

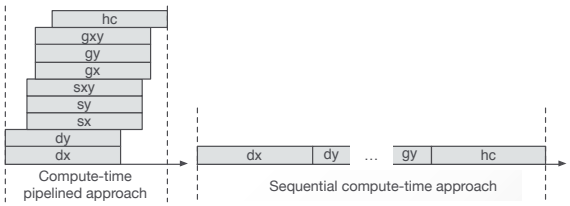


Figure 4.10.: Pipelined approach using the `dataflow` directive and sequential execution.

pipeline. A very prominent example for a similar approach to facilitate image processing on FPGAs using HLS is the Open Source Computer Vision (OpenCV) library [Pul+12], which is included with Vivado HLS. We will first introduce the environment in Section 4.6.1 and the evaluated algorithms in Section 4.6.2 before presenting experimental results in subsequent sections.

4.6.1. Evaluation Environment

Xilinx Kintex 7 XC7K325TFFG900

The here used Kintex 7 is a mid-range FPGA of the Xilinx Kintex 7 family, which is used on the Xilinx KC705 development platform. The chip is produced in 28 nm technology and integrates over 326 K logic cells, 445 32 Kb BRAMs, and 840 Digital Signal Processor (DSP) slices. In addition, the FPGA is aimed at connectivity, providing a hard block for Peripheral Component Interconnect Express (PCIe) Gen2 and 16 Multi-Gigabit Transceivers (MGTs), each providing transfer rates of up to 12.5 Gb/s.

Xilinx OpenCV

The Xilinx version of the OpenCV library is a Vivado HLS-specific video processing library, similar to the popular computer vision framework. Although the library is designed to be used exclusively with the AXI4 streaming interfaces, we have extended it to also support standard streaming interconnects. In its current

state, only a subset of the algorithms offered by OpenCV have been ported by Xilinx for synthesis using Vivado HLS. According to the availability at the time of the conducted experiments, we have used different versions of the OpenCV library, which are indicated for each experiment.

4.6.2. Algorithms

For the experiments, we consider several of the image processing algorithms described in Section 2.3, of which most are currently also available in the Xilinx version of OpenCV. All of these algorithms embody a high degree of parallelism and are therefore popular targets for acceleration on FPGAs. Although these algorithms are well known, their implementation details on the FPGA may differ significantly, thus we briefly clarify the algorithm implementation used for our evaluation.

Gaussian

The Gaussian (G) filter is a common example for two-dimensional convolution and was already introduced in Section 4.3 as motivational example. For this assessment, we have evaluated the filter with mask sizes of 3×3 (G3), 5×5 (G5), as well as 7×7 (G7) single-precision floating-point coefficients.

Laplace

The Laplacian (LP) filter is based on a local operator. Depending on the mask variant used, either horizontal and vertical edges or both including diagonal edges can be detected. In our results, we denote the first variant as LP3HV and the second as LP3D. Both use a mask size of 3×3. Additionally, we have evaluated a third variant, LP5, which uses a mask size of 5×5 and can also detect diagonal edges.

Harris Corner

The Harris Corner (HC) detector serves as an example of a complex image pipeline, depicted in Figure 4.9. Since the Harris corner detector is available as a complete algorithm implementation in the OpenCV library, we can compare the results from our approach directly to it. It uses a mask size of 3×3 and is therefore denoted as HC3.

Optical Flow

The Optical Flow (OF) algorithm issues a Gaussian blur and computes signatures for two input images using the census transform. Therefore, for each image

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

For the evaluation, we have used single-precision floating-point data types for the Gaussian filter as well as 8-bit integer data types for the other algorithms. As input we have use images of size 1024×1024 pixels. The algorithms were synthesized in Vivado HLS with high effort settings for scheduling and binding. Post Place and Route (PPnR) resource requirements were obtained by implementing the generated VHDL code as out-of-context modules using the Vivado Design Suite in version 2014.1, which also matches the evaluated version of OpenCV. Power requirements were assessed by performing a timing simulation on the post-route simulation netlists and evaluation of the switching activity in Vivado. The evaluation results include achieved minimum initiation interval (II) and latency (LAT) (total number of clock cycles), required slices (SLICE), lookup tables (LUT), flip-flops (FF), block RAMs (BRAM), and DSP slices (DSP). Each 7 series FPGA slice contains four Lookup Tables (LUTs) and eight FFs. BRAMs are 32 Kb in size, however, each block can also be used as two independent 18 Kb blocks. In the result listings in this work, we always provide BRAM count in terms of 32 Kb blocks. Moreover, the results also specify the maximum achievable clock frequency (F [MHz]), throughput (TP [fps]), and simulated power requirements (P [mW]). The throughput is evaluated using the latency (LAT) and the clock frequency (F) of the PPnR implementation as

$$TP = \frac{F}{LAT} \tag{4.1}$$

4.6.3. Comparison with OpenCV

In order to assess the performance of the image processing accelerators developed using our proposed library, we have first evaluated two-dimensional convolution with Gaussian (G) kernels using a single-precision floating-point data type. For these kernels, Vivado HLS contains an explicit OpenCV implementation, we were able to compare our results to, as shown in Table 4.2. The comparison of the two-dimensional convolution kernels between our approach and OpenCV shows that for the used version of Vivado HLS, our approach can produce faster accelerators at more efficient resource usage, in terms of choosing DSP slices for arithmetic operations over LUTs. The reason for this is that our implementation performs

Table 4.2.: Comparison of Gaussian filter implementations using OpenCV and the here proposed approach.

	Proposed Library									
	II	LAT	SLICE	LUT	FF	BRAM	DSP	F [MHz]	TP [fps]	P [mW]
G3	1	1050702	1759	3909	5314	2	45	226.9	215.9	221
G5	1	1052873	4865	10908	14634	4	125	215.1	204.3	350
G7	1	1055107	11140	22285	28702	6	246	213.9	202.7	463
	OpenCV 2014.1									
	II	LAT	SLICE	LUT	FF	BRAM	DSP	F [MHz]	TP [fps]	P [mW]
G3	1	1082517	3625	10293	6437	3	30	175.9	162.5	226
G5	1	1088785	8876	26893	16528	5	75	165.8	152.3	362
G7	1	1097149	16027	50868	31700	7	147	176.4	160.8	546

all computations using floating-point arithmetic, which is mostly implemented using DSP slices, whereas the OpenCV approach uses fixed-point data types internally, which use more LUTs for implementation.

To give an idea of how productively the accelerators may be developed, Table 4.3 lists the required LoC to specify the image filters. As the Gaussian filter can be computed using predefined coefficient windows, the required code is quite short, if compared to a hand coded (h/c) implementation. As the table also lists the LoC for the specification of the filter in OpenCV, it can be seen that our approach is almost equally compact.

Table 4.3.: Lines of Code for the evaluated filters.

	Gauss (h/c)	Gauss	Gauss/OpenCV
LoC	187	31	29

To evaluate our approach for vector operators, we have used the Laplacian filter to process Red Green Blue Alpha (RGBA) images, which use 8-bit integer data types for the individual channels. The experimental results are listed in Table 4.4. In contrast to our approach, which uses a common line buffer and memory window for the channels and only divides these when applying the actual operator, the OpenCV implementation uses individual line buffers for each channel. We can also observe that the OpenCV implementation makes more use of logic resources for arithmetic operations instead of DSP blocks. Especially for the smaller kernel sizes, our approach can provide highly more efficient implementations than the Xilinx library in terms of resource usage and throughput. Although we can reach a higher throughput with the 5×5 kernel, the OpenCV implementation shows its advantage of using a different data type internally by requiring only half of the FFs and DSPs.

Table 4.4.: Comparison of Laplacian filter implementations using OpenCV and the here proposed approach.

	Proposed Library									
	II	LAT	SLICE	LUT	FF	BRAM	DSP	F [MHz]	TP [fps]	P [mW]
LP3HV	1	1050638	141	288	521	2	0	349.9	333.0	232
LP3D	1	1050641	226	398	1034	2	0	341.1	324.7	232
LP5	1	1052768	3917	4521	23795	4	200	220.1	209.1	247
	OpenCV 2014.1									
	II	LAT	SLICE	LUT	FF	BRAM	DSP	F [MHz]	TP [fps]	P [mW]
LP3HV	1	1092846	1659	3515	8772	12	75	258.3	236.4	260
LP3D	1	1092846	1717	3470	8768	12	75	247.2	226.2	258
LP5	1	1098218	2171	5316	9076	10	103	201.7	183.7	268

We have implemented the Harris corner detector and the Optical Flow algorithm as examples for image filter pipelines. Although we were able to compare the Harris corner algorithm to an implementation in OpenCV, the Optical Flow is not yet part of the Vivado HLS library. The results of the implementation using our proposed library in comparison to the Xilinx library are shown in Table 4.5. Whereas our approach can achieve a minimum II of one clock cycle,

Table 4.5.: Comparison of Harris corner detector implementations using OpenCV and the here proposed approach.

	Proposed Library									
	II	LAT	SLICE	LUT	FF	BRAM	DSP	F [MHz]	TP [fps]	P [mW]
HC3	1	1063233	9349	23331	31102	8	254	239.4	225.2	498
HC3	2	2101442	6097	17129	21138	8	154	207.9	98.9	552
OF	1	1063233	5669	11321	18587	62	0	204.7	192.53	470
	OpenCV 2014.1									
	II	LAT	SLICE	LUT	FF	BRAM	DSP	F [MHz]	TP [fps]	P [mW]
HC3	2	2349003	6782	20037	26259	12	84	224.1	95.4	428

we were not able to reach an II below two clock cycles for the OpenCV version of the Harris Corner. To be able to compare the resource usage and performance, however, we also created a version of our algorithm that is scheduled with an II of two clock cycles. Due to the improved pipelining, our low II approach can clearly outperform the OpenCV implementation in terms of throughput. Although the Xilinx version can reach a higher clock frequency for the two clock cycle II version, it has a higher latency than our approach, so that the version created with our library can also achieve a higher throughput.

To give an overall view of the experimental results, Figure 4.11 compares the resource usage and throughput of the individual algorithms. The illustration

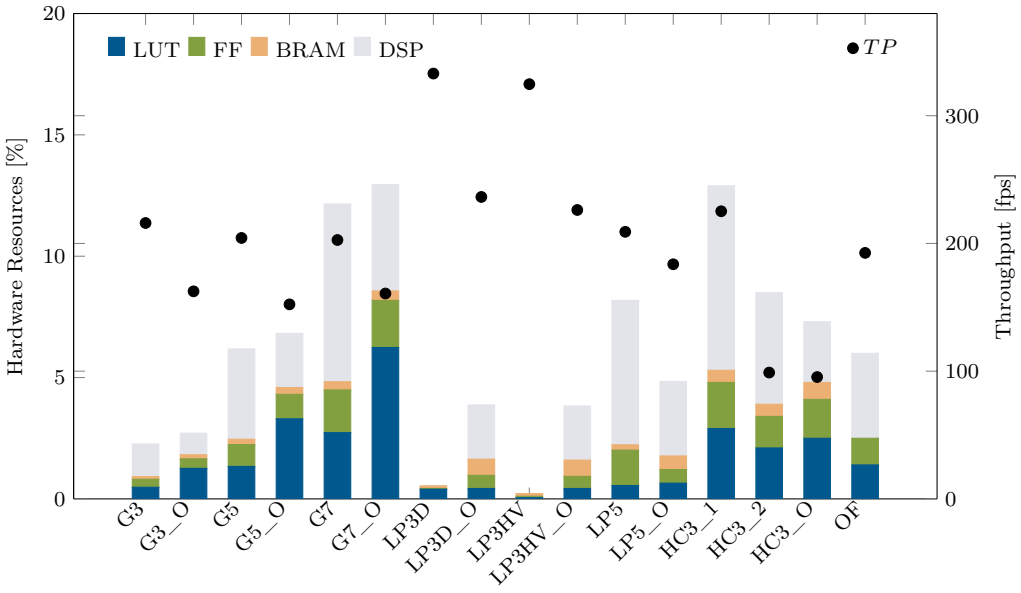


Figure 4.11.: Comparison of the amount of used FPGA resources and achievable throughput in frames per second (fps). The suffix O denotes OpenCV.

shows that we can achieve a higher throughput with all of our implementations in comparison to the OpenCV solutions. In most cases, however, this comes at the price of a higher resource utilization. Since our approach does not provide complete algorithms, but building blocks for their implementation, we believe that our approach opens up more chance for customization due to its low hierarchy. For example, a designer can easily choose which data type is used for the internal calculations by explicitly specifying it. For the OpenCV solution, the designer has no direct influence on how the algorithm works internally.

4.6.4. Comparison to Other Accelerator Technologies

In this section, we will analyze the above presented FPGA implementations in comparison to two software-based accelerators, the ARM Mali-T604, an embedded GPU, and the Tesla K20, which is server-grade GPU, in terms of performance and energy consumption. We will first briefly introduce the platforms.

ARM Mali-T604

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

Nvidia Tesla K20

The Nvidia accelerator is a discrete GPU. In contrast to consumer products for graphics acceleration it has no video output and is therefore specifically targeted at General Purpose Computation on Graphics Processing Unit (GPGPU). It contains 2496 scalar compute cores, distributed among 13 multi-processors, each running at 705 MHz. This results in approximately 3520 GFLOPs, doubled as usual, because of FMA. Besides 5 GB of GDDR5 RAM, it also includes L1 and L2 caches.

Comparison

The algorithm implementations for the GPUs were obtained using *HIPAcc*, a domain-specific language framework, developed solely for the purpose of providing highly optimized image processing implementations for heterogeneous accelerator targets [Mem+15b]. The performance results, comparing the two GPUs to the above presented results for the Kintex 7 FPGA are presented in Table 4.6. The throughput and energy efficiency are moreover compared in Figures 5.23 and 5.24, respectively.

Table 4.6.: Comparison of throughput and energy consumption for the ARM Mali-T604, Xilinx Kintex 7, and Nvidia Tesla K20.

	Mali-T604		Kintex 7		Tesla K20	
	TP [fps]	E [fpW]	TP [fps]	E [fpW]	TP [fps]	E [fpW]
LP3HV	100.4	41.8	333.3	1423.1	10000.0	74.1
LP3D	62.2	25.9	324.7	1387.6	6250.0	46.3
LP5	22.0	9.2	209.2	846.9	2631.6	19.5
HC	11.9	4.9	225.3	458.5	1098.9	8.1
OF	0.4	0.2	192.5	409.6	452.5	3.4

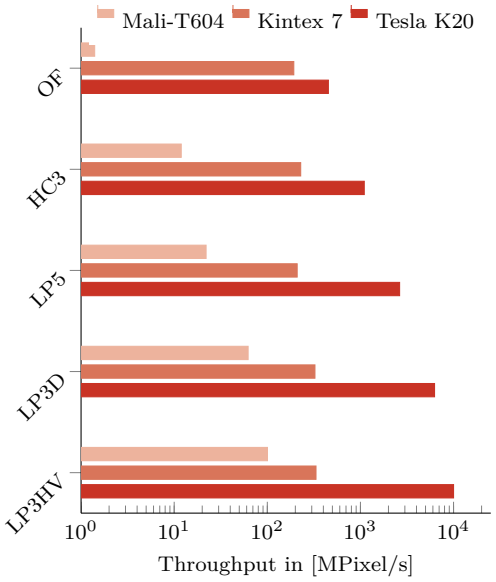


Figure 4.12.: Throughput comparison.

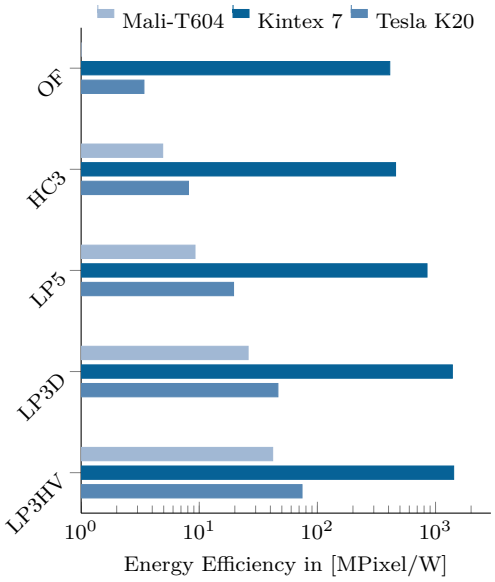


Figure 4.13.: Energy efficiency.

The most efficient configuration for the GPU implementations was found using the design space exploration feature of HIPAcc. Except for the optical flow, the Tesla K20 can exceed the performance of the embedded Mali GPU by a factor of approximately 100, and also the FPGA implementation clearly. The window size of 15×15 used in the optical flow leads to devastating results for Mali, presumingly because of missing L1 cache and on-chip memory. The throughput for both GPUs decreases with the window size and kernel complexity, clearly decelerated by the number of memory loads. For the FPGA, the impact of larger window sizes is by far less noticeable.

The power consumption of the GPUs can be estimated by considering about 60 % of the reported peak power values (2.28 W for the ARM Mali and about 135 W for the Nvidia Tesla). The FPGA provides the most energy efficient solution. Even though achievable frame rate per watt decimates notably faster than throughput, this effect is much more distinctive on GPUs, giving FPGAs a clear advantage in particular for more complex algorithms.

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

results of Tesla GPUs are not affected by this problem and therefore outperform Mali in terms of energy efficiency.

4.7. Image Pyramids for Multiresolution Analysis

Gaussian and Laplacian pyramids are a fundamental concept in multi-rate image processing and a natural playground to demonstrate the benefits of high-level synthesis compared to traditional hardware design. As explained in Section 2.3, the Gaussian pyramid consists of low-pass filtered, reduced density images of the preceding stage of the pyramid. The Laplacian pyramid can be obtained from a Gaussian pyramid by upsampling the Gaussian image to its original dimension and subtracting it from the Gaussian image of the previous stage.

4.7.1. Up- and Downsampling

Essential steps for both pyramids, which have to be repeated at each stage are *upsampling* and *downsampling*. Downsampling in stream processing is implemented by simply dropping sample rows according to a specific factor. To avoid aliasing, the image must be reduced in bandwidth, for example by Gaussian filtering. The principle procedure is illustrated in Figure 4.14 for a downsampling factor of two, meaning that we drop every other sample and every other row. Dropping samples for downsampling can be implemented rather easily by constraining the output assignment with a modulo function, as shown in Listing 4.17.

Listing 4.17: Constraining the output assignment with a modulo function for downsampling.

```
1  void downsample(hls::stream<in_t> &img_in,
2    hls::stream<out_t> &g_out){
3  ...
4  // ASSIGN OUTPUTS
5
6    if((row >= GROUP_DELAY) && (col >= GROUP_DELAY)){
7      if((row%2 == 1) && (col%2 == 1))
8        g_out << g_pixel;
9    }
```

Note that we must take the group delay into account for determining the output of the downsampling process, if it is combined with the Gaussian filter. For upsampling, the sampling rate is increased by a factor of two and the new samples of value 0 between successive values of the signal are interpolated. As the resulting signal's spectrum is a two-fold periodic repetition of the input spectrum and only the frequency components of the original signal are unique, the repetitions should be rejected by passing the resulting sequence through a lowpass filter. The loss in brightness can be compensated by subsequent

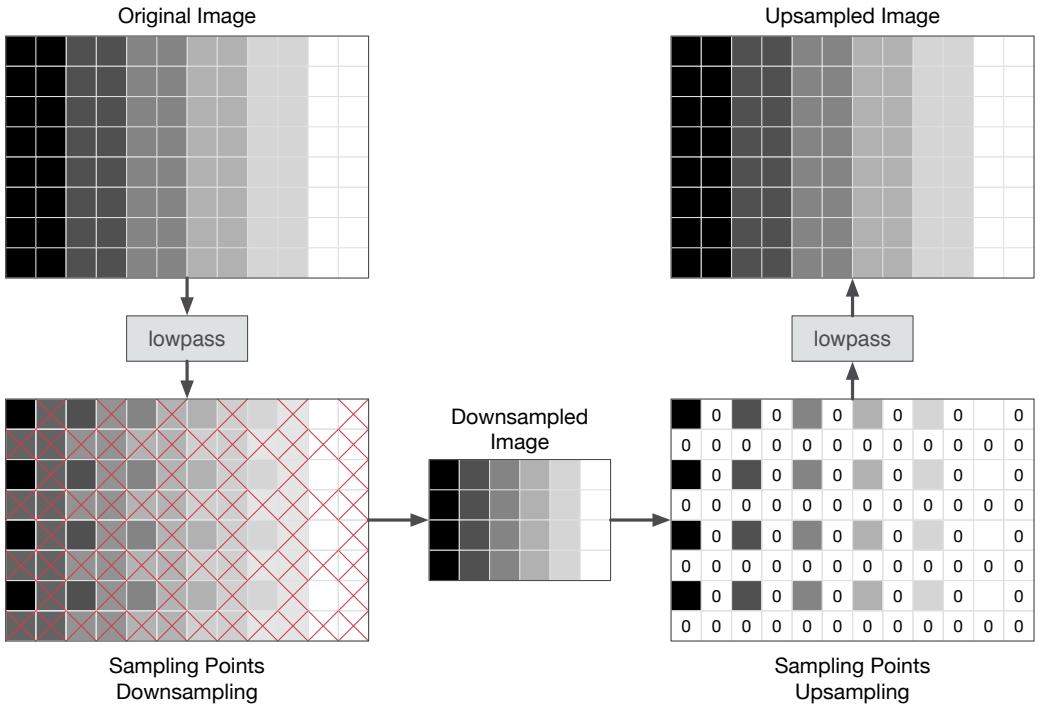


Figure 4.14.: Down- and upsampling images by a factor of two. From the original image, all the crossed pixels are dropped to obtain the downsampled image. To upsample to a higher resolution, zero values are inserted to fill the intermediate pixels.

upscaling. An alternative to using a Gaussian filter for implementation is bilinear or bicubic interpolation [TBU00]. Upsampling can be implemented analogous to downsampling, in contrast we must impose the condition on obtaining the input, as shown in Listing 4.18

Listing 4.18: Constraining the input with a modulo function for upsampling.

```
1 // OBTAIN INPUT
2   if(col < X && row < Y){
3     if(row%2 == 1 && col%2 == 1)
4       g_in >> in_pixel;
5     else
6       in_pixel = 0;
7   }
```

4.7.2. Library Integration

We have included the necessary functions for up- and downsampling as in the library. All of the functions use C++ template arguments to specify the targeted initiation interval (`II_TARGET`), the kernel size for local operators (`KERNEL_SIZE`),

the width and height of the input image (MAX_WIDTH and (MAX_HEIGHT), as well as the input and output data types (IN and OUT).

Upsampling

The library function prototype for upsampling a data stream is shown in Listing 4.19.

Listing 4.19: Library function prototype for upsampling a data stream.

```
1  template<int II_TARGET, int MAX_WIDTH, int MAX_HEIGHT, int KERNEL_SIZE,
     typename IN, typename OUT, class Filter>
2  void upsample(
3    hls::stream<IN> &data_in,
4    hls::stream<OUT> &data_out,
5    const int &width,
6    const int &height,
7    const int &factor,
8    Filter &filter,
9    const enum BorderTreatment::values borderTreatment
10 );
11
```

The input to the function includes the input and output data streams, as well as the width and the height of the image. Furthermore, the sampling factor must be provided to define how many samples between signal values will be interpolated. The local operator to perform the interpolation is provided as an instance of the class Filter. Finally, the designer can also define whether to use border treatment.

Downsampling

The library function prototype for downsampling a data stream is shown in Listing 4.20.

Listing 4.20: Library function prototype for downsampling a data stream.

```
1  template<int II_TARGET, int MAX_WIDTH, int MAX_HEIGHT, int KERNEL_SIZE,
     typename IN, typename OUT, class Filter>
2  void downsample(
3    hls::stream<IN> &data_in,
4    hls::stream<OUT> &data_out,
5    const int &width,
6    const int &height,
7    const int &factor,
8    Filter &filter,
9    const enum BorderTreatment::values borderTreatment
10 );
11
```

The input to the function includes the input and output data streams, as well as the width and the height of the image. Furthermore, the sampling factor must be provided to define how many samples between signal values will be dropped. The local operator to perform the interpolation is provided as an instance of

the class `Filter`. Finally, the designer can also define whether to use border treatment.

4.7.3. Pyramid Construction

Using the above described library functions for up- and downsampling, we can construct an N-stage Gaussian pyramid as a sequence of pipelined function calls in Vivado HLS that. A schematic representation of an N-stage Gaussian pyramid is shown in Figure 4.15

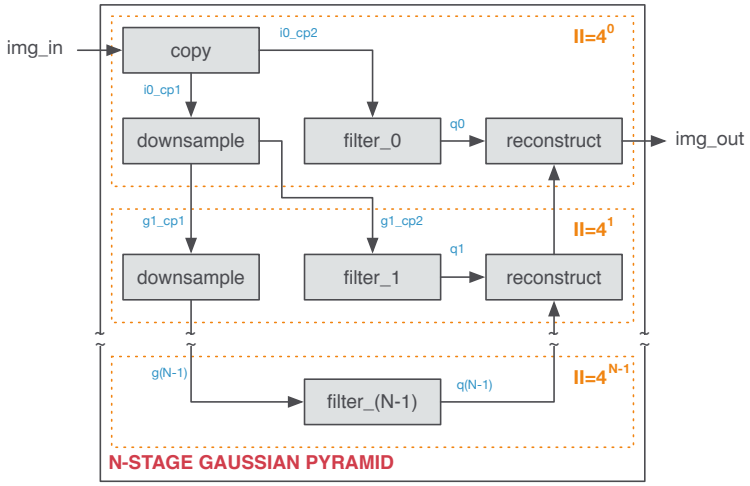
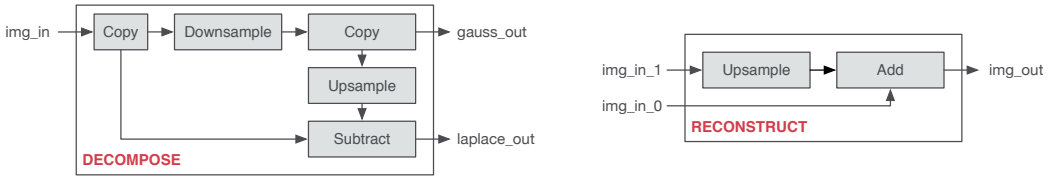


Figure 4.15.: Schematic representation of an N-stage Gaussian pyramid.

The *reconstruct* block in Figure 4.15 encapsulates upsampling the image from the lower level, by calling the `upsample` function, and combining the result with the filtered image of the current level, for example, by simply adding the intensities, as depicted in Figure 4.16(b). For the Laplacian Pyramid we must additionally devise a *decompose* module to obtain the Laplacian image, which is computed by upsampling the Gaussian image and subtracting it from the original, which is shown in Figure 4.16(a). The Laplacian pyramid can then be constructed in analogy to the Gaussian pyramid, as shown in Figure 4.17.

As the images become smaller by a factor of four at every stage, the throughput of a stage can also be reduced by a factor of four compared to its predecessor stage. An ideal method to achieve this in high-level synthesis is to adapt the pipeline rate of each stage, as it is indicated in Figures 4.15 and 4.17. This is however a loop optimization and must be specified in all loops of all functions of the stage. The major advantage of this approach is that the implementation of a stage can be reused and the functions can be optimized for each stage according to the required throughput. As the highest throughput is required for the bottom level of the pyramid, it is here where most of the hardware



(a) Decompose. (b) Reconstruct.

Figure 4.16.: Schematic representations of the *decompose* and *reconstruct* modules.

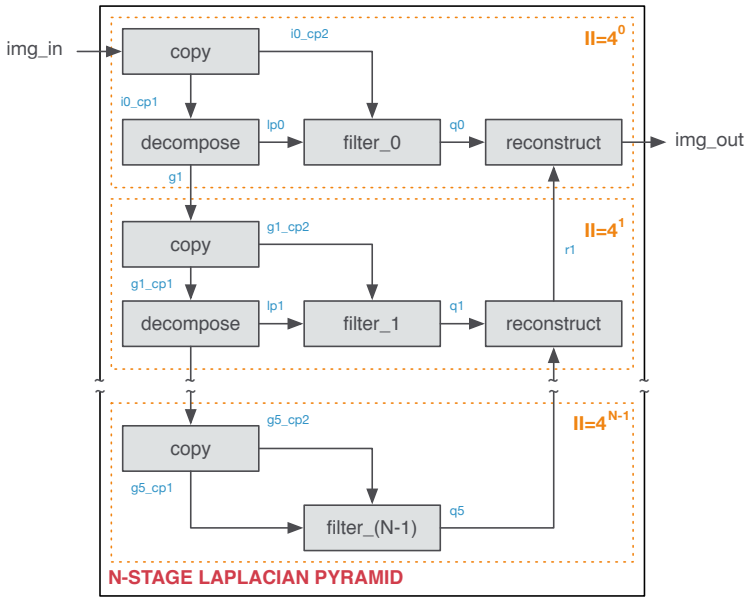


Figure 4.17.: Schematic representation of an N-stage Laplacian pyramid.

resources of the design will be needed. On higher levels, as the performance constraints are relaxed, resource requirements can be greatly reduced by sharing operators between computations and by adapting operators to the performance requirements of each stage.

The functions to implement the building blocks of an image pyramid, such as upsampling and downsampling can be synthesized by Vivado HLS as IP cores and assembled using a hand coded wrapper design in an Hardware Description Language (HDL). A more convenient implementation is to encapsulate the sequence of function calls in a top-level function for HLS. Data distribution between the functions can be handled efficiently using `hls::stream` objects as an interconnect with FIFO semantics. Listing 4.21 shows the implementation of a simple two-stage Gaussian pyramid for images of size 512×512 pixels.

Listing 4.21: Implementation of a Gaussian pyramid comprising two stages.

```
#define TOP_WIDTH    512
#define TOP_HEIGHT   512

void gauss_top(hls::stream<data_t> &img_in, hls::stream<data_t> &img_out) {
  #pragma HLS DATAFLOW

  hls::stream<float> i0_cp1;
  #pragma HLS STREAM variable=i0_cp1 depth=1
  hls::stream<float> i0_cp2;
  #pragma HLS STREAM variable=i0_cp2 depth=1
  hls::stream<float> g1;
  #pragma HLS STREAM variable=g1 depth=127
  hls::stream<float> c0;
  #pragma HLS STREAM variable=c0 depth=2126
  hls::stream<float> c1;
  #pragma HLS STREAM variable=c1 depth=1

  // STAGE 0
  //copy(input, output1, output2)
  cpy_512(img_in, i0_cp1, i0_cp2, TOP_WIDTH, TOP_HEIGHT);
  //filter(input, output)
  filter_512(i0_cp2, c0, TOP_WIDTH,TOP_HEIGHT);
  //downsample(input, output)
  downsample_512(i0_cp1, g1);

  // STAGE 1
  //filter(input, output)
  filter_256(g1, c1, TOP_WIDTH/2,TOP_HEIGHT/2);

  // RECONSTRUCT
  //reconstruct(input0, input1, output)
  reconstruct_512(c0, c1, img_out,TOP_WIDTH,TOP_HEIGHT);
}
```

The `cpy_512`-function encapsulates a simple point operator that duplicates the input stream. The `downsample_512`-function contains the function call to the `downsample`-function of the library, as well as the definition of a local operator for interpolation, for example a bilinear filter. The `reconstruct_512`-function contains a sequence of function calls, as shown in Figure 4.16(b). The `filter_512`- and `filter_256`-functions implement the image processing algorithm applied at each stage. Streams are very flexible to distribute data within a hardware module generated in Vivado HLS. As a default, however, all streams are implemented as registers with a depth of one. They can also be enlarged to be implemented as FIFO buffers using either shift-registers or BRAMs, however, Vivado HLS currently does not determine the stream depth automatically. Therefore, the stream depths are set manually in Listing 4.21 using the `depth`-option of the `STREAM` pragma.

4.7.4. Optimization of Buffer Requirements

From the structural representation, it already becomes clear that certain streams will require to hold more than one element before the receiving node will start

to consume pixels. In fact, it is vital that all interconnecting streams provide enough space for pixel buffering. Consequences of insufficient buffering might be inferior performance, data loss or even deadlock. In general, determining buffer sizes for data streaming applications is not a trivial problem, especially in the presence of irregular production and consumption rates. For moderately complex designs it is possible to calculate an approximate buffer size by evaluating when and at which rate data is produced and consumed. Another viable procedure for this is to first increase the default buffer size for all streams to a value that will most likely not be exceeded during runtime. The hardware can then be simulated using an external simulation tool in order to verify functional correctness and determine the buffer sizes.

As stream buffers are mapped to DPRAM on FPGAs, there exists a specific step size for the buffers, depending on the used data type. Thus, it is often not necessary to obtain exact buffer sizes. An approximation within the step size interval is sufficient. Unfortunately, it is not possible to determine buffer requirements during C-simulation, since functions are run sequentially and it is therefore not possible to obtain fill rates in case of concurrent access. In addition, Vivado will not infer registers to enforce a pipeline rate that is larger than the latency for a single iteration of the loop. For example, Vivado can pipeline a function with a latency of multiple clock cycles to reach an II of a single clock cycle, if there are no data dependencies preventing this. On the opposite, a function having a latency of only two clock cycles cannot be forced to operate at an II of three or more clock cycles, but will always fall back to a maximum II equal to the latency. Due to this, the design must be synthesized first to obtain the actual production rates before buffer dimensions can be approximated. Alternatively, it is possible to enforce a higher II by applying the *latency* pragma to the function, however, experimental results have shown that this will cause a higher overall latency, increase resource requirements, and might deteriorate performance.

The amount of buffer space required is also greatly influenced by delays in the pipeline. Especially on lower levels, which are scheduled with a higher II, a delay in pixel production or consumption patterns can severely affect the required buffer space on the top-most level. An optimization exists in irregular sampling positions for up- and downsampling. Dropping and sampling pixels performed at the same relative image positions during both processes will either cause a longer delay until the first pixels are produced by the downsampling module or otherwise, cause a stuttering delay during upsampling. The constellation presented in Figure 4.14, immediately starts producing pixels after downsampling. The upsampler also starts consuming pixels immediately, but the producer operates at a four times slower production rate than the upsampler would require to operate seamlessly. Figure 4.18 shows the opposite case, where the situation is improved for the upsampler by sampling pixels as late as possible. The

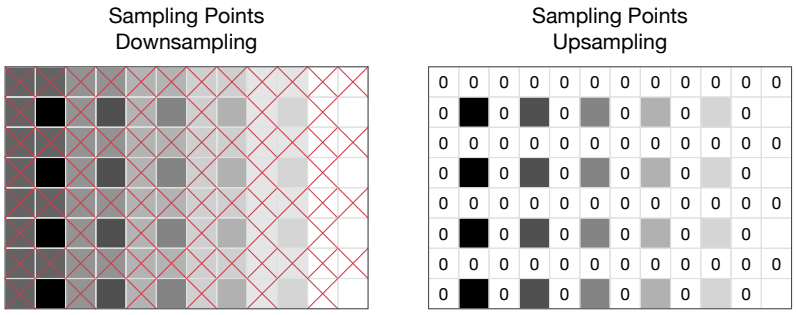


Figure 4.18.: One additional image line delay caused by downsampling.

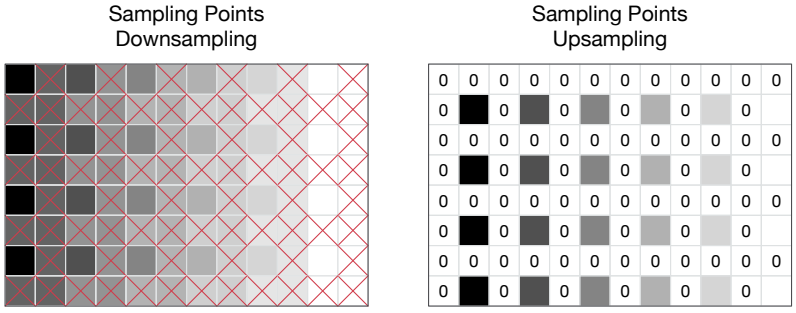


Figure 4.19.: Alternate sampling and interpolation points.

downsampler, however, uses the same relative image positions for the output, which, in addition to the delay created by the local operator, causes a delay of one image line, before the receiver can start processing. An improved constellation is presented in Figure 4.19, where the downsampler immediately starts producing pixels. The upsampler can already start producing zero elements before the actual pixels from the producer start arriving. Although this causes an irregular rate with intermediate waits, due to the head start it will average to a seamless production rate over the course of two consecutive image lines.

Although it is possible to instantiate buffers on every stream, there are actually only three cases where the interconnection requires buffering. These are (a) after downsamplers, (b) before upsamplers, and (c) before nodes that combine data streams and have different path lengths. The necessity for buffering after downsampling and before upsampling is due to different iteration intervals between the stages and the irregular production and consumption rates. The buffer requirement for combining nodes, such as at the reconstruct step, becomes evident from the structure of the accelerator. For example, the connection between filtering and reconstruction on each level requires a very large buffer, since filtering immediately starts to produce pixels, whereas reconstruction must wait until all the lower stages of the pyramid have been traversed. Constraining

buffering to these specific locations can lead to a reduced resource usage and a more regular execution pattern that simplifies buffer size approximation.

4.8. Evaluation of Pyramid Algorithms

We have conducted several experiments to evaluate the proposed library components and optimizations for pyramid construction. For the evaluation, we have used the Xilinx Kintex 7 FPGA, as described in Section 4.6.1, and hardware resource requirements for the implementations were obtained, as described in Section 4.6.2. Image sizes and data types vary for the experiments and will be indicated where appropriate.

4.8.1. Pyramid Optimizations

In Section 4.7.4, we have proposed several optimizations to improve buffer requirements. These include to not enforce pipeline rates, to restrict buffer locations, as well as to use irregular sampling points for down- and upsampling. To evaluate these optimizations, we have implemented a multiresolution algorithm from the domain of medical image processing that uses a Laplacian filter for feature controlled image denoising. The design is closely related to the general structure of Laplacian pyramids, shown in Figure 4.17.

We have first evaluated how the overall latency (LAT) is affected by using different sampling locations for down- and upsampling using a six-stage Laplacian pyramid. Sampling as early as possible is referred to as *ASAP*, as late as possible is called *ALAP*, and using different sampling points (ASAP for downsampling and ALAP for upsampling) is denoted by *MIXED*. There were no further constraints imposed on the pyramid implementation.

To obtain the measurements, we have first generated a VHDL implementation using Vivado HLS using sufficiently large streaming buffers. The next step involved simulation of the pyramid algorithm and obtaining the maximum fill state of the FIFO buffers, as well as the latency for calculating one complete image. The maximum buffer requirements were then used as size constraints for a second synthesis run using Vivado HLS and subsequent implementation using the Vivado Design Suite for PPnR resource assessment. The experimental results are listed in Table 4.7.

The results of the experiment indicate that the proposed scheme for asynchronous down- and upsampling can effectively reduce the latency introduced by these components and therefore also reduces the buffer requirements and improve the throughput.

Moreover, we have evaluated the other proposed implementation optimizations. Starting from an image resolution of size 512×512 floating-point values, we

Table 4.7.: Comparison of Laplacian pyramid implementations using different sampling points for down- and upsampling.

	II	LAT	SLICE	LUT	FF	BRAM	DSP	F [MHz]	TP [fps]
$P6$ MIXED	1	302382	15093	43237	48551	229	217	202.5	674.1
$P6$ ALAP	1	316421	14698	43135	48543	229	217	204.6	646.6
$P6$ ASAP	1	316316	14841	43033	48525	223	217	203.8	640.2

have incrementally increased the number of levels in the Laplacian pyramid until up to six stages, which will be referred to as $P2, \ldots P6$. For each pyramid, we have evaluated three different implementations, as follows.

No Constraints (NC) – The implementation does neither use constraints for buffer locations nor enforce a specific II.

Full Constraints (FC) – The implementation uses constraints for buffer locations and strictly enforces the II at each level.

Buffer Constraints (BC) – The implementation uses constraints for buffer locations, but does not enforce the II.

The experimental results of the evaluation, which were obtained using the method described above, are listed in Table 4.8.

Table 4.8.: Comparison of different buffer and scheduling optimizations for Laplacian pyramid implementations.

	II	LAT	SLICE	LUT	FF	BRAM	DSP	F [MHz]	TP [fps]
$P2$ NC	1	265403	6427	16801	19313	19	115	203.8	815.4
$P2$ FC	1	265368	6038	17248	17243	18	125	210.5	815.8
$P2$ BC	1	265401	5955	16722	19225	18	115	210.3	815.9
$P3$ NC	1	269622	9554	27431	29587	43	147	210.9	779.7
$P3$ FC	1	269684	9313	27546	27043	43	157	207.0	772.8
$P3$ BC	1	269618	9752	27215	29992	43	147	205.6	764.6
$P4$ NC	1	277947	10930	29799	34818	77	169	210.6	757.8
$P4$ FC	1	277893	10272	29903	32469	77	175	207.2	745.7
$P4$ BC	1	277939	10863	29807	35188	77	169	203.7	733.1
$P5$ NC	1	294427	13402	36506	41466	132	193	210.2	716.2
$P5$ FC	1	294631	12305	36368	39094	131	199	208.4	702.6
$P5$ BC	1	285640	12932	36215	41787	129	193	206.1	719.8
$P6$ NC	1	302382	14841	43033	48525	223	217	216.4	674.1
$P6$ FC	1	328623	15199	43622	45668	224	223	216.5	640.5
$P6$ BC	1	302382	14802	42979	48310	219	217	216.5	695.5

The results are also illustrated in Figure 4.20. As the evaluation shows, the effects of the optimizations do not become clear for small pyramids, and may even cause more resource usage and a lower throughput than an unconstrained

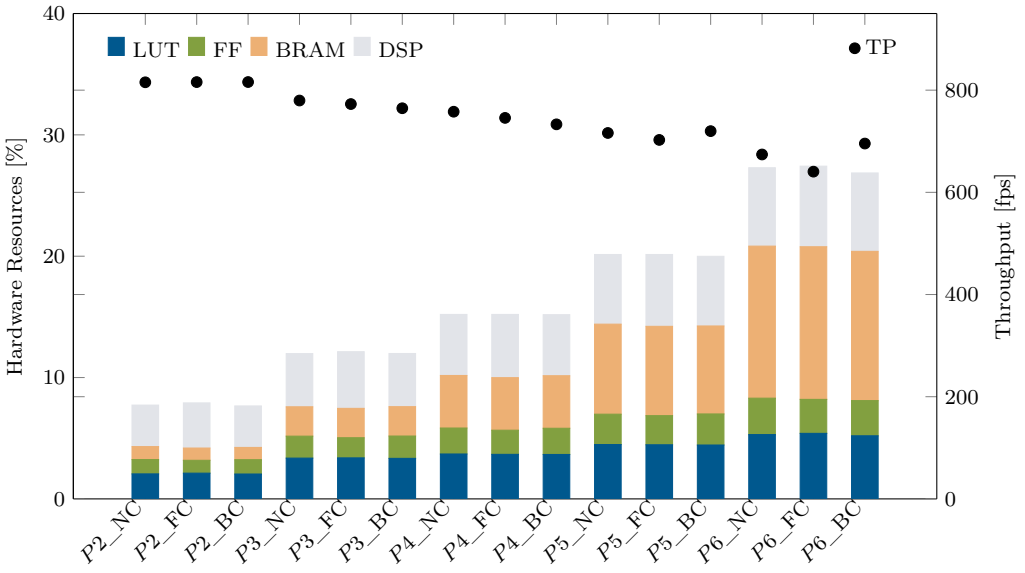


Figure 4.20.: Comparison of the amount of used FPGA resources and achievable throughput in frames per second (fps). The suffix denotes the implementation scheme, no constraints (NC), full constraints (FC), and buffer location constraints (BC).

implementation. For larger designs that incorporate a higher number of stages, it becomes clear that the combination of constraining the buffer locations and not strictly enforcing a specific II is the best combination to obtain the highest throughput and lowest memory requirement, however, at the expense of an increased logic utilization. Not imposing any of the constraints can almost always achieve a better implementation than strictly enforcing buffer locations and a specific pipeline rate.

4.9. Summary

HLS tools are currently not able to synthesize software programs written for sequential execution on a microprocessor into highly efficient hardware accelerators. In consequence, software specifications must be programmed for HLS so that (a) they implement an optimal processing paradigm, (b) make efficient use of the architecture, and (c) exploit the capabilities of the compiler. To facilitate the design of highly efficient image processing accelerators using HLS for FPGAs, we have therefore proposed a library approach in this chapter, which (a) implements a hierarchical memory infrastructure for data-streaming, (b) supports point and local operators to specify simple algorithms and complex filter pipelines, and (c) applies appropriate synthesis directives to optimize the throughput as well

as allows design space exploration. We have shown that the proposed approach delivers superior synthesis results in terms of throughput, resource usage and efficient exploitation of available resources than the OpenCV library, designed by Xilinx. We have also compared the presented algorithm implementations to two software-based accelerators, an embedded GPU and a server-grade GPU. The FPGA implementations can clearly outperform the embedded GPU (eGPU) in terms of performance and energy efficiency. However, the comparison to the sever GPU shows that FPGA is more efficient in terms of energy requirements, but it is still far behind considering throughput. If we analyze these results carefully, we can observe that the FPGA implementations already fully exploit Instruction-Level Parallelism (ILP) through pipelining and the resource requirements only cover a fraction of the logic available on the platform. As the accelerators do not exploit Data-Level Parallelism (DLP) but only process one pixel per clock cycle, the performance results can still be improved, if we exploit the high data-rate serial Input/Output (I/O) available on the FPGA and increase the level of parallelism. Appropriate techniques for such an approach are discussed in Chapter 5.

The parallelism of the FPGA architecture makes the platform also an attractive target for highly parallel filter pipelines, such as used in MRA. Therefore, we have also introduced library components for the construction of Gaussian and Laplacian image pyramids. As a contribution here, we have (a) proposed a hierarchical pipelined structure for parallel processing of the pyramid stages, (b) proposed an asymmetric scheme for down- and upsampling, and (c) tuned stream buffering for the pyramid architecture. Realizing such structures as streaming pipelines can produce a high degree of parallelism, but the cyclic structure with long delays requires a high amount of on-chip memory elements. Hence, it is important to mention that current imaging systems use image resolutions that exceed the here used 512×512 pixels by far. Experimental results have shown that the on-chip BRAMs resources of the here used Kintex 7 FPGA can accommodate a Laplacian pyramid with eight stages for image resolutions of up to 1024×1024 pixels. Imaging systems using larger image sizes will overwhelm the on-chip memory and must therefore be able to use external memory to off-load critical buffers. In Chapter 6, we will introduce an appropriate method to facilitate access also to such resources as part of an HLS design.

5 Beyond Instruction-Level Parallelism

As the experimental performance results of Chapter 4 have shown, current tools for High-Level Synthesis (HLS) excel at exploiting Instruction-Level Parallelism (ILP) and can reasonably satisfy real-time constraints in video-based imaging, that is, to achieve a throughput that is higher than the frame rate most sensors are capable to offer [DL95]. When the results are compared with software-based accelerators, such as the Nvidia Tesla Graphics Processing Unit (GPU) or the ARM Mail embedded GPU (eGPU), the FPGA implementations are much more energy-efficient than the GPUs. With respect to the accomplished throughput, the Field Programmable Gate Array (FPGA) results can only outperform the Mali, but are still far behind the Tesla. A higher performance is desirable, since it will allow larger problem sizes (larger images) or higher frame rates. Moreover, the implemented algorithm might only be a small part of an imaging system and reducing the response time allows for more processing time before and after the algorithm is computed. Therefore, this chapter examines the exploitation of Data-Level Parallelism (DLP) on FPGAs using C-based HLS of image filters and streaming pipelines, consisting of point and local operators. In addition to well known *loop tiling* techniques, we propose *loop coarsening*, which delivers superior performance and scalability. Loop tiling corresponds to splitting an image into separate regions, which are then processed in parallel by replicated accelerators. For data streaming, this also requires the generation of glue logic for the distribution of image data. Conversely, loop coarsening allows to process multiple pixels in parallel, whereby only the kernel operator is replicated within a single accelerator.

5.1. Spatial and Data-Level Parallelism

Parallelization of loop programs by exploiting *spatial parallelism* means the distribution and concurrent execution of loop iterations over different processing units, or in our case, the generation of multiple accelerators to process more

than one loop iteration in parallel. In principle, it is possible to implement any algorithm using a dedicated resource and execute all of the computations in parallel, given enough resources. An algorithm, however, will 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 and have complex data dependences, the effect of parallelization might only be marginal. This observation has first been formulated by Amdahl in [Amd67], commonly known as Amdahl's law, and is often used to estimate the theoretical maximum speedup to be expected from parallelization.

Denote s as the proportion of an algorithm that is constrained to be executed serially and let p be the proportion of the algorithm that is capable to be run fully in parallel. Furthermore, let n be number of parallel units for the execution of p. The best possible speedup S that can be obtained by parallelization of p using n resources can be obtained by comparing the time necessary to run the algorithm sequentially to the time required for parallel execution.

$$S \leq \frac{s+p}{s+p/n} \tag{5.1}$$

Equality can typically only be achieved if the parallelization of the algorithm does not cause any additional overhead, such as extra communication or stall cycles. 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* [Hwa+93].

On the operation level, we have made use of spatial parallelism in Chapter 4 by unrolling the innermost loops and balancing the expressions, thereby executing as many instructions in parallel as possible. To enable high clock frequencies, most of these instructions are implemented as multi-cycle operations in hardware. Excessive use of hardware resources can be avoided by using the concept of data streaming, which turns spatial parallelism into temporal parallelism. A common form to exploit this is by overlapping the execution of instructions for successive loop iterations. The methodology is called *pipelining* [Lam88] and is also often referred to as ILP [HP14]. Allocating dedicated accelerators for each step of the imaging algorithm and interconnecting these in the form of a streaming pipeline additionally allows the exploitation of temporal parallelism contained in the algorithm at application level [Bai11].

Current FPGA high-speed serial transceiver technology enables the communication interfaces to operate at very high data rates and deliver multiple pixels per clock cycle in an aggregated bit vector, which we refer to as a *data beat*[7].

[7]For instance, a single serial interface to a Double Data Rate Type Three (DDR3)-Synchronous Dynamic Random Access Memory (SDRAM) can deliver up to 512 bits at 200 MHz, which would, for example, be equal to a vector of 32 16-bit pixels.

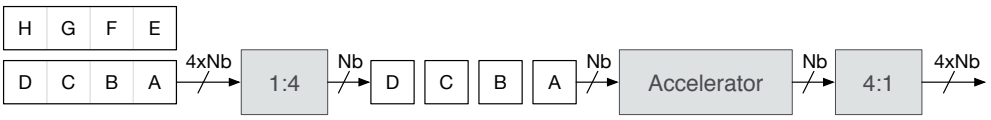


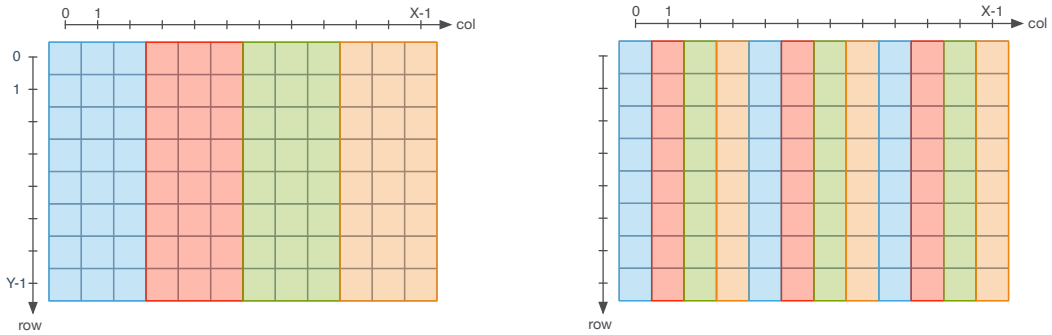
Figure 5.1.: Separation of data beats into individual pixels by external glue logic. After the processing by the accelerator, pixels are reassembled into data beats for communication.

If we examine the results of the previous chapter, we have so far separated the incoming data into individual pixels outside of the accelerator by external logic, as Figure 5.1 shows. Using such an approach, the accelerator can only achieve a fraction of the data rate provided by the communication interface. A widely recognized method to increase the throughput exploits spatial parallelism and DLP by duplicating the accelerator and letting each of the accelerators process different parts of the image concurrently. Instead of separating the pixels, the accelerator receives an appropriately sized vector. The data beats are then disassembled into individual pixels for processing internally, depending on the processing methodology. The performance gain is of course limited by the data rate of the Input/Output (I/O) protocol and the available resources on the FPGA.

DLP can be exploited by either *block-wise* or *cyclic* data distribution, as shown in Figure 5.2. The chosen data distribution method also determines the architecture of the accelerator. The block-wise distribution scheme requires the replication of the whole accelerator and demands for additional components for distributing and reassembling data. The methodology is well established and commonly known as *loop tiling*, see e. g., [Wol89]. We discuss its implementation in more detail in Section 5.2. Cyclic distribution can in contrast be used by an implementation that only replicates the accelerator kernel and processes the pixels in parallel. Applying it 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 5.3.

5.2. Loop Tiling

Loop tiling is one of the most widely applied parallelization techniques to exploit spatial parallelism at the operation level. Similar to the well-known *divide and conquer* methodology, the data set is split orthogonal to the scanline into multiple parts, which are then processed by multiple dedicated accelerators in parallel. An illustration of the concept is shown in Figure 5.3. Every accelerator receives



(a) Block-wise data distribution.

(b) Cyclic data distribution.

Figure 5.2.: Methods for exploiting data-level parallelism. Colored data segments will be assigned each to one replicated accelerator for parallel processing of the loops.

a contiguous part of the image row in a round robin fashion. After all have received their part, the distribution restarts with the next row of the image.

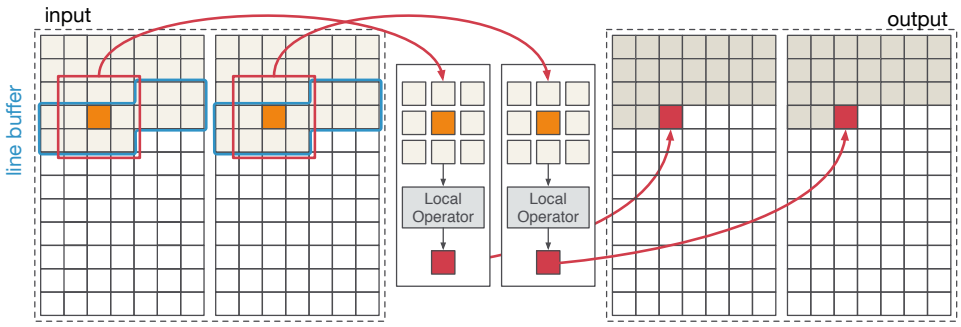
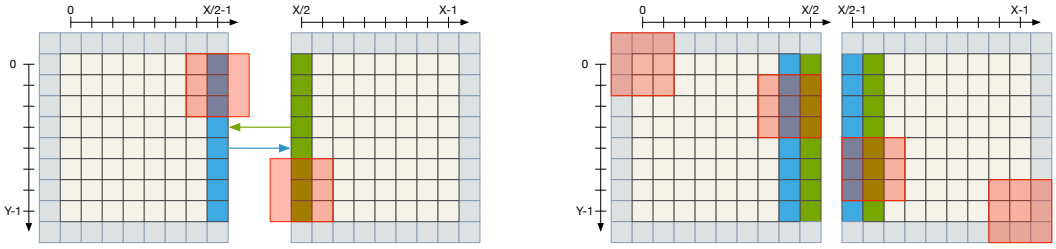


Figure 5.3.: Loop tiling for local operators by partitioning the input image into multiple parts and processing each by a dedicated accelerator concurrently.

5.2.1. Overlap Area for Local Operators

No data exchange is necessary when loop tiling is employed for the parallelization of point operators. In the case of local operators care must be taken at the borders shared among two image stripes, since the kernel windows overlap into the area outside of the image region, as shown in Figure 5.4(a). At the actual borders of an image, this problem can be solved efficiently using border treatment.

Applying the technique also for the splitting border might leave visible traces in the output image. A better solution is to define an *overlap area* around the splitting border and distribute the pixels within this area to both neighboring accelerators. This procedure is commonly known as *overlapped tiling* [BDQ98]. The size of the overlap region, which is also known as *halo* or *ghost region*, must be chosen according to the window size of the local operator. For a window size of $w \times w$ pixels, the amount of pixels that the kernel window overlaps into the neighboring region, which we refer to as the *overlap* Ol, is equal to the kernel radius $Ol = r = \lfloor w/2 \rfloor$. The overlap area OlA, however, must cover these pixels in both directions, to the left, as well as to the right of the splitting border and is therefore equal to twice the radius r of the window, $OlA = 2Ol = 2r$. An example for a window with radius $r = 1$ is shown in Figure 5.4(b). A



(a) Window overlaps into non-image region.

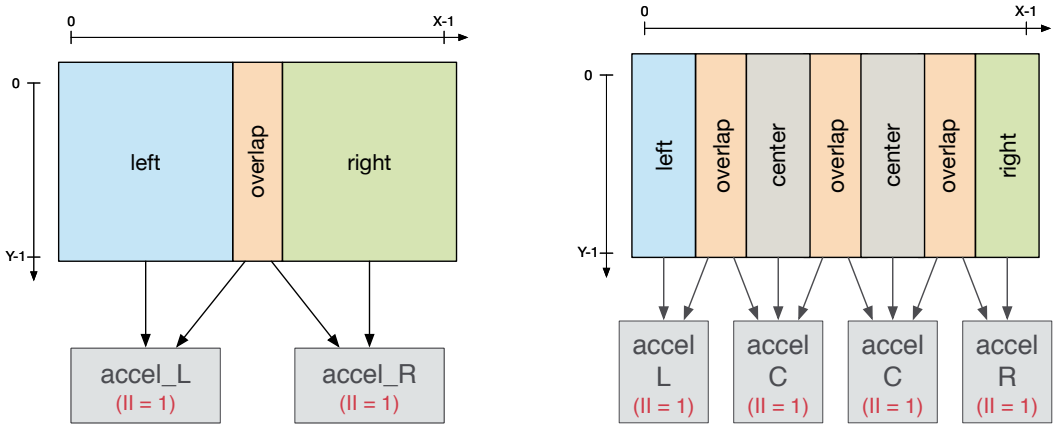
(b) Border treatment and overlap region.

Figure 5.4.: Missing image data at the actual image borders can be handled by border treatment. At the splitting borders, data from the neighboring image region is required to not leave visible traces in the output image.

typical scenario for splitting an image into two symmetric halves, which are then processed by two accelerators is shown in Figure 5.5(a). Pixels that fall into the overlap area between the neighboring image regions are distributed to both accelerators. Moreover, if the image is split into more than two regions, see Figure 5.5(b) for an example of four regions, accelerators operating on one of the central regions must be assigned the overlap area to the left, as well as the one to right of their exclusive region.

5.2.2. Implementation Structure

The highest yield can be obtained in parallel processing if the data is distributed in such a way that the involved accelerators can compute their individual image stripes without interruption. In order to avoid any stalls and excessive use of memory, the rate of incoming data should ideally match the combined processing rate of the accelerators. For the implementation, we instantiate a local buffer in



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

Figure 5.5.: Loop unrolling for local operators must make use of overlap areas between the image regions. The overlap areas are distributed to neighboring accelerators.

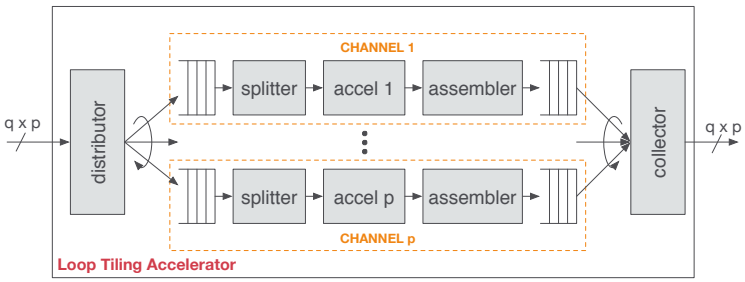
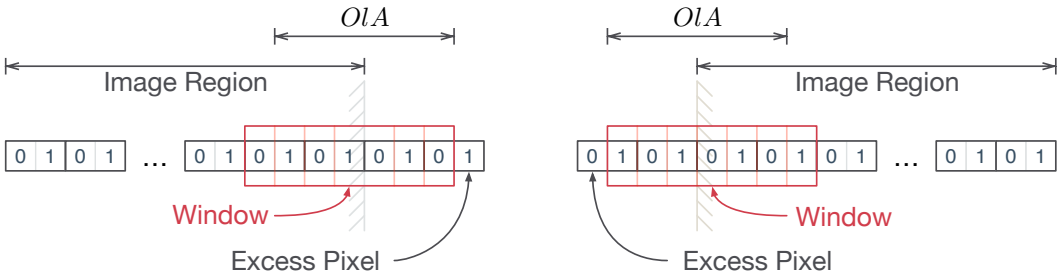


Figure 5.6.: General structure for parallel processing of an image using loop unrolling with p accelerators.

front of every accelerator that can be filled with the higher I/O data rate, but is emptied only with the lower processing rate of the accelerators. To achieve this, we combine several pixels into data beats, where the amount of pixels per data beat is defined as the *vector length* VL. If we assume p accelerators to concurrently process data with a rate of q pixels per clock cycle, a reasonable choice is to set $VL = q \cdot p$. An architecture to achieve this using only HLS is shown in Figure 5.6. The first stage of the implementation is a distributor that assigns incoming data beats to the buffer queues. The distributor must also assign data from the overlap area correctly. From the buffer queues, data is consumed by splitters, the second stage, which extract the pixels from the data beats and forward them to the accelerators. Often, only a subset of the pixels aggregated in a data beat at the beginning or the end of the overlap area belong



(a) Omitting excess pixel to the right.

(b) Omitting excess pixels to the left.

Figure 5.7.: Example of excess pixels in the data beats of the overlap region. Here, $VL = 2$, $r = 3$, and thus, $Ol = 3$ and $OlA = 6$.

to the actual iteration space of the accelerator. However, the accelerators should only receive pixels that belong to their iteration space. If the data beats supplied as part of the overlap area contain excess pixels, that is $Ol \mod VL \neq 0$, the splitter must extract only those pixels that contribute to the computation and omit excess pixels. An example is provided in Figure 5.7. If there is an overlap to the right of the image region, the splitters must skip the last $Ol \mod VL$ pixels from the last data beat belonging to the overlap, as shown in Figure 5.7(a). Likewise, the first $VL - (Ol \mod VL)$ pixels from the first data beat representing the overlap to its left must be skipped, which is illustrated in Figure 5.7(b) The third stage are the actual processing elements, also called accelerators. We distinguish between accelerators that process the far left, the far right, and the center regions of the image. The reason for this distinction are varying iteration spaces and differences with respect to where to read pixels and where to apply border treatment. Figure 5.8, shows an image divided into four symmetric regions including pixels from border treatment and pixels from the overlap areas. Processing the far left region requires border treatment for the left, upper and lower border, whereas pixels on the right border must be read in explicitly. The opposite is the case for processing the far right region of the image. For the regions in the center, pixels on the left and on the right border must be read in, whereas the upper and lower border are handled using border treatment. After processing, the output of the accelerators is reassembled into data beats by so-called assemblers at the fourth stage and gathered by a collector in the same order as supplied by the distributor at the final stage.

As shown in Figure 5.6, the architecture requires two First In, First Out (FIFO) buffers per accelerator, one before the splitter $buff_i$ and one before the assembler, $buff_o$. In HLS these FIFOs are implemented as *streams*, for which we must provide the size before synthesis. As an example, consider the round robin distribution scheme, as depicted in Figure 5.9. If we assume $p = 4$ symmetric

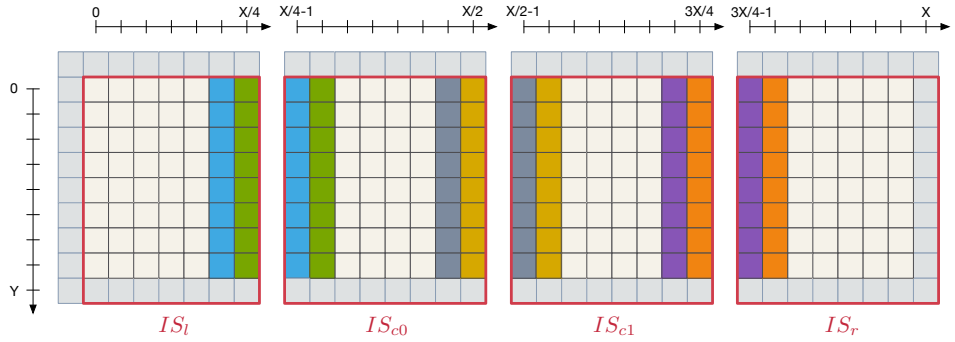


Figure 5.8.: Splitting an image into four symmetric parts. For processing a local operator, the Iteration Spaces IS for the accelerators must take the overlap area, as well as their relative position (left, right, or center) within the image into account. After the index shift due to the local operator, we obtain IS_l, IS_{c0}, IS_{c1}, and IS_r

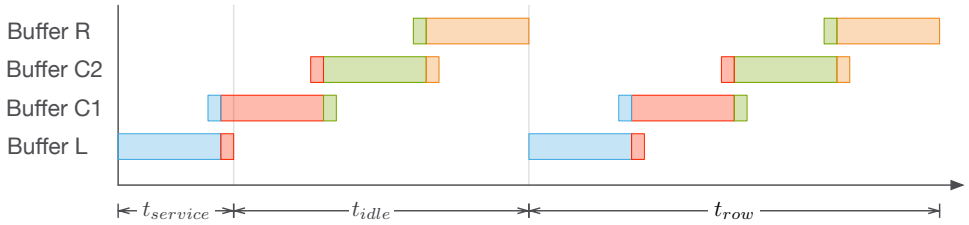


Figure 5.9.: Illustration of the division of the number of clock cycles for distributing one image line to the buffers, t_{row}, into the service cycle, $t_{service}$, where the buffer receives data, and the idle cycle t_{idle}, in which the buffer is only read for the left buffer (Buffer L).

regions in an image of size $x \times y$ pixels, where $VL = p$ and $x \mod p = 0$, we must use $p - 1$ overlap areas in between neighboring regions. The number of clock cycles necessary to distribute one image row to the individual buffer queues, t_{row}, can be obtained from the amount of pixels in one row, x, and the vector length VL as $t_{row} = x/VL$. For the far left and the far right accelerators, the distributor assigns $(x + Ol)/VL$ data beats, and $(x + 2Ol)/VL$ data beats to the central accelerators. The necessary buffer size can then be computed using the rate for data input, $rate_i$, data output $rate_o$, and length of the service cycle as the amount of data beats inserted into the buffer, subtracted by the number of data beats removed, $size_{buff} = t_{service} * rate_i - t_{service} * rate_o$. Figure 5.10 shows the progression of the fill count, that is the number of data beats currently stored in the FIFO, over the course of t_{row} for $VL = p = 4$.

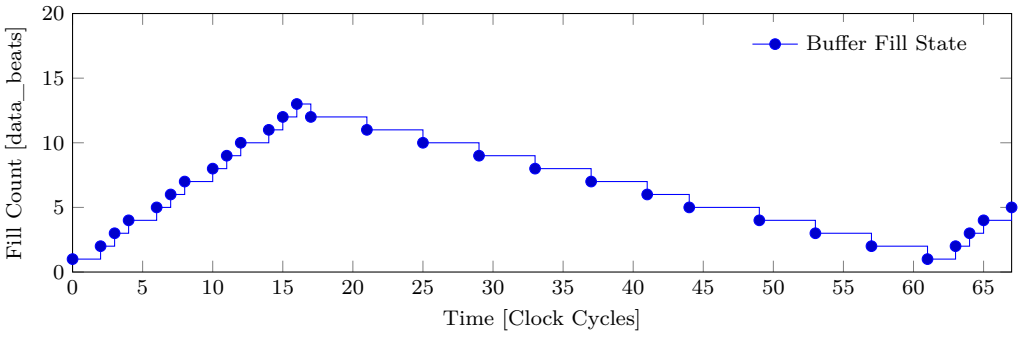


Figure 5.10.: Progression of the buffer fill count over time for $VL = p = 4$.

5.2.3. Asymmetric Image Regions

In addition to modules that implement the desired functionality, FPGA designs must often also incorporate other modules, for example, for data communication or external memory. As a consequence, the designer of an image filter for an FPGA might not have all of the hardware resources at disposal, but only a certain *resource budget*. If the current filter implementation only takes up less than 50% of the resources available in the budget, the accelerator can be simply duplicated to increase the processing speed. However, the approach cannot be applied if the current implementation already takes up more than half of the budget. In case of insufficient resources, loop tiling nonetheless offers a possibility to increase the processing speed. For pipelined designs running at an Initiation Interval (II) of one clock cycle, every operation inside the loop body requires its own hardware resource. For instance, every multiplication requires its own multiplier. Using a higher II is an effective method to allow for *operator sharing* and may decrease the amount of required resources. However, sharing a resource requires an additional multiplexer in the datapath, which might increase the delay of the critical path and thus decrease the processing speed of the accelerator. In addition to partitioning the input image into symmetric regions, loop tiling can also be used to split the image into a larger and a smaller region, where the larger region is processed with a high processing rate and the smaller region is handled by an accelerator with slower speed that requires less hardware resources. Figure 5.11 shows an example for this method, where the region on the left contains two thirds of an image and the right region only contains one third. As the accelerator on the right only needs to process half of the amount of data as the accelerator on the left, we can use an II of two clock cycles for the right accelerator. Unfortunately, there is no linear relationship between increasing the II and reducing the required resources. A prime example is Gaussian convolution, where allowing operator sharing can reduce the amount of required multipliers and adders, but the computations will always need at least one divider.

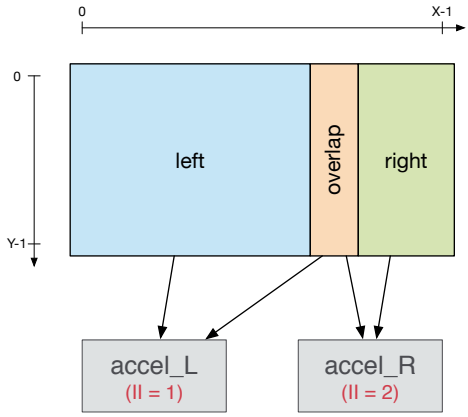


Figure 5.11.: Asymmetric splitting of image data. The accelerator on the right only has to process half of the amount of data than the accelerator on the left and can therefore use a higher II.

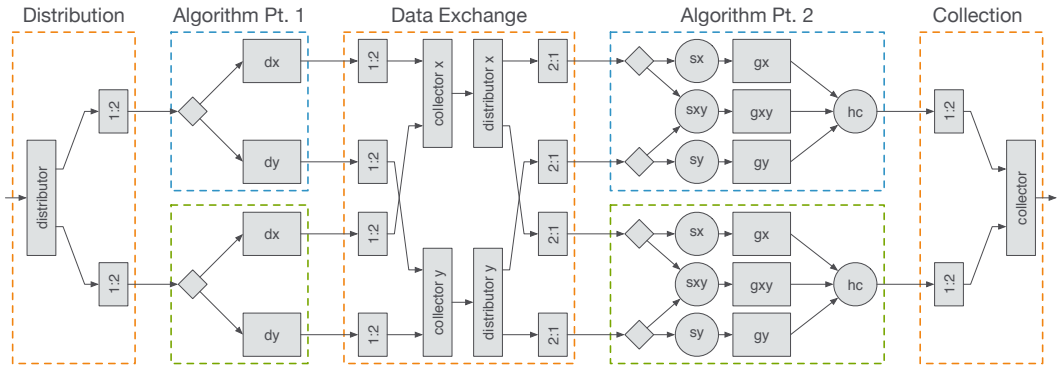


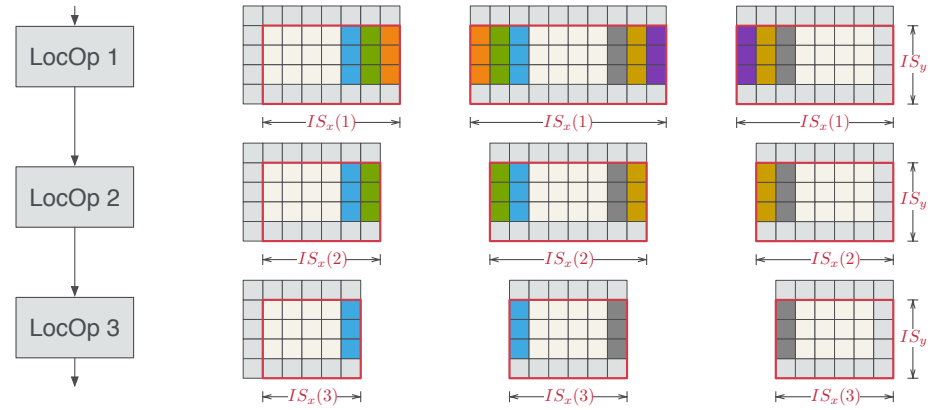
Figure 5.12.: Loop tiling to parallelize the implementation of the Harris corner detector, using explicit data exchange.

5.2.4. Loop Tiling for Streaming Pipelines

Special considerations must be made for applying loop tiling to streaming pipelines consisting of multiple local operators due to the overlap between neighboring image regions. The Harris corner detector, depicted in Figure 4.9 in Section 4.5.4, is an example of such an algorithm. When applying loop tiling to parallelize its implementation, a certain part of the image regions computed by the first local operator becomes the new overlap for the next local operator. One way to take care of this is an explicit data exchange. Figure 5.12 shows an example of the proposed architecture for streaming pipelines implementing the Harris corner detector. The implementation consists of two operator pipelines for

algorithm computation, as well as components for data distribution, collection and exchange. Although the point operators immediately following the first local operator in the pipeline do not require the data exchange, we nonetheless perform the exchange right after the local operators to minimize the amount of logic for the exchange. The problem of finding the optimal place to insert the data exchange between two local operators in a pipeline can be efficiently solved by a minimum cut algorithm.

The explicit data exchange might become costly due to the additional FIFO buffers and logic resources. An alternative approach for such pipelines is to widen the overlap area, so that the preceding local operators in the pipeline already compute data for the following operators. This allows us to omit the data exchange at the expense of redundant computations and an increased iteration space. Figure 5.13(a) shows an example pipeline of three subsequent



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

Figure 5.13.: Example of widening the overlap of iteration spaces to avoid any explicit data exchange.

local operators. The increase in the iteration space for the left, center, and right accelerators is shown in Figures 5.13(b), 5.13(c), and 5.13(d), respectively.

Suppose a pipeline consists of m local operators, each having an overlap $Ol(i)|1 \leq i \leq m$. As Figure 5.13 shows, operators implemented in the left, center, and right channels require different iteration spaces, where $IS = IS_y \times IS_x$. For example, the iteration space in x-direction of the ith local operator in the sequence $IS_x(i)$ in the left and right channels can be computed by adding the overlap of the local operator $Ol(i)$ and all subsequent operators. This correspond to increasing the overlap area for omitting the data exchange to the amount of assigned image pixels, $IS_x(i) = x/VL + \sum_i^m Ol(i)$. For the center channel, the additional overlaps must be applied for the left and the right border, yielding

$IS_x(i) = x/VL + 2 \cdot \sum_i^m Ol(i)$. In contrast, the y-direction of the iteration space, I_y, is not affected.

5.2.5. Library Integration

We have implemented the necessary functions for loop tiling as part of the library, presented in Chapter 4. All of the functions use C++ template arguments to specify

- the targeted initiation interval (`II_TARGET`),
- the tiling factor (`TILE`),
- the kernel size for local operators (`KERNEL_SIZE`),
- the width and height of the input image (`MAX_WIDTH` and `MAX_HEIGHT`), as well as
- the input and output data types (`IN` and `OUT`).

The size of the overlap and the amount of excess pixels depend on the tiling factor (`TILE`) as well as the kernel size (`KERNEL_SIZE`) and are calculated by the preprocessor according to the template arguments, as shown in Listing 5.1.

Listing 5.1: Preprocessor calculations to determine the size of the overlap region and the amount of excess pixels.

```
1  // kernel radius
2  #define RADIUS (KERNEL_SIZE/2)
3  // group delay
4  #define GROUP_DELAY RADIUS
5  #define OVERLAP ((RADIUS>TILE) ? ((RADIUS%TILE)?RADIUS/TILE+1:RADIUS/TILE):
     RADIUS)
6  #define EXCESS_LEFT ((RADIUS%TILE) ? (TILE-RADIUS%TILE) : 0)
7  #define EXCESS_RIGHT ((RADIUS%TILE) ? (RADIUS%TILE) : TILE)
```

Distribution

The library function prototype for distribution of the input data beats to streams is shown in Listing 5.2.

Listing 5.2: Library function prototype for distribution of input data to streams.

```
1  template<int II_TARGET, int TILE, int MAX_WIDTH, int MAX_HEIGHT, int
     KERNEL_SIZE, typename IN, typename OUT>
2  void distributeN(
3      hls::stream<IN> &data_in,
4      hls::stream<OUT> &data_out0,
5      ...
6      hls::stream<OUT> &data_outN,
7      const int &width,
8      const int &height);
9
```

The function takes the input data stream, several output data streams, as well as the width and the height of the image as input.

Splitting

The library function prototype for splitting data beats into individual pixels at the far left image region (`split_L`) is shown in Listing 5.3.

Listing 5.3: Library function prototype for splitting a data beat into individual pixels.

```
1  template<int II_TARGET, int TILE, int MAX_WIDTH, int MAX_HEIGHT, int
     KERNEL_SIZE, typename IN, typename OUT>
2  void split_L(
3    hls::stream<IN> &data_in,
4    hls::stream<OUT> &data_out,
5    const int &width,
6    const int &height);
```

Furthermore, the library contains appropriate splitting functions for image regions in the center (`split_C`), as well as at the far right (`split_R`). The function takes the input and output data streams, as well as the width and the height of the image as input.

Processing

The library function prototype for processing the image region far left (`process_L`) is shown in Listing 5.4.

Listing 5.4: Library function prototype for processing the far left image region.

```
1  template<int II_TARGET, int MAX_WIDTH, int MAX_HEIGHT, int KERNEL_SIZE,
     typename IN, typename OUT, class Filter>
2  void processL(
3     hls::stream<IN> &in_s,
4     hls::stream<OUT> &out_s,
5     const int &width,
6     const int &height,
7     Filter &filter,
8     const enum BorderPadding::values borderPadding);
```

In addition to the input and output data streams, processing also requires the operator as function argument. The library also contains appropriate functions for processing the center regions (`process_C`) and the far right image stripe (`process_R`).

Assembling

The library function prototype for assembling pixels into data beats is shown in Listing 5.5.

Listing 5.5: Library function prototype for assembling pixels into data beats.

```
1  template<int II_TARGET, int TILE, int MAX_WIDTH, int MAX_HEIGHT, int
     KERNEL_SIZE, typename IN, typename OUT>
2  void assemble(
3     hls::stream<IN> &data_in,
4     hls::stream<OUT> &data_out,
5     const int &width,
6     const int &height);
```

Collection

The library function prototype for collecting data beats to assemble the output image is shown in Listing 5.6.

Listing 5.6: Library function prototype for collecting the data beats from the internal data streams.

```
1  template<int II_TARGET, int MAX_WIDTH, int MAX_HEIGHT, typename IN, typename
     OUT>
2  void collect2(
3     hls::stream<IN> &in0_s,
4     hls::stream<IN> &in1_s,
5     hls::stream<OUT> &out_s,
6     const int &width,
7     const int &height);
```

Support for Floating Point Arithmetic

The library uses the arbitrary length integer class in Vivado HLS to pack individual pixels into data beats. As Vivado does not support the `reinterpret` cast of the C++ language, a simple cast from integer to floating causes the synthesis tool to instantiate an Intellectual Property (IP) core for the conversion. Instead, special conversion functions must be used to extract pixels that use floating-point as data type. For extraction we use the `i2f`-function, shown in Listing 5.7.

Listing 5.7: Library function for extracting a floating-point value from a data beat.

```
1  float i2f(int arg)
2  {
3    union {
4      float f;
5      int i;
6    } single_cast;
7    single_cast.i = arg;
8    return single_cast.f;
9  }
```

Conversely, inserting a floating-point pixel into a data beat requires the `f2i`-function, shown in Listing 5.8.

Listing 5.8: Library function for inserting a floating-point pixel into a data beat.

```
int f2i(float arg)
{
  union {
    float f;
    int i;
  } single_cast;
  single_cast.f = arg;
  return single_cast.i;
}
```

Example

The library functions can be used to assemble the top level implementation for the Laplacian operator using loop tiling where the input image is partitioned into four tiles, as shown in Listing 5.9.

Listing 5.9: Tiling the Laplacian operator implementation with a tiling factor of 4.

```
void lp3hv(hls::stream<t_major> &data_in, hls::stream<t_major> &data_out)
{
  // distribute input to streams
  distribute4<1,4,256,1024,3>(
    data_in, fifo0_in, fifo1_in,
    fifo2_in, fifo3_in, 64, 1024);

  //split data beats into pixels
  split_L<1,4,256,1024,3>(fifo0_in, filter0_in, 256, 1024);
  split_C<1,4,256,1024,3>(fifo1_in, filter1_in, 256, 1024);
  split_C<1,4,256,1024,3>(fifo2_in, filter2_in, 256, 1024);
  split_R<1,4,256,1024,3>(fifo3_in, filter3_in, 256, 1024);

  //process the laplacian operator
  ccLaplaceFilter_L(filter0_out, filter0_in, 256, 1024);
  ccLaplaceFilter_C(filter1_out, filter1_in, 256, 1024);
  ccLaplaceFilter_C(filter2_out, filter2_in, 256, 1024);
  ccLaplaceFilter_R(filter3_out, filter3_in, 256, 1024);

  //assemble processed pixels to data beats
  assemble<1,4,256,1024,3>(filter0_out, fifo0_out, 256, 1024);
  assemble<1,4,256,1024,3>(filter1_out, fifo1_out, 256, 1024);
  assemble<1,4,256,1024,3>(filter2_out, fifo2_out, 256, 1024);
  assemble<1,4,256,1024,3>(filter3_out, fifo3_out, 256, 1024);

  //collect data for output
  collect4<1,256,1024>(fifo0_out, fifo1_out, fifo2_out, fifo3_out,
                       data_out, 256, 1024);
}
```

5.2.6. Evaluation

We have evaluated overlapped loop tiling using the Xilinx Vivado Design Suite in version 2014.2 for implementation on a Xilinx Kintex 7 XC7K325TFFG900 FPGA. The algorithm specifications in C++ were synthesized into VHDL on the

Register Transfer (RT) level using Xilinx Vivado HLS with high effort settings for scheduling and binding. Post Place and Route (PPnR) hardware resource and performance characteristics were obtained by implementing the VHDL-code as an Out-Of-Context (OOC) module using the Vivado Design Suite. The provided measurements are explained in Section 4.6.2.

Symmetric Accelerators

To asses the performance of processing symmetric image regions in parallel, we have created three different implementations of the Laplacian operator, LP3HV, LP3D, and LP5 (refer to Section 4.6.2 for implementation details). The input images in Red Green Blue Alpha (RGBA) format have a resolution of 1024×1024 pixels. We have parallelized the algorithm implementations using loop tiling to partition the input image into P=2, 4, 8, and 16 symmetric image regions which are processed by as many parallel accelerators (P). Table 5.1 lists the estimated FPGA resource requirements reported by Vivado HLS after synthesis. We have distinguished the results into *functional* resources, which are required for the individual accelerators, as well as resources for data distribution, which are noted as *overhead*. To assess the *increase* in resource demand by accelerator replication, we also list the resource estimates for the non-parallelized version (P=1).

The information from Table 5.1 is also illustrated in Figure 5.14. Since the overhead is the same for all three versions of the Laplacian operator, the resource estimates of the overhead are depicted on the far right.

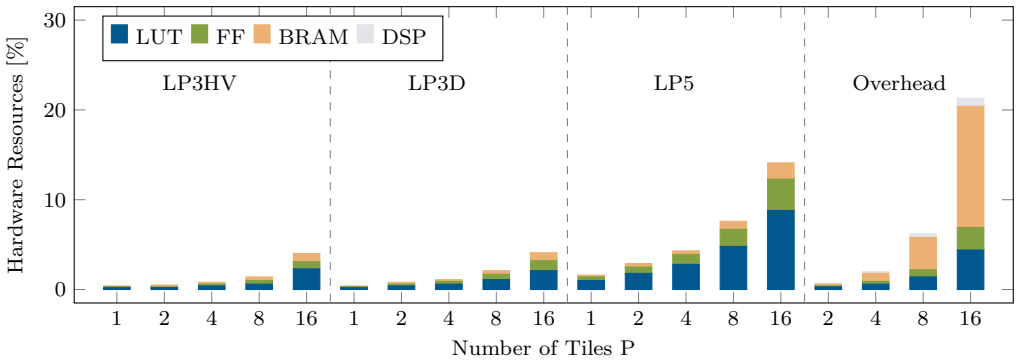


Figure 5.14.: Estimated hardware resources for parallelization of different versions of the Laplacian operator using loop tiling.

A first observation is that the accelerators, although simply replicated without any optimizations, do not cause a linear increase in the required resources. One reason for this is that the Laplacian operator has only a very simplistic loop

Table 5.1.: Comparison between synthesis estimates for *functional* resources and the *overhead* caused by loop tiling, for the Laplacian operator.

LP3HV										
	Functional			Increase			Overhead			
P	LUT	FF	BRAM	LUT	FF	BRAM	LUT	FF	BRAM	DSP
1	1699	1221	2	1.0	1.0	1.0	-	-	-	-
2	1849	1767	4	1.1	1.4	2.0	2178	1305	4	2
4	2925	3343	4	1.7	2.7	2.0	5094	4116	16	6
8	5063	6271	8	3.0	5.1	4.0	11467	12473	64	14
16	19133	12687	16	11.3	10.4	8.0	35477	41270	240	30
LP3D										
	Functional			Increase			Overhead			
P	LUT	FF	BRAM	LUT	FF	BRAM	LUT	FF	BRAM	DSP
1	1479	1377	2	1.0	1.0	1.0	-	-	-	-
2	3294	2872	4	2.2	2.1	2.0	2178	1305	4	2
4	5298	4998	4	3.6	3.6	2.0	5094	4116	16	6
8	9288	9222	8	6.3	6.7	4.0	11467	12473	64	14
16	17074	17614	16	11.5	12.8	8.0	35477	41270	240	30
LP5										
	Functional			Increase			Overhead			
P	LUT	FF	BRAM	LUT	FF	BRAM	LUT	FF	BRAM	DSP
1	8190	6122	4	1.0	1.0	1.0	-	-	-	-
2	14591	12148	8	1.8	2.0	2.0	2178	1305	4	2
4	22757	18576	8	2.8	3.0	2.0	5094	4116	16	6
8	39067	31394	16	4.8	5.1	4.0	11467	12473	64	14
16	71476	56952	32	8.7	9.3	8.0	35477	41270	240	30

kernel, so that a major part of its required resources is due to counters for loops and memory access. With an increasing number of replicated accelerators, the image stripes become narrower, which decreases the size of the counters and the logic to compare them. The same effect can be observed for the memory required for the internal line buffers. Another observation is that the overhead caused by the modules for data distribution is substantial and even exceeds the hardware resources required by the smaller operator versions. A large part of this overhead is due to the high amount of FIFO buffers, which do not only consist of Block Random Access Memory (BRAM) but also require logic for control.

Furthermore, we have obtained the PPnR resource and performance characteristics, which are listed in Table 5.2.

By comparing the overall latency of the non-parallel accelerator to the latencies of the parallel implementations with varying number P of replicated accelerators, we have calculated the *theoretical speedup* (ThSU). As the vector length VL of the data beats is equal to the number of replicated accelerators P, processing algorithms with a small window causes a growing number of stall cycles when increasing the length of the data beats. For example, the small Laplacian operators use a window size of 3×3 pixels. When we pack 16 pixels into a

Table 5.2.: PPnR resource requirements and performance results for loop tiling implementations of the Laplacian operator.

LP3HV										
P	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
2	528397	1023	2275	3020	4	2	302.1	571.8	1.6	2.0
4	272393	2193	5424	7520	20	6	279.1	1024.6	2.9	3.9
8	149513	6036	14755	21199	72	14	253.0	1692.4	4.8	7.0
16	100361	25170	46106	66695	256	30	169.2	1686.3	4.7	10.5
LP3D										
P	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
2	528389	1295	2973	4136	4	2	304.2	575.8	1.6	2.0
4	270393	2791	6622	9438	20	6	275.0	1016.9	2.9	3.9
8	149513	7170	17119	25091	76	14	263.6	1762.9	5.0	7.0
16	100361	27049	50080	73007	256	30	185.2	1844.9	5.2	10.5
LP5										
P	LAT	SLICE	LUT	FF	BRAM	DSP	F [MHz]	TP [fps]	SU	ThSU
1	1052691	993	2333	2015	2	0	354.4	336.6	1.0	1.0
2	529434	3291	7928	12858	12	2	290.7	549.1	1.6	2.0
4	270341	6077	16281	22711	24	6	269.4	996.5	2.9	3.9
8	149513	12672	33540	46562	80	14	258.9	1731.4	5.1	7.0
16	100361	34003	81727	112648	272	30	164.0	1633.7	4.9	10.5

single data beat, the data splitters for the far left and the far right channels must skip 15 pixels that are part of the overlap data beat, but do not belong to the image data for computation. At the center channels, this number is doubled. Hence, we can observe the rather large difference between the number of replicated accelerators and the theoretical speedup, once the ratio between the number of pixels for the overlap and those packed into one data beat becomes too high. One solution to this problem would be a dynamic trip count for the splitter implementation, however, this is currently not supported for pipelining by Vivado HLS. Another solution would be to use independent functions for extracting pixels from the overlap data beat and extracting pixels from regular data beats. As stepping into a function as well as leaving it causes several stall cycles, each, this is only viable if the amount of pixels to omit is very high. Using the maximum clock frequency (F) from the PPnR timing report, we can calculate the throughput (TP) for each evaluated implementation, which is also depicted in Figure 5.15, and determine the actual speedup (SU). As the clock frequency degrades with an increasing number of parallel accelerators, the actual speedup stays far behind the theoretical speed up. In fact, for the highest number of replicated accelerators, the speed up even decreases, compared to the next lower evaluated implementation, in two cases. If we consider that the performance

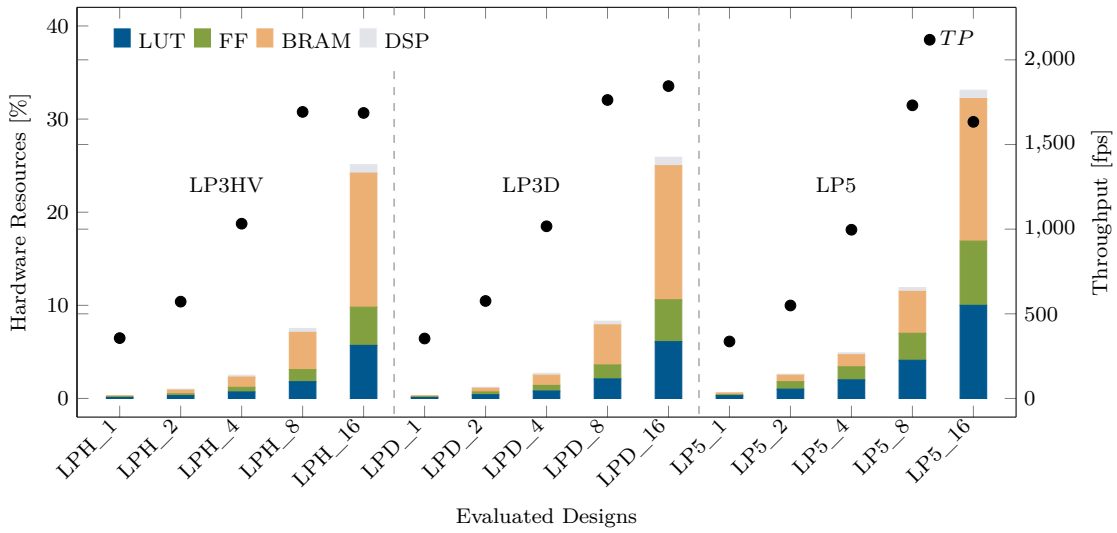


Figure 5.15.: Visualization of the PPnR FPGA resource requirements and performance measurements of the loop tiling implementations of the Laplacian operators.

gain is small and may even increase between using eight and sixteen parallel accelerators, the substantial resource overhead cannot be justified.

Pipelines

We have described two different implementation approaches for applying loop tiling to streaming pipelines in Section 5.2.4. The first approach, hereafter referred to as method *A*, performs an explicit data exchange between two successive local operators in a streaming pipeline, which requires collecting and redistributing the image regions. The following second approach, referred to as method *B*, uses redundant computations to avoid the exchange. We have evaluated the two approaches using an implementation of the Harris corner (HC) detector, which, as described in Section 4.6.2, contains two local operators. The input image of size 1024×1024 pixels, using 8-bit unsigned integer as data type, is split into two symmetric regions and processed by two accelerators in parallel. Table 5.3 lists the PPnR results for both implementations, as well as a non-prarallel version for comparison. Both methods can achieve near linear theoretical and actual speedup. In fact, due to the higher clock period of 2_B, the actual speedup is higher than the theoretical. Moreover, widening the overlap requires less resources than the explicit data exchange.

Table 5.3.: PPnR results for loop tiling implementations of the Harris corner detector.

P	LAT	SLICE	LUT	FF	BRAM	DSP	F [MHz]	T [fps]	SU	ThSU
1	1050668	5059	13232	16066	5	63	221.4	210.75	1.0	1.0
2_A	528389	10516	26525	32629	16	124	218.0	412.6	1.9	1.9
2_B	528388	9432	25783	30904	12	112	229.0	433.5	2.0	1.9

Asymmetric Accelerators

Splitting input data in an asymmetric ratio is intended for cases, where the throughput must be increased, but a full replication is not possible. This could for example be due to resource constraints. The idea behind asymmetric splitting is that the accelerator intended for the smaller part can use a higher II, which decreases the required resources.

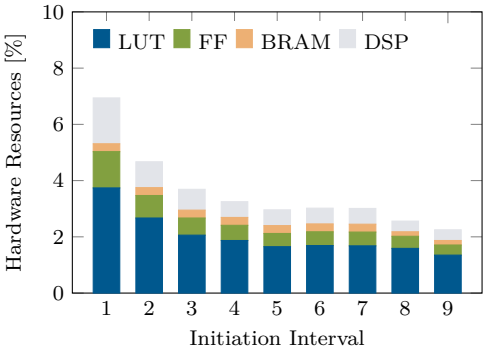


Figure 5.16.: Synthesis estimates for the Harris corner detector at varying II.

II	COL	LUT	FF	BRAM	DSP
1	512	30498	21113	5	55
2	340	21778	13131	5	31
3	256	16825	10051	5	25
4	204	15276	8915	5	19
5	174	13463	7803	5	19
6	146	13800	8024	5	19
7	128	13751	8024	5	19
8	114	12986	7020	3	13
9	102	11061	5939	3	13

Table 5.4.: Synthesis estimates for the Harris corner detector at varying II.

We have evaluated the approach using the above presented implementations of the Harris corner detector, which uses two accelerators for concurrent processing of the left and right image regions. Starting from an II of one clock cycle and symmetric image regions, we keep the pipeline rate of the left accelerator static while the II of the right accelerator is gradually increased. The image is split in inverse proportion so that the left accelerator receives the larger part for processing, whereas the right accelerator receives the smaller part. We have mentioned before that increasing the II allows a reduction in resources, however, as the synthesis estimates reported by Vivado HLS in Table 5.4 and Figure 5.16 show[8], the reduction is not linear.

[8]To apply this technique for asymmetric loop tiling, the width of the image partition also becomes more narrow. To reflect this, we also reduce the amount of image columns (COL)

We have furthermore evaluated a loop tiling implementation of the Harris corner detector using explicit data exchange (method *A*). Table 5.5 shows PPnR resource and performance results for splitting ratios R, ranging from 1:1, that is, symmetric splitting, to 9:1, that is, the accelerator on the left is scheduled using an II=1 and receives nine times as much data to process as the accelerator on the right, which is scheduled with II=9.

Table 5.5.: PPnR resource requirements and performance results for loop tiling implementations of the Harris corner detector using explicit data exchange.

R	LAT	SLICE	LUT	FF	BRAM	DSP	F [MHz]	TP [fps]	SU	ThSU
1	1050668	5059	13232	16066	5	63	221.4	210.7	1.0	1.0
1:1	528389	10516	26525	32629	16	124	218.0	412.6	1.9	1.9
2:1	704550	9753	27050	28784	16	102	194.5	276.0	1.3	1.4
3:1	793419	8796	24728	27181	16	93	219.0	276.0	1.3	1.3
4:1	844669	9012	24473	25812	16	88	213.1	252.3	1.1	1.2
5:1	902068	8645	23902	25336	16	87	228.8	253.6	1.2	1.1
6:1	910266	8438	23646	25395	16	87	215.7	237.0	1.1	1.1
7:1	921638	8755	23563	25370	16	87	215.2	233.5	1.1	1.1
8:1	951264	8619	23570	24202	14	81	221.2	232.6	1.1	1.1
9:1	959470	8466	22673	23667	13	81	215.3	224.4	1.0	1.0

The PPnR results for the alternative implementation of the Harris corner detector, which uses redundant computations to avoid the data exchange are listed in Table 5.6. The results of both tables are compared in Figure 5.17, where

Table 5.6.: PPnR resource requirements and performance results for loop tiling implementations of the Harris corner detector omitting explicit data exchange.

R	LAT	SLICE	LUT	FF	BRAM	DSP	F [MHz]	TP [fps]	SU	ThSU
1	1050668	5059	13232	16066	5	63	221.4	210.7	1.0	1.0
1:1	528388	9432	25783	30904	12	112	229.0	433.5	2.0	1.9
2:1	706563	8371	23093	25615	12	90	193.8	274.3	1.3	1.4
3:1	791556	8245	22886	25608	12	90	193.3	244.3	1.1	1.3
4:1	848712	7749	20940	22747	12	76	230.0	271.1	1.2	1.2
5:1	907136	7604	20485	22271	12	75	217.5	239.8	1.1	1.1
6:1	916360	7599	20321	22332	12	75	216.6	236.4	1.1	1.1
7:1	939934	7730	20553	22086	11	75	208.5	221.9	1.0	1.1
8:1	935840	8212	22977	25439	9	90	193.4	206.7	0.9	1.1
9:1	968632	6912	19280	20636	9	69	218.1	225.1	1.0	1.0

we have highlighted the throughput of the non-parallel version (R=1). In terms of the actual speedup, the results are unfortunately not as conclusive as expected, which is mostly due to variation in the achievable clock period. If we consider the theoretical speedup, we can argue that the speedup becomes insignificant, once we reach beyond a ratio of 4:1. Resource utilization, however, is mostly

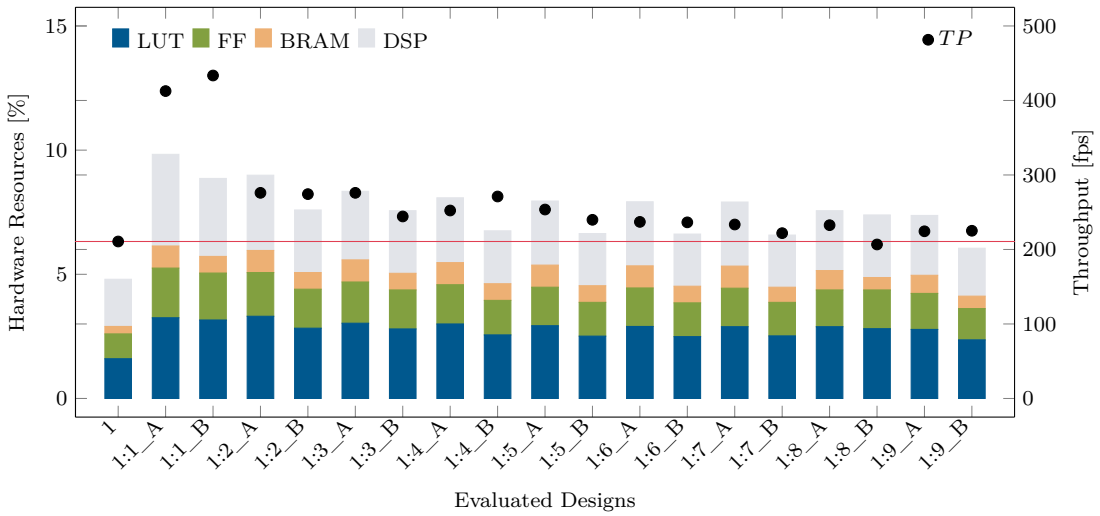


Figure 5.17.: Comparison of the PPnR results of the asymmetric loop tiling implementations of the Harris corner detector.

in favor of omitting the explicit data exchange. Especially in case of streaming pipelines with many local operators, this result can be expected to become more significant. In conclusion, we can determine that asymmetric tiling can deliver additional speedup if replicating a full speed accelerator is infeasible. Beyond a ratio of 4:1, however, the method can hardly be of benefit.

5.3. Loop Coarsening

The previous section discussed how the degree of parallelism in image processing can be raised using overlapped loop tilings on the algorithm and operation level. The evaluation has, however, shown that the method may not offer a linear speedup and may require a substantial overhead of both, logic and memory resources. In this section, we present *loop coarsening* as an alternative approach for parallel processing. The method makes use of aggregating input pixel data in so-called *superpixels*, where the amount of pixels contained in a superpixel is denoted by the superpixel length (SpL). In contrast to loop tiling, loop coarsening uses a cyclic distribution of the pixels contained in a superpixel and only replicates the loop kernel instead of the whole accelerator. In terms of the hardware implementation, all SpL pixels belonging to a superpixel are then processed in parallel. Thus, the approach does not require to synthesize any additional modules for data distribution. Although the methodology is similar

to vector operations in Single Instruction, Multiple Data (SIMD) architectures, there are distinct differences.

5.3.1. Differences to Vector Operations

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

5.3.2. Superpixel and Superwindow Concept

As local operators process the pixels of a local neighborhood, an important aspect is to reduce repeated memory access. For data streaming of single pixels as data units, a *line buffer* is typically used to retain data for repeated access and a register-based *memory window* provides the pixels for the local operator. An II of one clock cycle for local operators can only be achieved, if the access to each of the Dual-Ported Random Access Memory (DPRAM)-based line buffers is restricted to only two accesses per clock cycle. We apply the same concept for loop coarsening but introduce a hierarchy for the memory window, where a so-called *superwindow* is used for gathering superpixels from the line buffer and retain them for repeated access. From the superwindow, the pixels are extracted and assigned to *subwindows*, which are then used by the replicated kernel operators for processing (refer to Figure 5.18).

5.3.3. Data Separation and Border Treatment

A challenge for hardware generation using the window hierarchy is to determine the static wiring for separating and assigning the elements of the superwindow to the appropriate locations of the subwindows. For different superpixel lengths SpL and kernel sizes, the size of the superwindow ($W_y \times W_x$) may not match that of the subwindows ($w_y \times w_x$). For example, using a window of 3 pixels

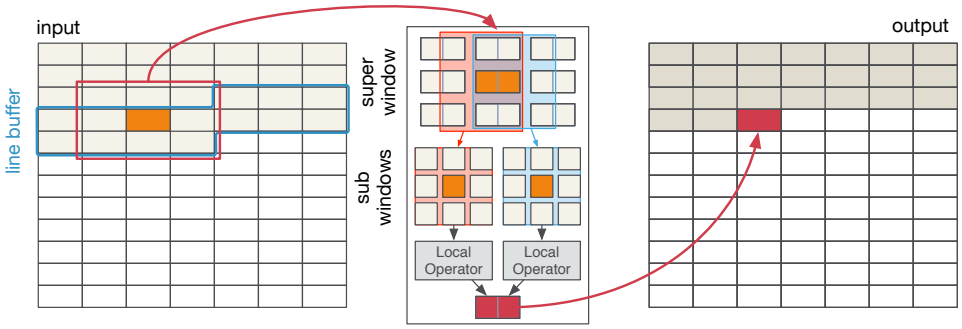


Figure 5.18.: Loop coarsening for parallel processing of superpixels ($SpL = 2$) with local operators by replicating only the kernel operator.

(radius $r = 1$) always requires a superwindow of the same size, regardless of the vector length. However, processing the image data with a window size of 9×9 pixels (radius $r = 4$) with $SpL = 2$, only needs a superwindow of size 9×5. Separating the data from the superwindow into the subwindows involves determining the correspondence between the elements of the windows. Both can be represented by triplets. To describe an element of the superwindow, we denote the point $(i, j, k) \in S_y \times S_x \times S_{sp}$, where $S_y = \{0, \ldots, W_y - 1\}$ and $S_x = \{0, \ldots, W_x - 1\}$ describe the position of the data beat in the superwindow and $S_{sp} = \{0, \ldots, SpL - 1\}$ describes the position of a pixel in the superpixel. The corresponding point in a specific subwindow is represented by $(i, j, k) \in s_y \times s_x \times P$, where $s_y = \{0, \ldots, w_y - 1\}$ and $s_x = \{0, \ldots, w_x - 1\}$ describe the window coordinates, and $P = \{0, \ldots, p - 1\}$ specifies the distinct subwindow. For calculating the correspondence between the windows, we must moreover determine the offset o of the first window in the continuous context of the data beats as

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

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

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

An example for one line using $SpL = 4$ and $W_x = w_x = 3$ is shown in Figure 5.19. At the top and at the 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 border treatment with data separation.

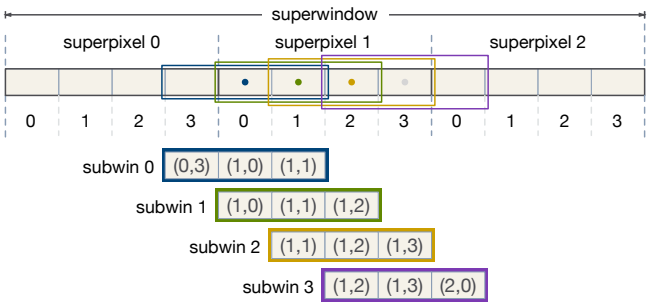


Figure 5.19.: Example for separating the data from the superwindow into subwindows. Here, $SpL = 4$ and $W_x = w_x = 3$.

5.3.4. Library Integration

We have included also loop coarsening as high-level abstraction in the library, presented in Chapter 4. In contrast to loop tiling, for which several functions had to be added for the complex architecture, loop coarsening only requires to change the `process` function. The library function prototype for loop coarsening is shown in Listing 5.10.

Listing 5.10: Library function prototype for loop coarsening.

```
template<int II_TARGET, int MAX_WIDTH, int MAX_HEIGHT, int KERNEL_SIZE,
  int VECT, typename INT, typename IN, typename OUT, class Filter>
void processVECTN(
  hls::stream<IN> &in_s,
  hls::stream<OUT> &out_s,
  const int &width,
  const int &height,
  Filter &filter,
  const enum BorderPadding::values borderPadding);
```

The function uses C++ template arguments to specify the factor for implicit vectorization (`VECT`) and the internally used data type (`INT`).

5.3.5. Evaluation

To assess the performance of loop coarsening, we have evaluated several of the algorithms discussed in Section 2.3, using the Xilinx Vivado Design Suite in version 2014.4 for implementation on a Xilinx Kintex 7 XC7K325TFFG900 FPGA. The algorithm specifications in C++ were synthesized into VHDL on the RT level using Xilinx Vivado HLS with high effort settings for scheduling and binding. PPnR hardware resource and performance characteristics were obtained by implementing the VHDL code as OOC module using the Vivado Design Suite. The provided measurements are explained in Section 4.6.2.

The PPnR resource requirements and performance characteristics for parallelizing the Laplacian operator with different factors v_i for *implicit vectorization*,

where $v_i = SpL$, are listed in Table 5.7. Supported by the evaluation of the

Table 5.7.: PPnR resource requirements and performance results for parallelizing the Laplacian operator using loop coarsening.

LP3HV										
v_i	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
2	525836	325	738	1331	2	0	366.4	696.9	2.0	2.0
4	263435	217	370	957	4	0	373.7	1418.5	4.0	4.0
8	132235	236	362	1068	4	0	368.6	2787.4	7.8	7.9
16	66635	1430	2725	7263	15	0	339.9	5101.0	14.3	15.8
32	33835	2259	4863	11523	28	0	356.5	10536.6	29.6	31.1
LP3D										
v_i	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
2	525837	584	1387	2428	2	0	351.6	668.7	1.9	2.0
4	263440	544	1262	2307	4	0	375.1	1423.8	4.0	4.0
8	132240	617	1368	3008	4	0	357.9	2706.5	7.6	7.9
16	66640	2880	6878	13525	15	0	310.1	4653.0	13.1	15.8
32	33840	5053	13602	24985	29	0	312.2	9225.98	26.1	31.0
LP5										
v_i	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
2	527380	2136	5874	9595	4	0	348.3	660.5	2.0	2.0
4	264737	2114	4564	9880	8	0	341.9	1291.4	3.8	4.0
8	133398	3608	8999	20697	16	0	331.9	2488.0	7.4	7.9
16	67734	10383	26546	48476	30	0	287.8	4248.5	12.6	15.5
32	34902	20099	39603	81036	116	0	228.9	6559.4	19.5	30.2

theoretical speedup (ThSU), the method can achieve near linear speedup. The difference is due to the increased group delay necessary for local operators, as discussed in Section 4.3.3. For the non-parallel accelerator ($v_i = 1$), the iteration space is enlarged by the group delay to 1025×1025. For loop coarsening, using two kernel operators for parallelization ($v_i = 2$), the iteration space must also be enlarged by the group delay to 513×1025. Although the difference is initially very small, it becomes noticeable for a high vectorization degree. For the Laplacian operators using a kernel of size 3×3, the clock frequencies are only moderately impaired, yielding an actual speedup (SU) that is very close to the theoretical value. The required resources and throughput are moreover depicted in Figure 5.20. Another observation is that the resources increase slower than expected. Since we execute the kernel operator multiple times and inline the code for compilation, the compiler recognizes when the same operation is executed multiple times on the same data and can eliminate the duplicates. Suppose a kernel of size 3×3 with a vectorization of $v_i = 4$, where each element of the window is multiplied with a constant coefficient. For simple duplication, the hardware implementation would require $9 \cdot 4 = 36$ multipliers. If the compiler

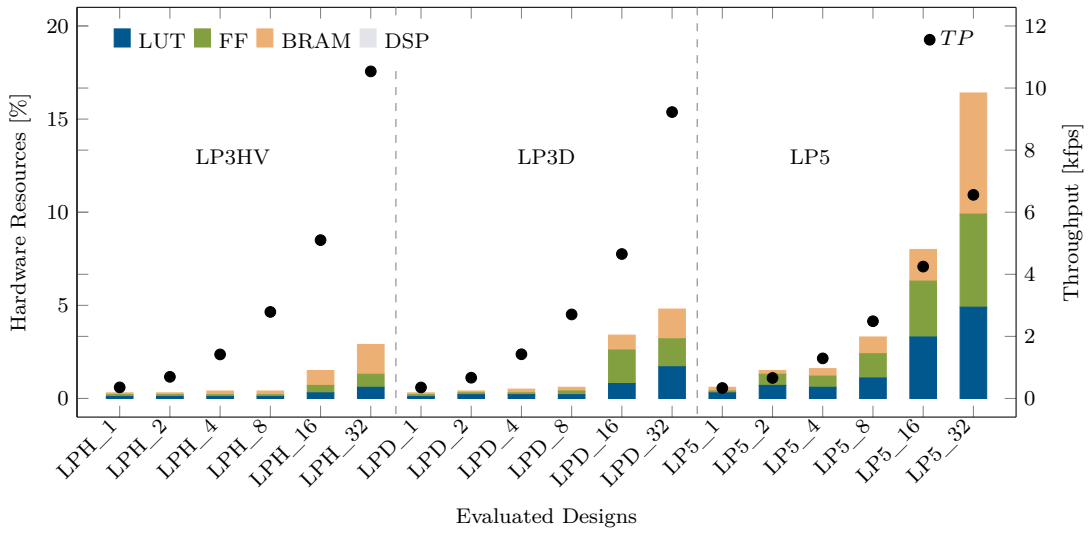


Figure 5.20.: PPnR FPGA resource requirements and performance measurements of the Laplacian operators using loop coarsening.

can eliminate the duplicates, it only requires $6 \cdot 3 = 18$ operators. PPnR results for up to 8 replicated kernel operators are listed in Table 5.8. We have moreover

Table 5.8.: PPnR resource requirements and performance results for applying loop coarsening to the Harris corner detector.

v_i	LAT	SLICE	LUT	FF	BRAM	DSP	F [MHz]	TP [fps]	SU	ThSU
1	1050668	5059	13232	16066	5	63	221.4	210.75	1.0	1.0
2	525867	8496	24356	30321	8	126	227.6	432.87	2.1	2.0
4	263467	15498	45514	54995	8	252	218.0	827.45	3.9	4.0
8	132268	28403	87916	104479	16	504	214.3	1620.31	7.7	7.9

evaluated the Harris corner detector as an example for a streaming pipeline. In contrast to loop tiling, we need not take any precautions regarding data availability for the local operators, but can simply apply loop coarsening to every instance of the pipeline. From the presented results, we can conclude that the methodology is ideally suited for accelerating pipelines, foremost due to its simplicity.

5.4. Comparison and Discussion

We have presented two methods for parallelization of FPGA hardware accelerators for image processing in this chapter. Both methods make use of aggregating

pixels to exploit data level parallelism on FPGAs. The first method, described in Section 5.2, is commonly known as *loop tiling* and uses the well-known *divide and conquer* concept to partition images into separate regions, which are then processed in parallel.

The second method, presented in Section 5.3, only replicates the actual operator kernel and is able to process multiple pixels within one accelerator in parallel. We refer to this method as *loop coarsening*.

5.4.1. Resource Requirements and Performance

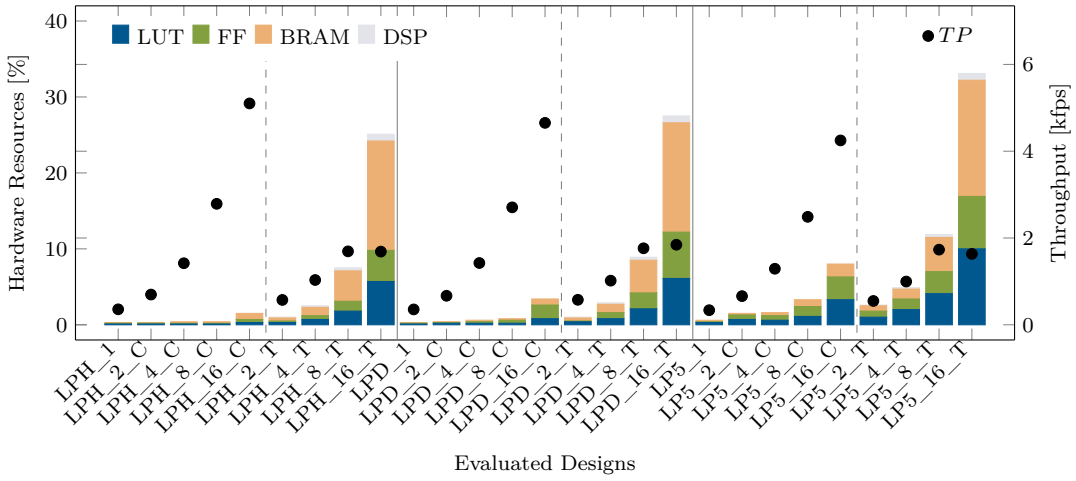


Figure 5.21.: Comparison of the resource requirements and performance characteristics for parallelizing the implementations of the Laplacian operator using loop coarsening (C) and loop tiling (T).

We first compare the resource requirements and performance characteristics obtained by parallelizing three different versions of the Laplacian operator. The results, depicted in Figure 5.21 are taken from Table 5.2 for loop tiling and from Table 5.7 for loop coarsening. Evidently, loop coarsening can achieve a significantly higher throughput while at the same time using considerably less resources. As we replicate only the operator kernel, this saves memory and logic required for the memory architecture, border treatment, and iteration counters. Furthermore, the compiler can optimize and eliminate redundant computations. Loop tiling, in contrast, must replicate entire accelerators and needs additional modules for data distribution. The resulting very high resource utilization of loop tiling also did not allow parallelization higher than 16 for tiling on the evaluated Kintex 7.

5.4.2. Speedup

We have furthermore analyzed the speedup that can be achieved by both approaches. Table 5.2 and Table 5.7 additionally list the theoretical speedup (ThSU), which is obtained by comparing the latency of the non-parallelized version ($R = 1$) to the latency in terms of clock cycles of the parallelized versions. The actual speedup (SU) is obtained using the throughput (TP), and thus reflects the influence of the clock frequency. Figure 5.22 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

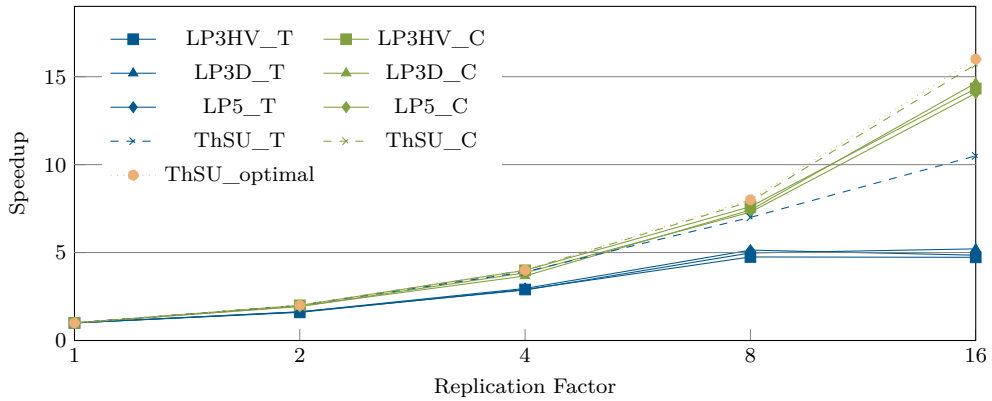


Figure 5.22.: Comparison of the actual and theoretical speedup achieved for parallelizing the implementations of the Laplacian operator using loop tiling (T) and loop coarsening (C).

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

5.4.3. Comparison to Software-based Accelerators

In our previous work, we have suggested to use Domain-Specific Languages (DSLs) and generate highly optimized code for synthesis by general purpose frameworks [Rei+14*]. Code generation offers true portability of programs and

performance across different platforms, and it delivers increased productivity, as developers need not be concerned about implementation details, but can focus on functionality. In combination with general purpose HLS frameworks, support of a wide range of applications can be accomplished efficiently. Our work utilized the open source DSL framework HIPAcc [Mem+15b] to generate highly optimized C++ code for Vivado HLS from Xilinx. HIPAcc is specifically targeted at accelerating image processing and can generate code for heterogeneous hardware targets, covering GPUs and FPGAs.

In [Sch+15a*], we have augmented the FPGA back end of HIPAcc with the ability to parallelize the generated FPGA accelerators with loop coarsening. 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 FPGA to the Nvidia Tegra K1 SoC as a representative for embedded GPU (eGPU), and the Nvidia Tesla K20 server-grade GPU. Table 5.9 lists the PPnR results obtained for algorithm specifications generated by HIPAcc, using the factor v_i for *implicit vectorization*. All of the algorithms use input images of size 1024×1024. Except for the variations of the Laplacian filter, which use 32-bit RGBA encoded color input data, the algorithms receive 8-bit unsigned integer input data. Using HIPAcc, we have moreover generated

Table 5.9.: PPnR results of algorithm implementations generated by HIPAcc on the Kintex 7.

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

Compute Unified Device Architecture (CUDA) code for implementation of the algorithms on the Tegra K1 and the Tesla K20 GPUs. The implementations were obtained using code generation based on *exactly the same DSL code*. To ensure a fair comparison, HIPAcc's exploration feature has been used to thoroughly evaluate suitable parameters for thread-coarsening, block size, and whether or not to use shared memory, in order to maximize the achievable throughput on the GPUs. Table 5.10 compares the throughput and energy efficiency in frames per Watt (E [fpW]) of the different accelerators. The power consumption of the GPUs can be estimated by considering about 60 % of the reported peak power values (2.28 W for the Tegra K1 and about 135 W for the Tesla K20). The throughput and energy efficiency are moreover compared in Figures 5.23 and 5.24, respectively. For algorithms that require many resources, the opportunity for parallelization of the FPGA accelerators is limited by the available resources

Table 5.10.: Comparison of the throughput (TP) and energy efficiency (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

on the FPGA. For example, we were able to replicate the kernel of the very simple Laplacian operators 32 times, however, the optical flow could only be replicated four times. In case of the GPUs, the throughput reduces dramatically for greater window sizes with an increased number of memory accesses, exposing the available memory bandwidth clearly as the main performance bottleneck.

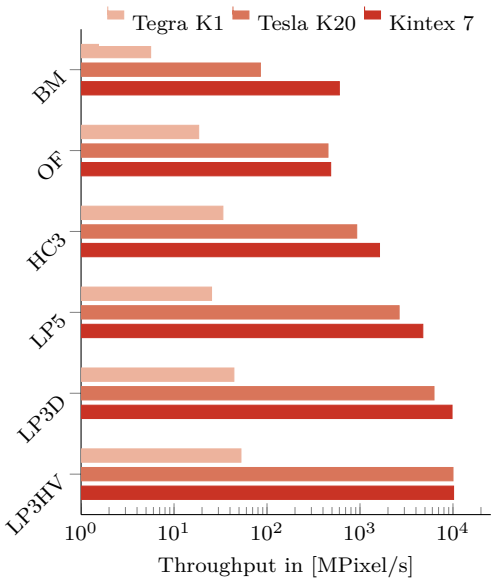


Figure 5.23.: Throughput comparison.

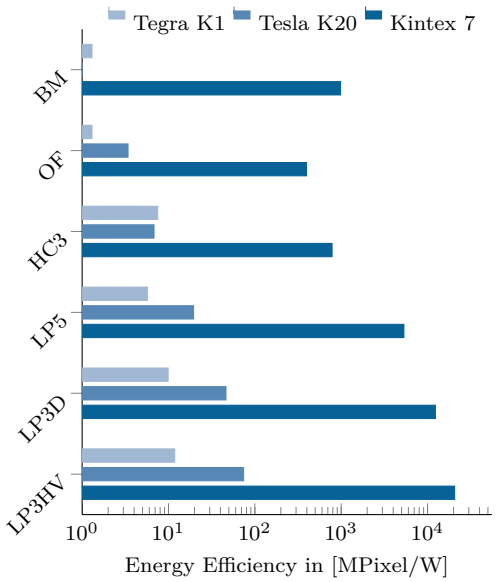


Figure 5.24.: Energy efficiency.

5.5. Summary

In this section, we have demonstrated how the general architecture to generate accelerators by HLS can be leveraged through *loop tiling* and *loop coarsening* to exploit DLP on FPGAs. We have shown that loop coarsening is highly scalable and delivers superior speedup than loop tiling.

Part II.

Interface Synthesis for FPGA-based SoC Integration of Image Processing IP Cores

As an Field Programmable Gate Array (FPGA) is initially unprogrammed, actually employing an image processing accelerator IP core, generated using High-Level Synthesis (HLS) as described in Part I, requires to integrate it into its on- and off-chip environment. In order to prepare automation of on- and off-chip integration, the second part of this work is concerned with *interface synthesis* for FPGA-based SoC integration of accelerator IP cores. According to Teich in [TH07], after the initial phases of *specification* and *hardware/software partitioning* at the Electronic System-Level (ESL), *interface synthesis* is a vital part between the subsequent hardware and software synthesis, as shown in Figure 5.25. Interface synthesis hereby specifies, allocates, and binds the

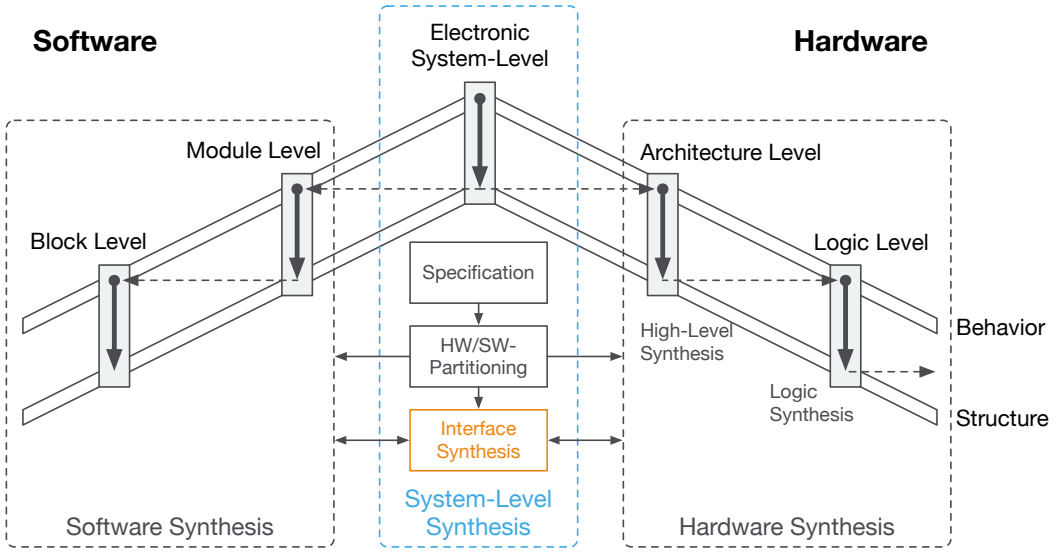


Figure 5.25.: Simplified view of system-level design and the vital role of *interface synthesis* adopted from Teich's *double roof* model [TH07].

necessary communication channels and protocols between the chosen system components. An example for the types of communication interfaces that might be encountered when integrating an accelerator Intellectual Property (IP) core into a distributed embedded system is given in Figure 5.26. To facilitate on-chip integration of accelerator IP cores into an FPGA-based System-on-a-Chip (SoC) design, Chapter 6 presents a solution for the architecture of an OCI that allows communication between the accelerator and other vital system components. In order to provide *transparent* communication, a key concern here is that any required conversion, for example, of data types, or between streaming-data and memory-mapped components, is implemented by the interconnect. Data and information exchange between the SoC and its off-chip environment is an equally important part of interface synthesis. In Chapter 7 we propose to establish

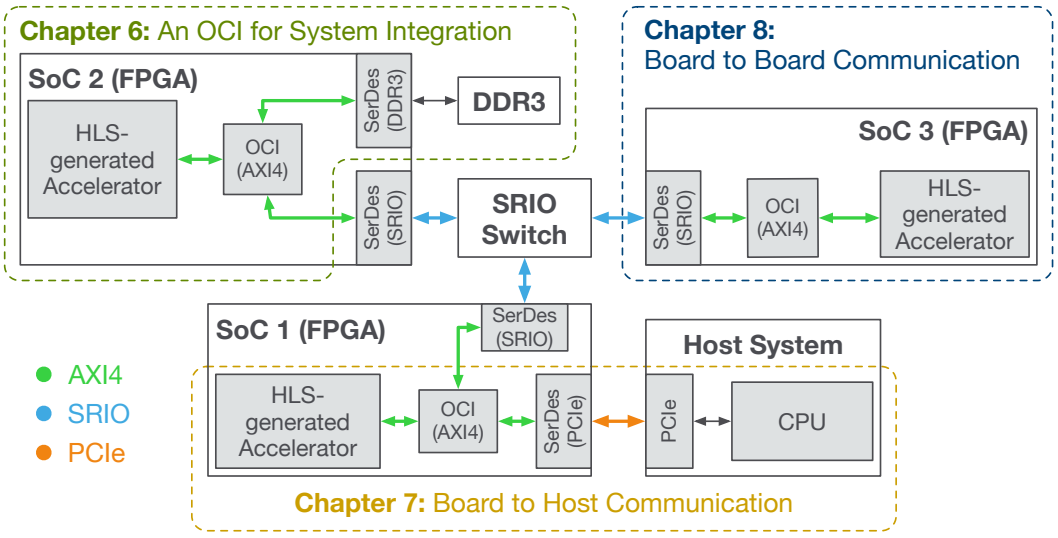


Figure 5.26.: Example of a distributed embedded system, comprising multiple FPGA-based SoCs. HLS-generated accelerator IP cores are integrated into the SoC designs using an AXI4S-base OCI. Board to board communication is established via a high-speed SerDes interconnect, for example SRIO. Data communication between the embedded system and a host computer uses PCIe.

communication between the FPGA-based accelerator and a host computer via PCIe in the form of a *hardware/software co-design*. Here, interface synthesis must provide a communication controller to be implemented as part of the SoC and also the necessary device drivers and software interfaces. Chapter 8 discusses board to board communication for distributed embedded systems using a high-speed Serializer/Deserializer (SerDes) interconnect. Although we make use of the SRIO protocol [Ful05], the approach followed here is also applicable to other SerDes protocols, for example Interlaken [GOW07].

6

An On-Chip Interconnect for System Integration

The previous chapters have provided details on how to prepare functional specifications for the High-Level Synthesis (HLS) of image processing Intellectual Property (IP) cores for Field Programmable Gate Array (FPGA) targets. In order to support rapid prototyping, we need to provide automatic synthesis also for data and information exchange among the system platform components, including, external memory, communication interfaces, but also embedded processors or controllers. An uprising trend to achieve this degree of automation for high-throughput streaming data applications is to design On-Chip Interconnects (OCIs) based on widely supported interface standards, such as the Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface 4 (AXI4), which provides specifications for streaming and memory-mapped data communication. Hereby, the OCI should be easily *adoptable* to a wide range of application scenarios without requiring very complex design tasks. Furthermore, access to the OCI and the system components should be completely *transparent* to the accelerator. This means that any necessary conversion requirements are handled by the interconnect and do not require any changes in the design of the accelerator. For passing configuration information, however, not all requirements can be realized using such an interconnect and must still be created by hand.

In this chapter, we first analyze the generic interface creation capabilities of Vivado HLS and describe how custom protocols can be implemented to interconnect accelerator IP cores with non-standardized system components. We proceed by introducing the AXI4 interface standard and describe how AXI4 interconnects can be constructed for high-throughput data communication between an accelerator and essential system platform components automatically. A case study to accelerate a multigrid algorithm is presented subsequently, which requires access to off-chip memory for intermediate data buffering.

The main contributions of this chapter can be summarized as follows:

- We propose *techniques for interfacing* image processing accelerators generated using only HLS. In detail, we propose how to synthesize interfaces for high-throughput streaming data communication and configuration parameter exchange between system platform components and hardware accelerators using HLS.
- We introduce a flexible, transparent, and *high-throughput OCI* based on AXI4 for a streaming data communication scheme between accelerators and other System-on-a-Chip (SoC) platform components. The interconnect is designed to be completely transparent to the accelertor and handles any required conversion, such as adapting data types and clock boundaries crossing.
- In the form of a *case study*, we demonstrate the design of an AXI4S-based OCI to facilitate transparent data exchange between an accelerator synthesized by Vivado HLS and off-chip memory resources.

6.1. Generic Interfaces in Vivado HLS

In Vivado HLS, the entry point for synthesis is a single top-level function, which contains the functional specification of the IP core. The interface of the component is hereby generated according to the arguments of the function. A certain interface protocol can be enforced through the `interface` directive where supported protocols greatly depend on the type of the function argument. Table 6.1 lists different function arguments types and the supported interface protocols.

Table 6.1.: Data type and interface support in Vivado HLS. D specifies default interfaces

Argument Type	Variable Pass By Value			Pointer Pass By Reference			Array Pass By Reference			Reference Pass By Reference		
Interface	I	I/O	O	I	I/O	O	I	I/O	O	I	I/O	O
ap_none	D			D						D		
ap_stable												
ap_ack												
ap_vld						D						D
ap_ovld					D						D	
ap_hs												
ap_memory							D	D	D			
ap_fifo												
ap_bus												
ap_ctrl_none												
ap_ctrl_hs			D									
ap_ctrl_chain												

Vivado HLS differentiates between C *function* and *block interfaces*. C function interfaces define how data is passed to the synthesized IP core, whereas the block level interface defines how the IP core is controlled, for example, when to start

execution or when the block is ready to accept new data. Function interfaces are applied to individual function arguments, block-level interfaces are applied to the function itself.

6.1.1. Block-Level Interfaces.

The standard block-level interface is `ap_ctrl_hs`, which implements a basic handshake. Its behavior is displayed in Figure 6.1. The IP core uses the

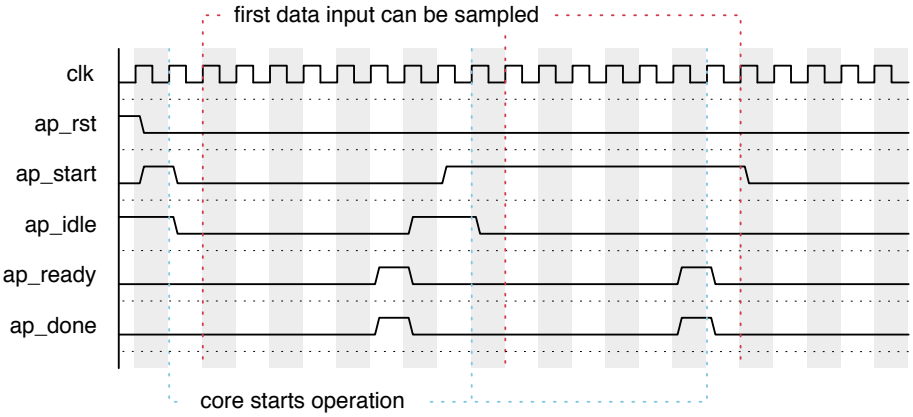


Figure 6.1.: Behavior of the `apctrl_hs` standard block-level interface in Vivado HLS

`ap_idle` signal to indicate that it is ready for operation. The signal is deasserted immediately after the `ap_start` signal has been asserted. The core asserts `ap_ready` after all the input data have been consumed and sets `ap_done` to high, once it has finished the computation and delivered all computation results to the output ports. Note that `ap_done` and `ap_ready` are asserted only for a single clock cycle. The port `ap_idle`, however, is asserted in the same clock cycle as the `ap_ready` and `ap_done` ports, but stays asserted until the `ap_start` signal is asserted again. One important aspect regarding `ap_idle` is concerned about input data sampling, which, in contrast to ready-valid combinations, does not happen at the rising clock-edge, where both idle and start are asserted, but one clock cycle later. Also, if `ap_start` is kept asserted after `ap_idle` goes down and remains in this state until `ap_ready` goes high, the core immediately resumes operation and will not assert `ap_idle`.

As Vivado HLS supports *pipelining* of not only loops but also subfunctions within a top-level function, iterations may be overlapped, so that the next iteration may already start, before the current iteration is finished. This is the case if the top-most loop is pipelined with the `-rewind` option. The core asserts `ap_ready` once it can accept new data and the next iteration can be started

before the current iteration ends. In order to chain blocks synthesized using Vivado HLS, the interface ap_ctrl_chain provides an extra ap_continue port to signal to the current block, whether the consumer is ready to accept new data. The functionality is essentially that of the ready signal of a streaming interface, although, the current block does not stall immediately, if the continue signal is deasserted but continues the operation until the downstream-block must consume the data. In other words, this *back pressure* signal allows a block to stop its producer from generating new input data.

In case of pipelined subfunctions, Vivado HLS synthesizes each function into separate block and function interfaces, as well as generates appropriate control logic for interconnecting these modules. This case is peculiar, as the block interface of the top-level module does not behave in exactly the same way, as it would in the single block case. Here, the ap_idle signal cannot be used to determine whether the function is able to start a new iteration, as the signal is generated by combination of the ap_idle signals of the individual blocks. Instead of waiting for the user of the top-level function to assert ap_start, the embedded function blocks are started automatically upon the release of ap_reset, causing the top-level ap_idle to be deasserted. Hence, the combined module must be initiated by asserting ap_start in the absence of an asserted ap_idle.

6.1.2. Function Argument Interfaces

The most simple interface for function arguments is the ap_none interface. It puts all the responsibility to generate data at the correct time and hold it for the required length into the hands of the producer block. The ap_stable interface also does not add any handshaking signals to the port. In contrast to ap_none ports, Vivado HLS expects that data on a stable port remain stable during the normal operation, but may change between executions of a block. A typical example, for which to use stable ports are non-constant configuration signals which should not be optimized.

The ap_hs interface provides an *acknowledge* signal (ap_ack) to indicate when data has been consumed and a *valid* (ap_valid) signal to indicate when data has been produced. The interface is a superset of ap_ack, ap_vld, and ap_ovld, for which

- ap_ack provides only the acknowledge signal,
- ap_vld provides only the valid signal,
- ap_ovld provides a valid signal only to the output port and the input defaults to ap_none.

The behavior of the ap_hs interface is illustrated in Figure 6.2.

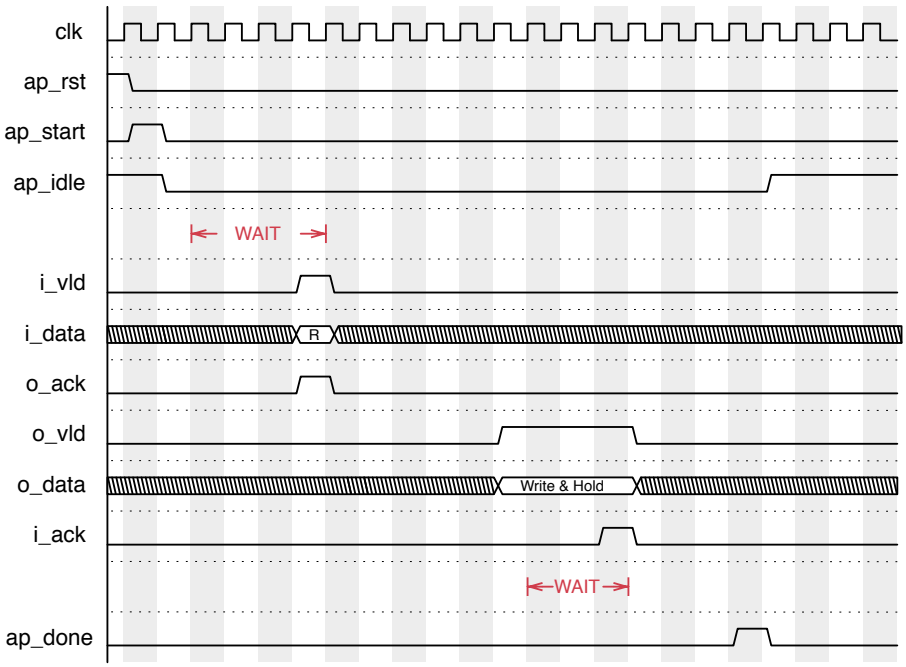


Figure 6.2.: Behavior of the `ap_hs` function argument interface.

For each input or output function argument, Vivado HLS adds `<signal>_vld` and `<signal>_ack` ports to facilitate the handshake protocol, where `<signal>` corresponds to the actual name of the function argument. The functionality is comparable to *ready-valid* interfaces, where the producer controls the `valid` signal to indicate valid data on the bus and the consumer shows the ability to consume data by asserting the `ready` signal. The difference between ready and acknowledge signals is hereby, that for `acknowledge` to be asserted, the valid signal must have been asserted first, whereas a ready signal is asserted as soon as the consumer is able to consume data, in general. The subset interface either does not synthesize the `valid` or the `ack` signal.

Array arguments default to the `ap_memory` interface, which is useful to communicate with Block Random Access Memorys (BRAMs) and Read Only Memorys (ROMs), as shown in Figure 6.3. A memory typically takes two clock cycles for a read and one clock cycle for a write. As it can be read from and written to, Vivado HLS synthesizes a *read* (`<port_name>_q`) and *write* (`<port_name>_d`) port, the associated *enable* signals (`<port_name>_ce` for read and `<port_name>_we` for write), as well as an *address* port (`<port_name>_address`). For dual-ported memories, the ports are differentiated by appending `0` and `1`. The behavior of the memory interface is illustrated in Figure 6.4.

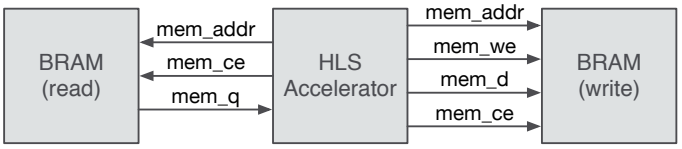


Figure 6.3.: Signals for interfacing a BRAM for reading (left) and writing (right).

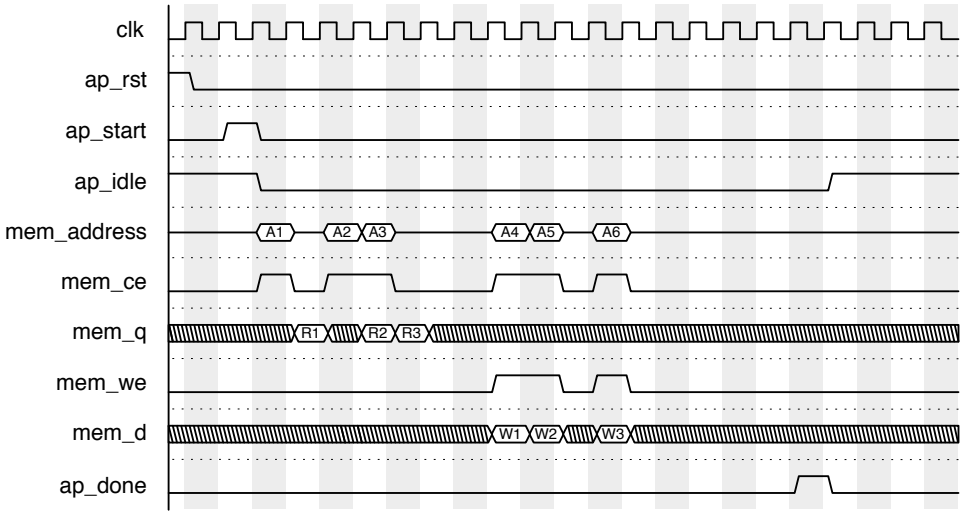


Figure 6.4.: Behavior of the `ap_memory` function argument interface.

Write or read operations can start one clock cycle after the idle signal has been deasserted. A read is performed by specifying the address of t he data and asserting the clock enable port. The desired data is delivered one clock cycle later, so that reads from memory will always take two clock cycles. In order to write to memory, it is necessary to specify the address and assert write enable in addition to the clock enable. A write is performed within one clock cycle.

Pass-by-reference arguments, such as *pointers* and *references*, can be synthesized using the `ap_fifo`-interface, if they do not require random access to the elements and are either read from or written to. For arrays, the First In, First Out (FIFO) interface can only be synthesized if the array is accessed from lowest to highest, i. e., `for(i=0;i<N;i++)...`. Otherwise, for example, if the array is accessed from highest to lowest, Vivado HLS determines a non-sequential access and cannot synthesize a FIFO interface. The interface allows the port to be connected to a FIFO, as shown in Figure 6.5, supporting full duplex communication using `write` and `full`, as well as `read` and `empty` control signals. The interface behaves as illustrated in Figure 6.6. The FIFO interface uses active low interfacing to signal if the input FIFO is empty and to indicate that the

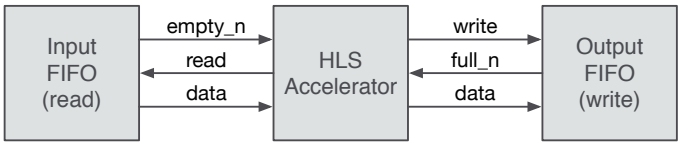


Figure 6.5.: Signals for interfacing a FIFO for reading (left) and writing (right).

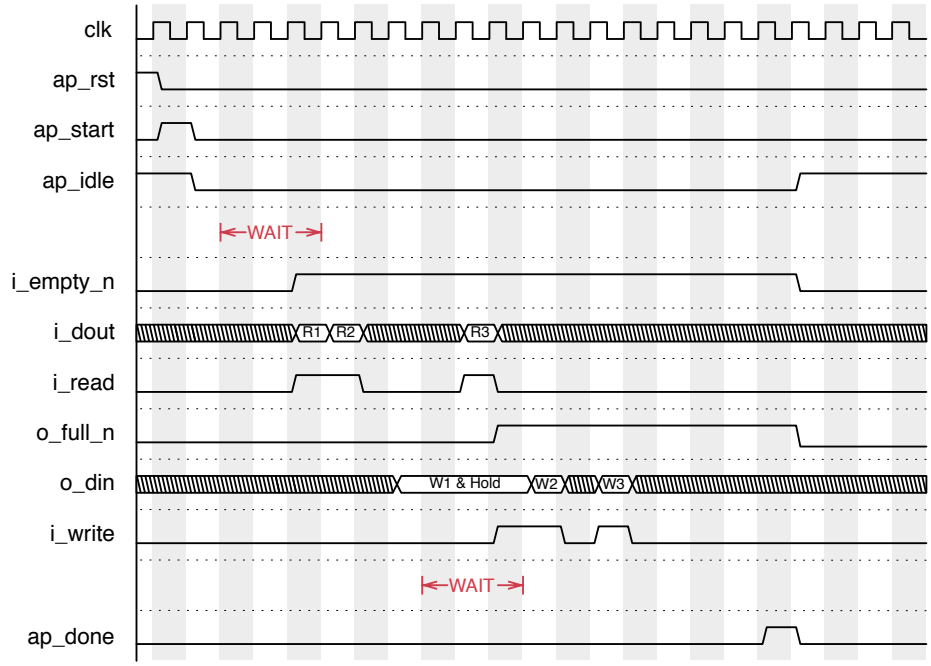


Figure 6.6.: Behavior of the `ap_fifo` function argument interface.

output FIFO is full. In both cases, the core must wait until the input FIFO contains data and the output FIFO has space left. To obtain data from the input interface, the core asserts the read port, which follows the acknowledge paradigm. In turn, if data was successfully written to a non-full output FIFO, the core acknowledges the operation by asserting the write port. FIFO interfaces are also supported for pipelined top-level functions that use the `-rewind` option. In this case, the last write of an iteration is indicated by an optional `_lwr` signal.

6.1.3. Creating Custom Protocol Interfaces in Vivado HLS

Configuration information for an accelerator, for example, received by off-chip communication, is often exchanged through a so-called shared register, which is essentially a very small Random Access Memory (RAM). Although these

values could be hard wired to a *stable* interface port, such a methodology might require long routes and thus may cause timing problems. A preferred approach is to implement a custom hand-shake mechanism to load individual parameters, which only requires to communicate the adress, the value, and the handshake signals.

An accelerator would usually access the shared register before it starts processing a new image to obtain the configuration parameters and write information to the register after processing in case it must provide meta data. Vivado HLS supports the specification of custom protocol behavior in dedicated code blocks. When the code within the section is implemented, the scheduling might keep the specified order of instructions or change it. The execution order can be enforced using the `HLS PROTOCOL` pragma. In addition, delays between operations, such as to wait for a response between issuing a read address and consuming the return data, the `ap_wait()`, or `ap_wait_n(int)` functions must be called.

A possible implementation reads values from the shared register by supplying the number of the register and a valid signal. The values must be kept stable until the register has processed the read, put the contents of the address on the data bus, and asserted the read by an acknowledgement. Listing 6.1 shows how this protocol can be implemented using Vivado HLS to read the register content.

Listing 6.1: Code in C++ to generate a custom interface for reading from a shared memory location with address `rd_addr`.

```
1  #include<ap_utils.h>
2
3  void func(int *rd_value, int *rd_addr)
4  {
5    #pragma HLS INTERFACE ap_hs register port=rd_addr
6    READ_REG:
7    {
8      #pragma HLS PROTOCOL floating
9      *rd_addr = 0x9004;
10     ap_wait_n(2);
11     img_size_x = *rd_value;
12     *rd_addr = 0x9008;
13     ap_wait_n(2);
14     img_size_y = *rd_value;
15   }
16 }
```

To force Vivado HLS to create valid and acknowledge signals, we use the `HLS INTERFACE` pragma to generate a handshaking interface for `rd_addr` and use `register` to keep the output stable. The corresponding module ports are shown in Listing 6.2.

Using the handshaking protocol in combination with the wait statement lets the flow of execution only continue if the acknowledge for the address transaction has been received, which is when the register content is ready to be consumed from `rd_value`.

Listing 6.2: Generated entity ports for interfacing the register.

```
entity func is
port(
  rd_value        : in std_logic_vector(31 downto 0);
  rd_addr         : out std_logic_vector(31 downto 0);
  rd_addr_ap_vld  : out std_logic;
  rd_addr_ap_ack  : in std_logic
);
end entity;
```

Writing to a shared register follows the same procedure, i. e., we must supply the value to be written, the address, and the valid signal and keep it stable until the shared register acknowledges the write. The corresponding code for writing the value `0xBEEF` to address `0xF00D` is shown in Listing 6.3.

Listing 6.3: Code in C++ to generate a custom interface for writing to a shared memory location with address `wr_addr`.

```
#include<ap_utils.h>

void func(int *wr_value, int *wr_addr)
{
  #pragma HLS INTERFACE ap_hs register port=wr_addr
  WRITE_REG:
  {
    #pragma HLS PROTOCOL floating
    *wr_value = 0xBEEF;
    *wr_addr = 0xF00D;
    ap_wait();
  }
}
```

Clock Domain Crossing

For communication via a shared register we must exchange a flag and data bus signals. If the register is inside an independent clock domain, we must apply appropriate Clock Domain Crossing (CDC) logic to ensure correct functionality. The methodology for syncing flags across different clock domains must ensure that a single cycle impulse in the originating clock domain will also only last a single cycle in the destination domain. In contrast, since the write and read is controlled by handshaking, the data bus signals must simply be adapted from one domain towards the other to ensure they are stable across a rising edge of the target. We have implemented dedicated CDC modules to properly synchronize the handshake and data signals between the HLS component and the register. To synchronize a flag, we must make use of a more complicated logic, which is described in Listing 6.4. An example of the behavior of the synchronization

Listing 6.4: Synchronizing a flag between different clock domains.

```
1  entity sync_flag is
2  port(
3    rst: in std_logic;
4    clka : in std_logic;
5    flag_clka : in std_logic;
6    clkb : in std_logic;
7    flag_clkb : out std_logic
8  );
9  end entity;
10
11 architecture rtl of sync_flag is
12
13   signal flag_toggle, s_flag_clkb : std_logic := '0';
14   signal synca : std_logic_vector(2 downto 0) := b"000";
15
16 begin
17
18   process(clka,rst)
19   begin
20     if(rst = '1')then
21       flag_toggle <= '0';
22     elsif(rising_edge(clka))then
23       flag_toggle <= flag_toggle xor flag_clka;
24     end if;
25   end process;
26
27   process(clkb,rst)
28   begin
29     if(rst = '1')then
30       synca <= b"000";
31     elsif(rising_edge(clkb))then
32       synca <= synca(1 downto 0) & flag_toggle;
33     end if;
34   end process;
35
36   s_flag_clkb <= synca(2) xor synca(1);
37   flag_clkb <= s_flag_clkb;
38
39 end architecture;
```

mechanism is illustrated in Figure 6.7. The toggle signal changes only when the flag is asserted at the faster clock domain but keeps it value steady when the flag is deasserted. At the slower clock domain, it passes through a shift, where the two most significant registers are connected by an `xor` gate. This produces a flag signal at the slower clock rate that is asserted for exactly one clock cycle. Using this mechanism, we can assure that even if the accelerator and the shared register are in different clock domains, the custom protocol implemented in Vivado HLS executes a single read or write transaction for every parameter value.

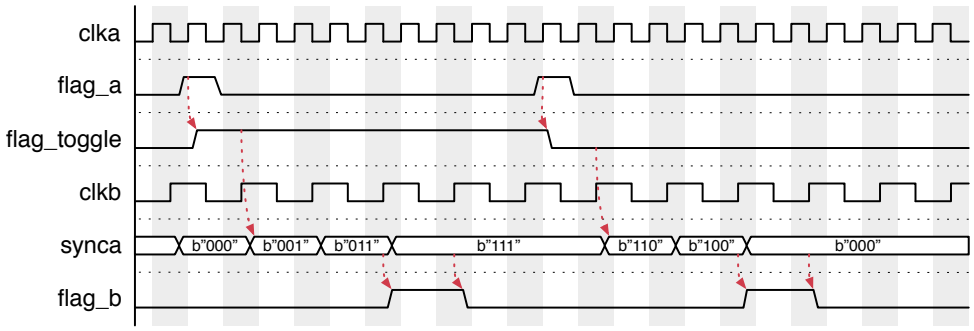


Figure 6.7.: Example for CDC by flag synchronization.

6.2. AXI4 Interfaces

To facilitate high-throughput data exchange between an accelerator and its environment in the SoC architecture, we make use of the open-standard AMBA [ARM14], which is an OCI specification specifically designed for connection and management of functional components of SoC designs. AXI4 is the fourth generation of the specification, which was adapted by Xilinx [Xil15a] as a new standard interface for their IP cores. AXI4 specifies three basic versions of the interface:

1. **AXI4**, which is intended for high-throughput memory-mapped communication,
2. **AXI4-Lite**, a reduced interface for low-throughput memory-mapped data and control communication, and
3. **AXI4-Stream**, designed for high-speed streaming data communication.

Compared to generic and legacy interfaces by Xilinx, the AXI4 specification provides a highly flexible family of interoperable interconnects. Moreover, the adoption of the ARM standard allows interoperability between the embedded ARM CPU cores and reconfigurable logic, which is essential for SoC-designs, such as the Xilinx Zynq FPGA family. The AXI specifications hereby define the communication between a single AXI4 master and single AXI4 servant. Memory-mapped and streaming designs requiring many-to-many communication between masters and slaves can be connected via an *interconnect*, allowing a very high degree of freedom for adapting the communication to various design requirements.

6.2.1. AXI4 Memory-Mapped Interfaces

Memory-mapped communication, supported by the fully fledged AXI4 and the reduced AXI4-Lite interfaces specifies individual channels for read and write

addresses, read and write data, as well as a write response channel. Data can move in both directions, as a master can write to and read from a slave simultaneously. Both, reads and write are referred to as *transactions*, where AXI4 supports a burst of up to 256 data transfers and AXI4-Lite allows only a single data transfer per transaction.

6.2.2. AXI4-Stream Intefaces

The AXI4-Stream (AXI4S) specification has been developed for data-flow applications requiring a high performance at a very high degree of flexibility. Each AXI4-Stream acts as a single uni-directional channel with handshake mechanisms at both sides of the interconnection. It is a relatively simple, yet powerful interface for large data distribution that also differentiates between master and servant interfaces, where data is passed from the master to the servant. Table 6.2 describes the typical interface signals.

Table 6.2.: AXI4S port signals.

Signal	Source	Description
`tvalid`	Master	Master has put valid data on the bus
`tready`	Servant	Servant is ready to consume data from the bus
`tdata`	Master	Primary payload
`tstrb`	Master	Marks position byte
`tkeep`	Master	Marks NULL byte
`tlast`	Master	Packet boundary
`tid`	Master	Indicates different data streams
`tdest`	Master	Target destination for routing
`tuser`	Master	Sideband user information

An example transmission between a master and a servant is shown in Figure 6.8. The only mandatory signals are `tdata` and `tvalid`, all other signals are optional and commonly referred to as *sideband* signals. In Xilinx IP cores that support AXI4S, `tdata` is limited to a whole number of bytes, i. e., 8 bits, 16 bits, and so on. The `tkeep` signal can be understood as a valid byte mask for a packet remainder, meaning that it is typically only evaluated in cycles where `tlast` is asserted as well. The AXI4 specification defines `tstrb` to support sparse streams, which may contain `NULL`-bytes, however, leading or intermediate invalid bytes are not supported. An interconnect network built from AXI4S components can multiplex multiple data streams, which are identified by their `tid` and `tdest` fields, where `tid` identifies the source of the stream and `tdest` determines its target destination. The FIFO interface synthesized by Vivado HLS is also compatible to AXI4S interfaces, as shown in Figure 6.9.

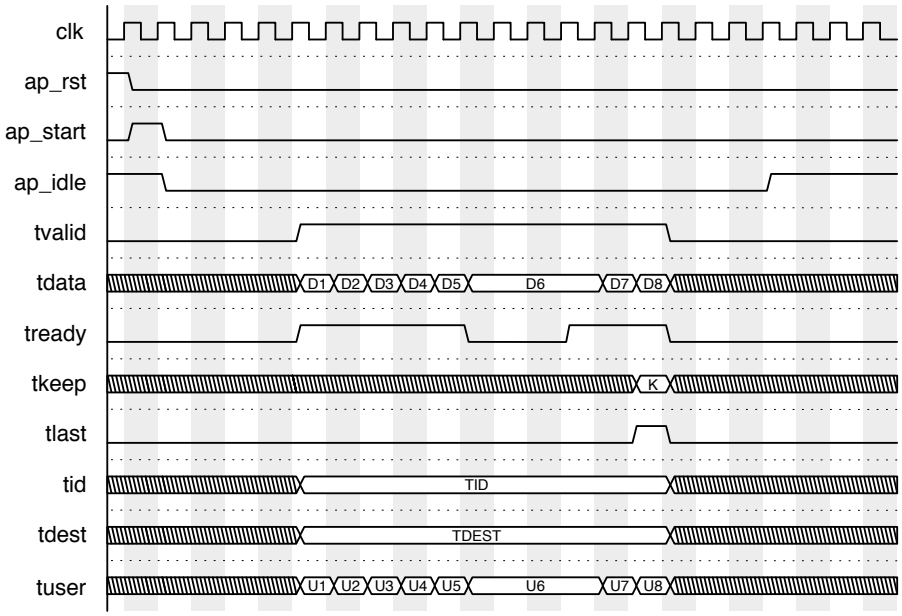


Figure 6.8.: Behavior of the AXI4-Stream interface.

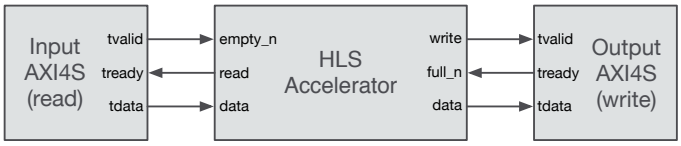


Figure 6.9.: Signal compatibility for interfacing AXI4S for reading (left) and writing (right) using a FIFO interface.

6.2.3. Support for AXI4-Interfaces in Vivado HLS

Vivado HLS supports the generation of AXI4 interfaces only for block-level port interfaces. Listing 6.5 shows a simple AXI4S interface, only carrying the mandatory interface ports and `tready`.

Listing 6.5: Simple AXI4S interface in Vivado HLS.

```
1  #define LEN 50
2  #include <hls_stream.h>
3
4  void func(hls::stream<int> &in, hls::stream<int> &out)
5  {
6    #pragma HLS INTERFACE axis port=in
7    #pragma HLS INTERFACE axis port=out
8
9    int i, temp;
10
11   for(i = 0; i < LEN; i++)
```

```
12    {
13      in >> temp;
14      out << temp + 5;
15    }
16  }
```

Sideband channel information can be inlcuded using a templated `struct` as function argument, as shown in Listing 6.6.

Listing 6.6: Templated struct for sideband information.

```
1  #include "ap_int.h"
2
3  template<int D, int U, int TI, int TD>
4  struct ap_axis{
5    ap_int<D>     tdata;
6    ap_uint<D/8>  tkeep;
7    ap_uint<U>    tuser;
8    ap_uint<1>    tlast;
9    ap_uint<TI>   tid;
10   ap_uint<TD>   tdest;
11 };
```

This struct can then be used as part of a Vivado HLS IP core as shown in Listing 6.7.

Listing 6.7: AXI4S interface using sideband information.

```
1  #include <hls_stream.h>
2  #include "func.h"
3
4  #define LEN 50
5
6  void func(hls::stream<ap_axis<32,2,5,6>> &in; hls::stream<ap_axis
     <32,2,5,6>> &out)
7  {
8    #pragma HLS INTERFACE axis port=in
9    #pragma HLS INTERFACE axis port=out
10
11   int i;
12   ap_axis<32,2,5,6> temp_in, temp_out;
13   for(i = 0; i < LEN; i++)
14   {
15     in >> temp_in;
16     temp_out.data   = temp_in.data.to_int() + 5;
17     temp_out.keep   = temp_in.keep;
18     temp_out.tuser  = temp_in.tuser;
19     temp_out.tlast  = temp_in.tlast;
20     temp_out.tid    = temp_in.tid;
21     temp_out.tdest  = temp_in.tdest;
22     out << temp_out;
23   }
24 }
25
```

In addition to the simple AXI4S interfaces, Vivado HLS can also generate AXI4-Lite and full AXI4 interfaces using Xilinx IP cores. In case of AXI-Lite, the interface is defined using `s_axilite` as interface type in combination with the `bundle` command. The IP core is then generated in conjunction with the `ap_bus` interface to implement the behavior of the bus bridge. The `standard` mode

of operation performs individual read and write operations and is suitable for implementation of an AXI4-Lite interface. Moreover, it can also be operated in *burst mode*, which implements the full AXI4 memory-mapped interface.

6.2.4. AXI4S Interconnect

Multiple AXI4S master and servant components can be interconnected using an AXI4S interconnect. The purpose of the interconnect is hereby to adapt the master data and protocol to ideally suit the requirements of the servant nodes. The center piece is the *switch*, which can act as a full $m : n$ crossbar. There are two interesting use cases for constructing an AXI4S interconnect, one is to route and switch streaming data, which is depicted in Figure 6.10 Using such a

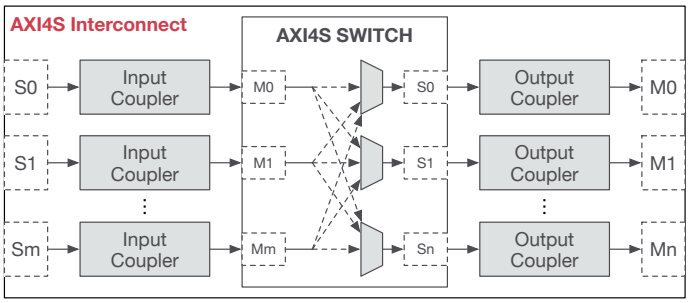


Figure 6.10.: General structure of an AXI4S interconnect to route and switch data streams.

design, data streams can be routed between different consumer and producer nodes in an FPGA design efficiently. Moreover, the AXI4S interconnect can also be constructed to multiplex different data streams onto a memory-mapped peripheral, as illustrated in Figure 6.11. Such a design can for example be used as an abstraction for intermediate buffering of individual data streams on an external memory, or to enable access to a processor bus.

Both types of architectures require input and output *couplers* to condition ingress and egress data. An illustration of a basic structural representation is given in Figure 6.12. For incoming data, the first step is to adapt the protocol to the internally used signaling subset by a *subset converter*. Recall that all signals but `tdata` and `tvalid` are optional. It is thus possible, that master and servant nodes only use a subset of the internally used signals. For example, in case of static routing, it might not be necessary that the master and servant nodes use the `tdest` signals. A subset converter would then insert the destination ID of the target output as a first step in the input coupler. Furthermore, the width of the data Input/Output (I/O) of a node might not match that of the destination node, or that of the interconnect. For these occasions *up-* and *downsizer* elements of

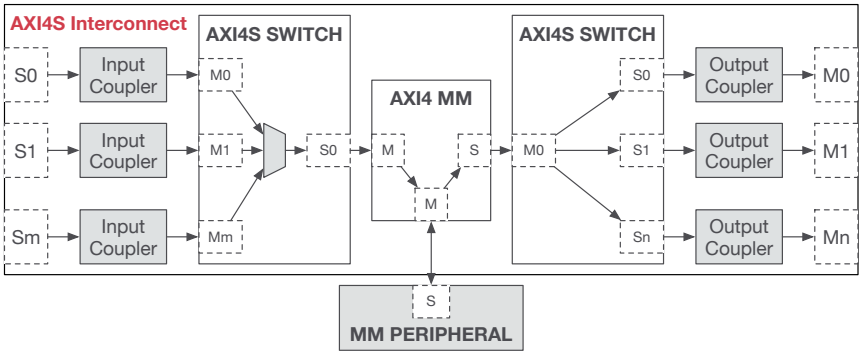


Figure 6.11.: General structure of an AXI4S interconnect to multiplex data streams onto a memory-mapped peripheral.

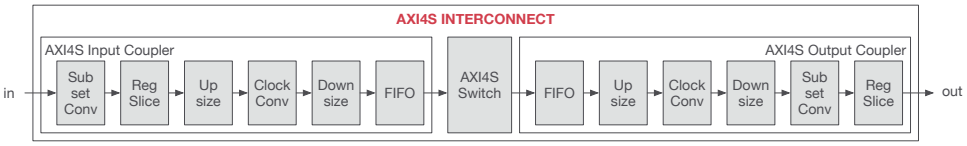


Figure 6.12.: Structure for input and output couplers to adapt data streams for AXI4S interconnects.

the couplers adjust the datawidth. Moreover, the master and servant nodes might be in different clock domains, or not share the interconnects clock frequency. To adapt the clock rate it is either possible to use a dedicated *clock converter* or an *asynchronous FIFO*. The purpose of the FIFO element is foremost to buffer incoming data elements. To relax timing requirements, it is also possible to include *register slices*. For the egress path, the *output coupler* simply uses the components in reverse order.

Often, it is not necessary to use every component of the interconnect in Figure 6.12. For example, most couplers only need to either upsize or downsize the data-width and also the FIFO buffers might not be necessary for the switch architecture, but are essential for multiplexing streams onto a memory-mapped component.

6.3. DDR3 Memory Abstraction Layer

As the memory resources are very restricted inside an FPGAs, applications may require access to off-chip memory, which may implemented as Double Data Rate

Type Three (DDR3)Synchronous Dynamic Random Access Memory (SDRAM)[9], due to its performance. Communication with such memories is typically achieved using a memory-mapped interface. In order to facilitate seamless integration of peripheral off-chip memories with the data streaming methodology used for the accelerators, we construct an AXI4S interconnect, as depicted in Figure 6.11. The AXI4S switch connects to an AXI4S virtual FIFO, an IP core from Xilinx that serves as an abstraction layer on top of off-chip DDR3 memory. An overview of the interconnect is given in Figure 6.13. The virtual FIFO can supports up to

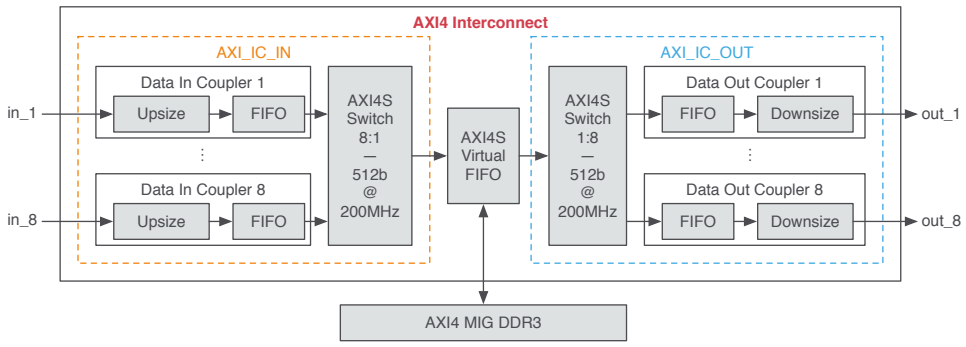


Figure 6.13.: Structure of an AXI4S interconnect to serve as an abstraction layer for DDR3 memory.

8 individual full duplex data channels, which are served in a round robin manner. To achieve a transparent interconnect and hide the multiplexing, we use the maximum word length and clock frequency supported by the DDR3 memory, which is 512 b at a clock rate of 200 MHz, to obtain a very high throughput. The path leading from a specific input to an output is referred to a a channel and is illustrated in Figure 6.14. In order to adapt incoming data to be stored in the

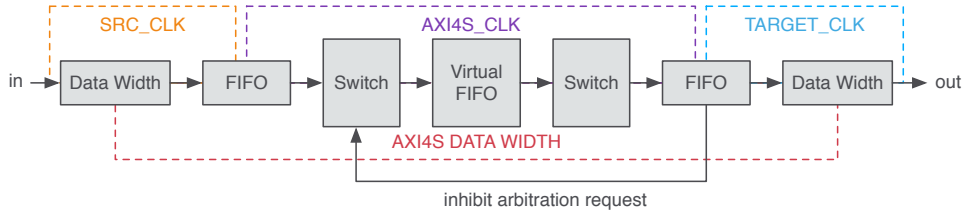


Figure 6.14.: Structure of a single channel of the AXI4S interconnect.

DDR3 memory, it is first necessary to adapt the data width of the source to the internally used 512 b, which is performed by the AXI4S data width converter.

[9]In this work we use the terms *DDR3* and *DRAM* synonymously to describe DDR3-SDRAM.

Following, we adopt the data stream to the internal clock speed of 200 MHz using the FIFO, which is also employed to decouple incoming data from the scheduling interval of the proceeding components of the interconnect. On the outgoing path, data must be adopted to the bit width of the target and its clock rate. Evene if there is no need to adapt the clock rate in this path, the data FIFO is still essential to accommodate for DDR3 burst data transfers in the case that the target instance happens to not accept data. The size of the outwards leading data FIFO must be carefully determined to be able to hold data from one full DDR3 scheduling interval which is determined by the configurable burst read size. As the data width converter and the data FIFO are straight forward components, we describe the virtual FIFO and the AXI4S switch component in more detail in the next sections.

AXI4S Virtual FIFO

As explained earlier, the AXI4S virtual FIFO (VFIFO) serves as an abstraction layer on top of the off-chip DDR3 memory which is interfaced by an appropriately configured Xilinx Memory Interface Generator (MIG) instance, using the AXI4 protocol. Configuration options for the VFIFO include the number of channels, as well as the maximum burst size in bytes for the memory interface. The VFIFO supports up to 8 channels which can be configured to hold between 32 and up to 256 MB of DRAM space, according to the selected burst size, ranging from 512 to 4096 bytes, respectively. Although the choice for maximum burst size always holds for all channels, the amount of allocated 4 KB pages can be varied for each channel, however, it cannot exceed the maximum value shown in Table 6.3. We

Table 6.3.: Virtual FIFO DRAM space per channel according to burst size.

Burst Size [B]	Max. Pages [4KB]	Max. DRAM [MB]	Image Size
512	8192	32	2048×2048 (2B)
1024	16384	64	2048×2048 (4B)
2048	32768	128	4096×4096 (2B)
4096	65536	256	4096×4096 (4B)

can conclude, that in order to prepare the VFIFO to accept a full image of size up to 4096×4096 pixels, a maximum burst size of 2048 would be sufficient for 16 b greyscale images, and 4096 would be sufficient for single precision floating-point data. The specific channel, a data stream is written to, or read from is selected by a combination of the `tid` and `tdest` descriptors. Another option that can be varied per channel is the *AR Channel Weight Allocation*, which specifies the maximum number of outstanding read requests generated for the channel. In other words, the combination of maximum burst size and channel weight

specifies the amount of data generated during each arbitration. In consequence, the remaining space in the outward FIFO must be closely monitored to ensure a read burst from the DDR3 will not overflow the external FIFO. For example, if we use 2048 bytes as the value for the burst size at an AR channel weight of 4, one arbitration delivers $4 \times 2048 = 8192$ bytes of data per read burst. If we take the internal data width of 512 bits into account, the output FIFO should be able to at least accommodate 256 words of size 512 bits in order to prevent a data overflow from the VFIFO.

In addition, the VFIFO provides several *per-channel flags* of special interest, which are listed in Table 6.4. If the VFIFO asserts `vfifo_s2mm_channel_full`,

Table 6.4.: Virtual FIFO per-channel flags.

Signal	Direction	Description
`vfifo_mm2s_channel_full`	IN	External output FIFO is full
`vfifo_s2mm_channel_full`	OUT	Virtual FIFO channel is full
`vfifo_mm2s_channel_empty`	OUT	Virtual FIFO channel is empty

it can consume a maximum of eight more data words on that specific channel, which in turn means that any further data transfers to that channel should be inhibited. This can be achieved in combination with the preceding AXI4S switchi, which can inhibit the arbitration to a slave channel, if the corresponding signal is asserted. However, there is still the possiblity that data streams arriving on a different slave port are routed to the master port. This becomes especially crucial if the switch is used in conjunction with the VFIFO DDR3 abstraction layer.

A Flexible Wrapper Design for AXI4S Interconnects

A key concept of using the AXI4S interconnect for the internal wiring between an accelerator IP core and other SoC resources implemented on an FPGA and its environment is that it can be easily adopted on a high abstraction level to cope with changing design properties. However, for the correct functioning of the design component, it is necessary to assure that the control signals for the VFIFO are set correctly for each channel. The interface is kept compact by bundling AXI4S write and read ports, which requires the designer to only change the channel parameters in the `generic` section of the `entity` in case that the component must be adopted. The available parameters are the number of channels, as well as the input and output data widths of the channels. Moreover, it must be determined whether the output FIFO of a channel can still accommodate at least two consecutive read bursts from the VFIFO. To properly synthesize logic for this task, we pass the VFIFO burst size, as well as the individual

channel weights and the output FIFO size as generic parameters to the entity. The actual task is then performed by logic inferred by a `generate` statement, as shown in the corresponding VHDL Listing 6.8.

Listing 6.8: Excerpt of the architecture description of the AXI4S wrapper in VHDL.

```
1  GEN_LOGIC : for i in NUM_PORTS-1 downto 0 generate
2  begin
3  ...
4  s_fifo_space(i) <= CH0_FIFO_SIZE - (BURST_SIZE*CH0_WEIGHT)/32 when i=0 else
5                     CH1_FIFO_SIZE - (BURST_SIZE*CH1_WEIGHT)/32 when i=1 else
6                    (others => '0');
7  ...
8  -- determine FIFO full
9  process(s_axi_clk)is
10 begin
11   if(rising_edge(s_axi_clk))then
12     if(s_axi_rst = '1')then
13       s_vfifo_out_full(i) <= '0';
14     else
15       if(s_cnt_fifo_wr(i) < s_fifo_space(i))then
16         s_vfifo_out_full(i) <= '0';
17       else
18         s_vfifo_out_full(i) <= '1';
19       end if;
20     end if;
21   end if;
22 end process;
23 ...
24 end generate;
```

Further tasks, performed by this VHDL wrapper are the generation of the reset signal for the AXI interconnect, which keeps it in the reset state as long as either an external reset is asserted or the DDR3 interface has not finished calibrating. For proper functioning of the design, it is helpful to be able to determine, whether there is still data stored in the VFIFO. Due to the latency between initiating a write to the AXI4S interconnect and the empty flag of the VFIFO being deasserted, we generate an additional flag that compares the two events supported by a timeout counter.

6.4. Case Study: Intermediate Stream Buffering for Multiresolution Analysis

In scientific computing, *multigrid methods* [ST82] are a popular choice for the solution of large systems of linear equations that may stem from the discretization of Partial Differential Equations (PDEs) [GL91]. One of the most researched PDEs is Poisson's equation (refer to [Du+13]) which is used for modeling diffusion processes, e. g., in the simulation of temperature distributions [Fre96].

The V-cycle, one variant of a multigrid method, is shown in Algorithm 6.1. In the pre- and post-smoothing steps, high-frequency components of the error are damped by smoothers such as the *Jacobi* or the *Gauss-Seidel* methods [Van94]. In the algorithm, ν_1 and $\nu_2 \in \mathbb{N}$ denote the number of applied smoothing steps. Low-frequency components are transformed into high-frequency components by restricting them to a coarser level, thus making them good targets for the smoother once more.

On the coarsest level, direct solving of the remaining linear system of equations is possible due to its low number of unknowns. However, it is also possible to apply a number of smoother iterations. In the case of a single unknown, one smoother iteration corresponds to directly solving the problem.

Algorithm 6.1: Recursive V-cycle to solve the PDE $A_h u_h = f_h$
$u_h^{(k+1)} = V_h\left(u_h^{(k)}, A_h, f_h, \nu_1, \nu_2\right)$.

if *coarsest level* **then**
  solve $A_h u_h = f_h$ exactly else by smoothing iterations
else
  $\bar{u}_h^{(k)} = \mathcal{S}_h^{\nu_1}\left(u_h^{(k)}, A_h, f_h\right)$ {pre-smoothing}
  $r_h = f_h - A_h \bar{u}_h^{(k)}$ {compute residual}
  $r_H = R r_h$ {restrict residual}
  $e_H = V_H\left(0, A_H, r_H, \nu_1, \nu_2\right)$ {recursion}
  $e_h = P e_H$ {interpolate error}
  $\tilde{u}_h^{(k)} = \bar{u}_h^{(k)} + e_h$ {coarse grid correction}
  $u_h^{(k+1)} = \mathcal{S}_h^{\nu_2}\left(\tilde{u}_h^{(k)}, A_h, f_h\right)$ {post-smoothing}
end

Figure 6.15 depicts the structure of a multigrid solver to obtain a solution to finite differences (FD) discretization of Poisson's equation with Dirichlet boundary conditions, as proposed in [Sch+15b*]. Obviously, the structure is very similar to our solution for Multiresolution Analysis (MRA) in image processing, detailed in Chapter 4. Multigrid methods in scientific computing typically use very large grids, for example and require many levels of downsampling to achieve small enough grids on the coarsest level to approximate the solution. The solution must also be processed by multiple smoother iterations, yielding a very long delay on the data path which causes a problem for buffering of intermediate data streams on the individual levels.

We have hand written the code to synthesize the structure shown in Figure 6.15 with Vivado HLS using the library constructs presented in Chapter 4. The resulting structural representation in VHDL was implemented and evaluated on a Xilinx Kintex 7 FPGA, as described in Section 4.6.1, using a grid size of

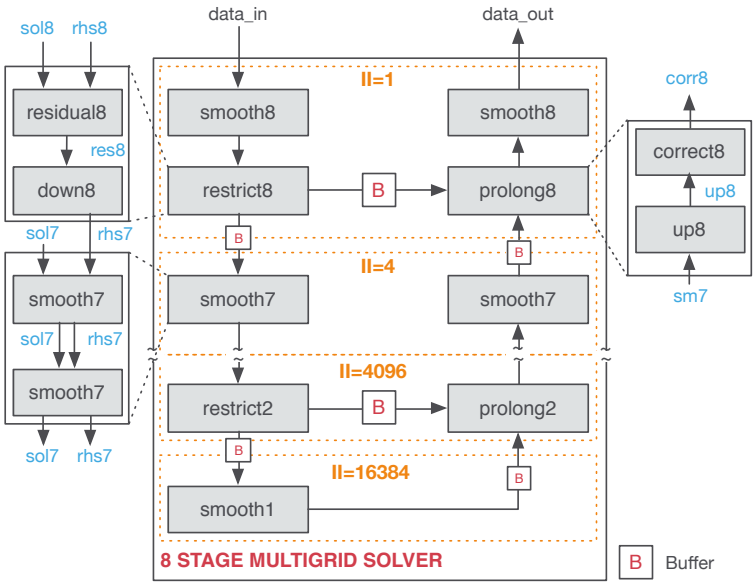


Figure 6.15.: Structural representation of a multigrid algorithm implementation to solve a PDE.

4096×4096 floating-point values. We were first of all interested in the necessary buffer sizes, which are listed in Table 6.5. Clearly, the buffers on the top four levels become very large and would overwhelm the amount of available memory resources, even on very large FPGAs. A solution to allow the facilitate the implementation and keep within the maximum amount of available resources is to offload the most challenging buffers to external DDR3 memory. We add input and output arguments for the streams to be externalized to the function definition and specify their type as AXI4S. In this way, we obtain a high-performance interface to the FPGA fabric for each data connection and do not need to make any further modifications to the actual accelerator source code. The FPGA support design uses an AXI4S interconnect built on top of the VFIFO as an abstraction to an off-chip DDR3 memory. A structural overview of the design

Table 6.5.: Buffer sizes for interconnecting the stages of the V-cycle.

Level	RHS buffer	Result buffer	4KB Pages
Stage 8	4430946	4430936	4328
Stage 7	1094407	1094397	1069
Stage 6	266942	266932	261
Stage 5	63728	63718	63
Stage 4	14443	14435	NA
Stage 3	2866	2856	NA
Stage 2	338	328	NA
Stage 1	1	64	NA

is shown in Figure 6.16. For our application, we use a 64 byte word width, as

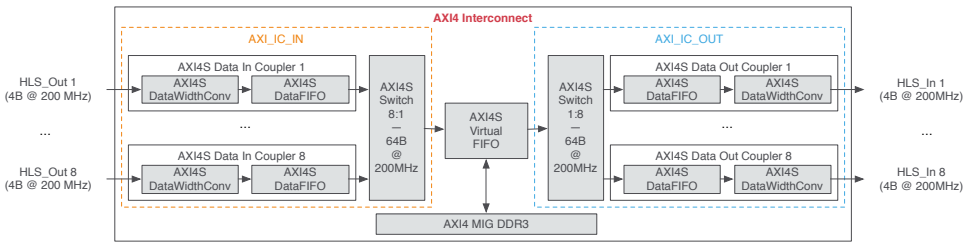


Figure 6.16.: Structural representation of the FPGA support design.

this is also the word width supported by the underlying DDR3 memory and select the smallest available burst size of 128 bytes. As the individual channels have different data production rates, we assign different weights for appropriate bandwidth allocation. To allow uninterrupted data exchange between the DDR3 and the accelerator, the interconnect aggregates data to the internal word width of the interconnect before passing it to the VFIFO.

Table 6.6 shows evaluation results of the hardware synthesis from Vivado HLS, which give a rough estimation of the resource requirements of the design. Indeed, after externalizing the top four largest buffers for results and Right-Hand

Table 6.6.: HLS resource estimates for the multigrid solver design comparing on-chip and external buffering.

Resource	On-Chip Buffering (%)	External Buffering (%)	Available
FF	256368 (62.9)	106311 (26.1)	407600
LUT	880314 (431.9)	177379 (87.0)	203800
BRAM	11812 (2654.4)	323 (72.6)	445
DSP48	442 (52.6)	442 (52.6)	840

Side (RHS), the design can be fit onto the chip. Table 6.7 lists the Post Place and Route (PPnR) results after integrating the modified accelerator into the described FPGA support system and shows that the multigrid solver can be implemented on a Kintex 7 with a maximum clock frequency of over 200 MHz. We

Table 6.7.: PPnR resource requirements of the complete multigrid solver design.

LUT	FFs	DSPs	BRAMs	Slices	F [MHz]
105951	135442	460	404	39147	202.34

have evaluated the PPnR hardware results on the Kintex 7 FPGA to measure the performance in terms of how many clock cycles it takes to process a 4096×4096

grid of floating-point values. In combination with the clock frequency of the design, this yields an accurate measurement of the performance. In contrast to a software solution, the pipelining principle also applies here, thus, it is not necessary to wait until the result is ready, but we can start processing a new grid, as soon as all of the input values of the previous grid have been consumed.

In order to compare the hardware accelerator to state-of-the-art software implementations, we have also implemented the multigrid solver for a single core of an Intel i7-3770, which is clocked at 3.40 GHz and features L2 and L3 cache sizes of 1 MB, respectively 8 MB. In addition, we have enabled AVX vectorization[10] for the Central Processing Unit (CPU). Table 6.8 lists the performance results in terms of latency in milliseconds for processing a single iteration of the V-cycle and the throughput in terms of how many full V-cycles can be processed per second ([Vps]), on average. It is also worth mentioning that

Table 6.8.: Comparison of the performance of the multigrid solver on different hardware targets.

Target	Latency [ms]	Throughput [Vps]
Kintex 7	83.1	12.3
Intel i7	223.1	4.5

it is irrelevant to the performance of the accelerator, whether the input data uses only single-precision floating-point or is implemented for double-precision input data. Although the chosen mid-range Kintex 7 FPGA cannot provide the necessary amount of logic and memory resources, switching to a larger FPGA, such as a member of the Virtex 7 family, here an XC7VX485t, can easily solve this issue. To underline this, we have also implemented the multigrid solver using double precision floating-point arithmetic, for which the estimated resource requirements are listed in Table 6.9.

Table 6.9.: HLS synthesis estimates for implementation of the multigrid solver using double precision arithmetic. Values are given as percentage of available resource type.

FPGA	LUT	FFs	DSPs	BRAMs	F [MHz]
Kintex 7	140	43	111	124	232.0
Virtex 7	73	29	33	53	229.4

[10]Refer to `https://software.intel.com/en-us/blogs/2013/avx-512-instructions`.

6.5. Summary

In this chapter we have explained how FPGA accelerators synthesize using HLS can be connected to essential components of the system architecture of an FPGA-based SoC design. We have first of all reviewed the interface generation opportunities of Vivado HLS and explained how custom protocols to interface non-standardized components can be implemented. Such communication is often suitable for exchanging control and meta data between the accelerator and other SoC components, such as an embedded processor or communication interfaces. Furthermore, we have introduced the AMBA AXI4 bus infrastructure, which we use to design a very flexible and high-throughput OCI for streaming and memory-mapped data communication between individual modules of the SoC architecture. We have concluded the chapter with a case study in which we have accelerated the solving of a PDE using a multigrid algorithm synthesized as an accelerator for an FPGA-based SoC design using HLS. Here the amount of memory required by the accelerator overwhelms the memory resources available on the allocated FPGA by far. Such memory requirements might be mapped onto an off-chip memory resource, for example, DDR3 memory at the system-level. To automate the system-level synthesis, an OCI as the here proposed AXI4S interconnect, can be allocated and the communication between the accelerator module on the FPGA and the off-chip memory can be bound to it automatically.

7

Board to Host Communication

Being able to initially *validate* and *verify* a newly developed design component using software has become a very important aspect of Electronic Design Automation (EDA). Our proposed methodology for prototyping of Field Programmable Gate Array (FPGA)-based image processing accelerators therefore includes the evaluation of Intellectual Property (IP) cores in a software-controlled environment. The goal hereby is to provide a hardware/software framework, in which a synthesized accelerator IP core is implemented on an actual FPGA and can be evaluated using the same software-based test bench as the reference software implementation. An important part to achieve this is to enable also the data communication of the generated accelerator IP core to its off-chip environment. Due to the high availability and relatively low hardware cost, Peripheral Component Interconnect Express (PCIe) has become a widely accepted standard for communication between a host processor and interconnected peripherals. It is therefore used in this chapter to establish the communication between the FPGA-based System-on-a-Chip (SoC) design and a software application that is running on a host computer.

The contributions of this chapter can be summarized as follows:

- An architecture for coupling an FPGA-based accelerator to a host computer via PCIe. Here we employ the AXI4S-based On-Chip Interconnect (OCI), as presented in Chapter 6, to establish the communication between an image processing IP core and a PCIe communication controller as part of an FPGA-based SoC design.

- A hardware/software co-design for the exchange of data and results between a software program and an accelerator using PCIe for communication. Moreover, a Linux-based software interface and driver is developed for this purpose.

7.1. The PCI Express Bus Architecture

PCIe has become the de facto communication standard for processor system peripherals and is available in almost all current computers, from consumer laptops to enterprise servers. This circumstance makes PCIe a very attractive interconnect due to its wide-spread availability and relatively low hardware cost. In this section, we first provide a high-level introduction to PCIe and its support on FPGAs. We then continue with the presentation of an architectural design implementation to include PCIe as a communication interface in an FPGA-based SoC system architecture. Furthermore, we detail a hardware/software solution to facilitate rapid prototyping of image processing IP cores in a host-based scenario.

7.1.1. Introduction to PCI Express

PCIe is the third generation high-performance Input/Output (I/O) bus to interconnect peripheral devices to a host-processor sub-system. The most current version of PCIe is version 3.0 (Gen3), however, over the course of this work we have only used PCIe in version 2.0 (Gen2). Therefore, the rest of this section provides and introduction to the specification for PCIe Gen2.

In analogy to its legacy predecessors, PCI and PCI-X, PCIe is often referred to as a *bus*, however, it is far from that but a high-speed serial, packet switched, point-to-point interconnect instead. In contrast to a parallel bus interface, serial links allow a reduction of the electrical load, higher interconnect frequencies, and lower pin count. PCIe supports full duplex communication between two endpoints over links, which may consist of multiple serial lanes. A so-called *root complex* interconnects the processor, the memory, and the PCIe fabric (refer to Figure 7.1. Every device is connected to the root complex either via direct link or a switch. The lanes use Low Voltage Differential Signaling (LVDS) in combination with 8b10b coding (refer to Section 8.1.4). Devices connected to a root complex are organized in the so-called PCIe tree, which is established by the root complex at system start up during a process called *enumeration*. Hereby, the root complex identifies existing links as well as devices. All devices within the PCIe tree share the same memory space, whereby the root complex determines the required address space and sets the Base Address Register (BAR) of each device. Furthermore, the root complex allocates bus numbers and configures the switches.

The packet-based protocol of PCIe is structured in layers, where the top layer is called the Transaction Layer (TAL). Data communication on the TAL is organized in transactions to provide support for read and write from and to *memory*, access to the *configuration space* of connected components to discover their capabilities, as well as signal the occurrence of events through *messages*. PCIe was designed with backwards compatibility to its predecessor protocols in

mind, which are supported through so-called I/O transactions. A transaction is issued by a *requester* and received by a *completer*. Read transactions require a response, which is called *completion*. In the terminology of PCIe, such transactions are referred to as *non-posted* transactions. In contrast, *write* transactions are called *posted* transactions, as they do not require a completion. Routing of Transaction Layer Packets (TLPs) is address-based for memory and I/O transactions. Completions and configuration space access are routed according to the device identifier.

Directly below the TAL is the Data Link Layer (DLL), which is responsible for *reliable packet transfer* between two end points and *flow control*. Reliable packet transfer in PCIe, as well as in many other protocols, is implemented on the DLL and therefore only supports point-to-point transfers. Packets exchanged between two link partners are acknowledged by the receiver, if received correctly, or the issuer of the packet must be notified if the packet is corrupted, in which case the packet is resent. Furthermore, the DLL implements the credit based flow control mechanism, which is similar to token bucket flow control. At link establishment, link partners exchange initial credits for transfer of header and data packets of posted, non-posted and completion transactions. It is possible to advertise infinite credits for any of those transactions, and end points must announce infinite credits for completions. One credit accounts for four double words, which is the maximum length of a header. Credits are depleted by transmissions and automatically replenished over time. Link partners regularly update connected end points and switches with their current credit information.

The lowest layer is the Physical Layer (PHY), which is split into two parts, the *logical* and the *electrical* PHY. The logical layer performs packet processing between the physical link and the DLL, and the electrical layer consists of differential drivers and receivers for Physical Media Attachment (PMA).

In addition, PCIe has several other features, such as dynamic power and link speed management, hot-swappable devices, and the ability to handle peer-to-peer data transfers between end points without routing the data through the host [BAS03].

7.1.2. FPGA Support for PCI Express

As FPGAs have become a very important accelerator technology, the connection to processor hosts is a vital application area. A possible topology to include FPGAs in a PCIe device structure is given in Figure 7.1. To simplify the design task, most modern FPGAs contain hard IP blocks to support PCIe, such as the Integrated Block for PCI Express from Xilinx [Xil14a] or the hardened IP block for PCIe from Altera [Alt15c]. Both solutions follow a common structure, as illustrated in Figure 7.2, where the three protocol layers are implemented as a hard IP block, building upon serial high-speed transceivers. Also, both

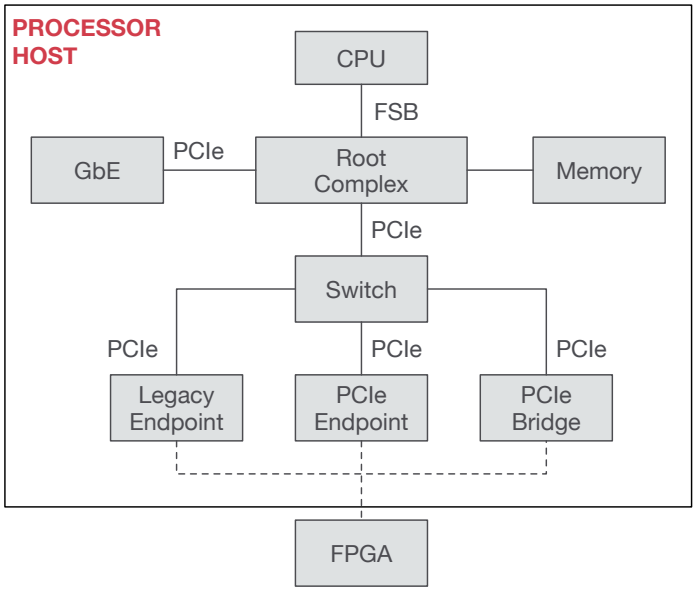


Figure 7.1.: General overview of the PCIe topology to include an FPGA in a processor host scenario.

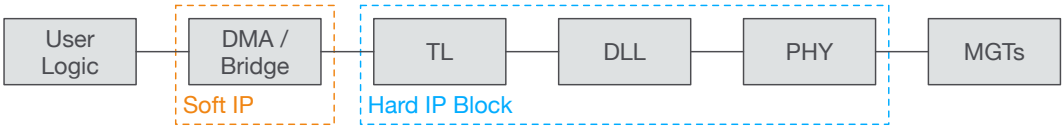


Figure 7.2.: Structural overview of FPGA support of PCIe endpoints.

companies provide optional soft IP for the implementation of bridging logic or Direct Memory Access (DMA) engines, which serve as entry point for user application logic. In this work, we have focused on the Xilinx Kintex 7 FPGA for which Xilinx provides a soft IP for the implementation of a DMA engine, which is responsible for managing the data transfer between the FPGA as a PCIe endpoint and the host processor system. In addition to the hardware implementation on the FPGA it is also necessary to implement a counterpart in software on the host in form of a device driver. In the following, we provide details on the complete hardware/software design to facilitate communication between the FPGA accelerator and the host system.

7.2. Host Coupling of FPGA-based Accelerators

The general approach is a combined hardware/software evaluation environment, where we use an FPGA accelerator card as peripheral to a host-computer system. In the following, we present a solution where an FPGA is connected to the host processor via PCIe. To establish the communication, it is essential to implement a PCIe endpoint and corresponding DMA support on the FPGA, as well as enable the processor to communicate with the PCIe peripheral through a device driver. A general system overview is given in Figure 7.3. The FPGA part

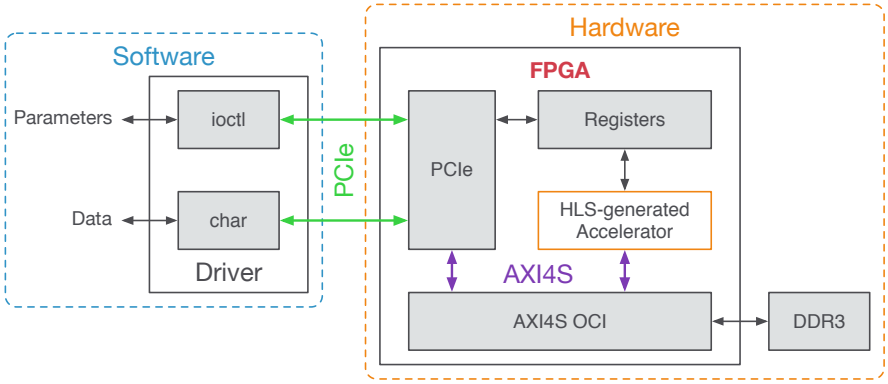


Figure 7.3.: General overview of the proposed host-based prototyping system.

makes use of an AXI4-Stream (AXI4S) interconnect, as discussed in Chapter 6, to establish communication between the PCIe endpoint and the High-Level Synthesis (HLS)-generated image processing accelerator. On the software side, we provide a device driver that enables bulk transferral of image data between the host and the FPGA card using a *character* device driver. Such a driver provides an easy interface, i. e., *write* and *read* commands, that can be included in a wide range of test applications. To be able to monitor the FPGA card even during a data transfer, design parameters and monitoring information can be exchanged using an *I/O Control* (ioctl) driver.

As we push data from the host memory via PCIe onto the FPGA peripheral, the data transfer is subject to operating system scheduling and cannot be influenced directly. The transmission speed of the interconnect might hereby be substantially higher than the processing speed of the accelerator, which might lead to back pressure if connected to the PCIe interface directly. To decouple the PCIe endpoint from the user logic, we use the AXI4S interconnect to buffer incoming data on a peripheral Double Data Rate Type Three (DDR3) memory before passing it on to the accelerator. The DDR3 might moreover be beneficial for Multiresolution Analysis (MRA) accelerators and multi-image algorithms.

7.2.1. PCI Express and DMA

To efficiently use the processor bandwidth, a bus-mastering *scatter-gather* Direct Memory Access (DMA) controller is used to communicate with the system memory. All data from and to the system is stored in DDR3 memory through a multi port virtual FIFO abstraction layer embedded in the AXI4S interconnect. The PCIe endpoint design uses the integrated IP block on the Kintex 7, which is compliant with the PCIe v2.1 specification. Although the core supports multiple lane and data rate configurations, this design uses the PCIe Gen2 configuration, which specifies the use of 8 serial lanes at 5Gb/s each.

DMA capability is facilitated by a scatter-gather DMA IP core. The term scatter-gather refers to the ability to write packet data segments into different memory locations and gather data segments from different memory locations to build a packet. This allows for an efficient usage of available memory resources, since a packet does not need to be stored in physically contiguous locations. Scatter-gather requires a common memory resident data structure that holds the list of DMA operations to be performed, which are organized as a linked list of buffer descriptors, i. e., a circular buffer. The buffer descriptors allow chaining, so that a packet can be described by more than one buffer descriptor. The DMA can support simultaneous operation of up to two user applications utilizing four channels, two receive System-to-Card (S2C) and two transmit Card-to-System (C2S) channels, respectively. The DMA requires a 64KB register space mapped to `BAR0`. Thus, all DMA registers are mapped to `BAR0` from `0x000` to `0x7fff`. This leaves the address range from `0x8000` to `0xffff` available for user applications. The front end of the DMA IP core interfaces to the Advanced eXtensible Interface 4 (AXI4) ports of the integrated PCIe block, whereas the back end of the DMA IP core provides an AXI4S interface, which connects to the AXI4S interconnect.

7.2.2. AXI4S Interconnect and DDR3 Abstraction

The here used AXI4S interconnect for the host-based system is depicted in Figure 7.4. It uses a dedicated receive channel, leading data from PCIe to the image processing accelerator, and a dedicated transmit channel, leading from the accelerator back to PCIe. Across the interconnect, the data width and clock rate are automatically adapted. Input couplers adopt incoming data streams to the internally used data width of 512 bits and clock rate of 200 MHz, before the streams are combined into a common data stream at the AXI4S switch. The input couplers consist of a data width converter followed by a data FIFO, where the latter performs clock rate conversion.

A sample stream, coming from PCIe, traversing the interconnect to the Vivado HLS component as target is shown in Figure 7.5. The PCIe protocol can deliver

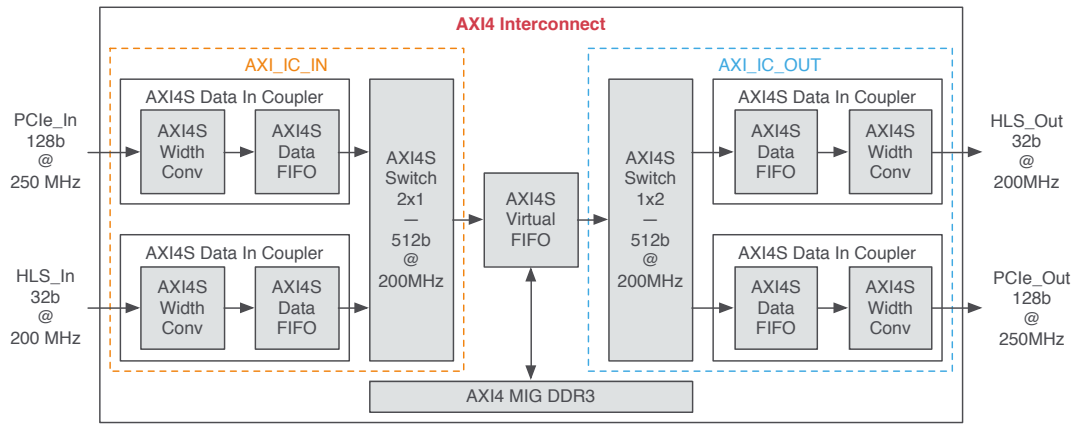


Figure 7.4.: General overview of the AXI4S interconnect between PCIe and the image processing accelerator.

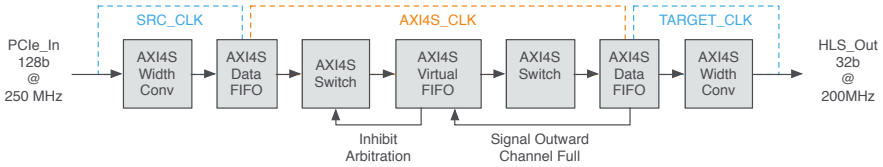


Figure 7.5.: Architecture and clock domains of on AXI4S interconnect channel.

128 bit data words at a clock rate of 250 MHz. Since the provided data rate likely exceeds the processing speed of the FPGA accelerator and due to PCIe being controlled on the host machine by an operating system, it is of benefit for the overall system performance to keep data communication independent of the actual processing speed of the accelerator on the FPGA. In order to adapt incoming data to be stored in the DDR3 memory, it is first necessary to increase the data width from 128 bits to 512 bits, which is performed by the AXI4S data width converter. We also adopt the data stream at 250 MHz to the internal clock speed of 200 MHz using an asynchronous First In, First Out (FIFO) memory. The FIFO is also employed to decouple incoming data from the scheduling interval of the proceeding components of the interconnect. On the outgoing path, data must be adopted to the target bit width of the accelerator and its clock rate. Although there might be no need to adapt the clock rate in this path, the FIFO is still essential to accommodate for DDR3 burst data transfers in the case that the target instance happens to not accept data.

7.2.3. Auxiliary Components

Apart from the so far discussed components, the host-based prototyping design also contain modules to implement a register space for the exchange of non-image data information between the FPGA-card and the host-system, as well as modules for monitoring the FPGA design.

Register Space

The DMA core provides the AXI4 target interface for the user space registers at the address offset between `0x8000` and `0x8fff` on `BAR0`. Transactions on these addresses are made available to the FPGA logic via the register space. In the current design, the register space is used to exchange the devices power and temperature monitoring information, the PCIe data rate measurements, as well as DUT configuration parameters.

PCIe Data Rate Measurement

Data about the performance of the PCIe interface is collected at the AXI4S 128 bit interface of the PCIe endpoint operating at 250 MHz. The performance values are derived from TX and RX byte counters, which count the active up- and downstream beats and therefore indicate the raw utilization of the PCIe transaction layer. The byte counts are truncated to a four-byte resolution and measured within one second. To reflect the performance correctly, the software should pull the register value every second. The two Least Significant Bits (LSBs) implement a two bit counter which can be checked to decide whether the current register transfer represents a new measurement to update the performance evaluation or should be discarded.

Power and Temperature Monitoring

Power and temperature of the Kintex 7 FPGA family are managed by a Texas Instruments power regulator that can be queried using the Power Management Bus (PMBus) interface, which is a proprietary extension of the I2C protocol. Since the 7-series FPGA family from Xilinx, the bus can be accessed directly from the chip to monitor the power consumption and temperature of the die. In this design we use a PicoBlaze based power monitoring logic that interfaces with the Xilinx Analog to Digital Converter (XADC) to read the temperature. Voltage and current values of different voltage rails in the FPGA are read by the PicoBlaze softcore microprocessor using the PMBus protocol and stored in a shared BRAM after the calculation of the actual values. The register interface interacts with the BRAM to provide the values to the software side of the application.

7.2.4. Clock and Reset Infrastructure

The design incorporates two different clock domains, a 250 MHz domain for PCI Express and a 200 MHz domain for DDR3. The clock and and reset infrastructure is depicted in Figure 7.6. The PCI Express design uses a pipe clock wrapper to

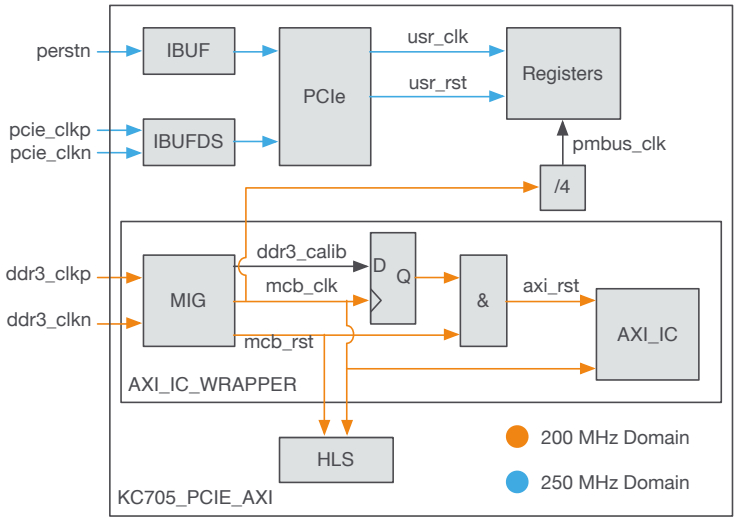


Figure 7.6.: Clock domains and reset structure.

adapt the PCI Express clock to the number of connected serial lanes and lane rate. As this design uses the eight lane and 5GB configuration, the PCIe clock always operates at 250MHz. The clock for the AXI interconnect, the Vivado HLS component, and the device monitoring is derived from the 200 MHZ clock generated by the DDR3 interface.

7.3. Software

For the software part of the host-based prototyping environment we provide a device driver to interface the FPGA accelerator card and a simple command-line application to exchange image data with the FPGA[11]. The here developed Linux kernel space driver provides access to user space applications through a character device driver for data exchange and through an `ioctl`-entry point for parameter exchange and device monitoring. The character device driver supports several of the standard user interface methods, including `open`, `close`, `write`, and `read`, however, the nature of the device does not support operations, such as `seek`.

[11] Xilinx provides a simple device driver as part of its FPGAs reference designs that mostly serves for demonstration purposes and is not intended to actually transfer any user data between the host computer and the FPGA

Listing 7.1 shows a simple program to open the device driver and exchange of data by using the `write` method to send and the `read` method to receive a certain amount of data to and from the device.

Listing 7.1: Example application for data exchange between the host and the FPGA.

```
1
2  int main(int argc, char **argv){
3  int fd = 0;
4  int buffsize = 1024 * 1024 * sizeof(int);
5  int *TX_BUF = malloc(buffsize);
6  int *RX_BUF = malloc(buffsize);
7  // open the device
8  fd = open("/dev/kc705", O_RDWR);
9  // write TX_BUF to the device
10 write(fd, TX_BUF, buffsize);
11 // read from the device into RX_BUF
12 read(fd, RX_BUF, buffsize);
13 // close the device
14 close(fd);
15 // free memory
16 free(buf);
17 free(ret);
18 }
```

When opening the device using the `open` method, the driver prepares descriptor rings, each containing a configurable number of descriptors for each DMA channel. Upon the write call, the character device driver directly maps the memory pages containing the user data from the user space into the kernel space and assigns these pages to buffer descriptors for mapping the data into the PCIe memory space. On the receiver side, buffers are pre-allocated to store incoming data upon calling the `read` method. After transfer completion of each buffer, the kernel space memory pages are directly mapped into the user space, where the data can be accessed after the specified amount of data has been transferred. When open, the driver can perform multiple data transfers or be closed by calling the `close` method.

Information from the register space on the FPGA can be accessed on Linux using the `ioctl`-interface. Listing 7.2 shows a short program to transfer the power and temperature measurements from the FPGA to the host system.

Listing 7.2: Example application for register exchange between the host and the FPGA.

```
1  #include <sys/ioctl.h>
2  #include "xpmon_be.h"
3
4  typedef struct {
5      int vcc;          // VCCINT Power     Consumption 0x9040
6      int vccaux;       // VCCAUX Power     Consumption 0x9044
7      int vcc3v3;       // VCC3V3 Power     Consumption 0x9048
8      int vadj;         // VADJ                         0x904C
9      int vcc2v5;       // VCC2V5 Power     Consumption 0x9050
10     int vcc1v5;       // VCC1V5 Power     Consumption 0x9054
11     int mgt_avcc;     // MGT_AVCC Power   Consumption 0x9058
```

```
12      int mgt_avtt;     // MGT_AVTT Power   Consumption 0x905C
13      int vccaux_io;    // VCCAUX_IO Power  Consumption 0x9060
14      int vccbram;      // VCCBRAM Power    Consumption 0x9064
15      int mgt_vccaux;   // MGT_VCCAUX Power Consumption 0x9068
16      int pwr_rsvd;     // RESERVED                     0x906C
17      int die_temp;     // DIE TEMPERATURE              0x9070
18  } PowerMonitorVal;
19
20  int main(){
21    int fd = 0;
22    PowerMonitorVal power;
23    fd = open("/dev/xdma_stat", O_RDWR);
24    ioctl(fd, IGET_PMVAL, &power);
25    close(fd);
26  }
```

The interface uses C structs to manage the information exchange.

7.4. Summary

We have started this chapter with a high-level introduction to PCIe, one of the most wide-spread peripheral communication protocols. We have continued by describing the basic architecture for PCIe endpoints on FPGAs and how they can be used in a SoC design to couple an FPGA-based accelerator to a host computer via an AXI4S OCI and advanced data exchange methods, such as DMA. Building on this architecture, we have developed a Hardware/Software Co-Design (HSCD) that in addition to high-throughput data transfer can also be used to monitor the state of the peripheral device using a register-based access. On the software side, we have developed a Linux-based software interface and a custom driver specifically for this purpose. In this way, a HLS-generated image processing IP core can be integrated into the proposed SoC design and evaluated using almost the same software test bench as for the initial verification of the reference software implementation.

8

Board to Board Communication

In this chapter we present a solution for board to board communication for distributed embedded systems, that is, facilitate data and information exchange in the absence of software control. Many new interconnect standards to overcome the problems imposed by legacy parallel bus architectures were proposed in recent years. Examples include RapidIO [Rap14], Hyper Transport [HTA08], Fibre Channel [Fib14], and Interlaken [GOW07], to only name a few. All of these provide scalable, high-throughput, packet-switched data transfer for distributed embedded systems. However, as shown in Figure 8.1, each of these protocols is targeted at a specific niche in the interconnect ecosystem. Although Field

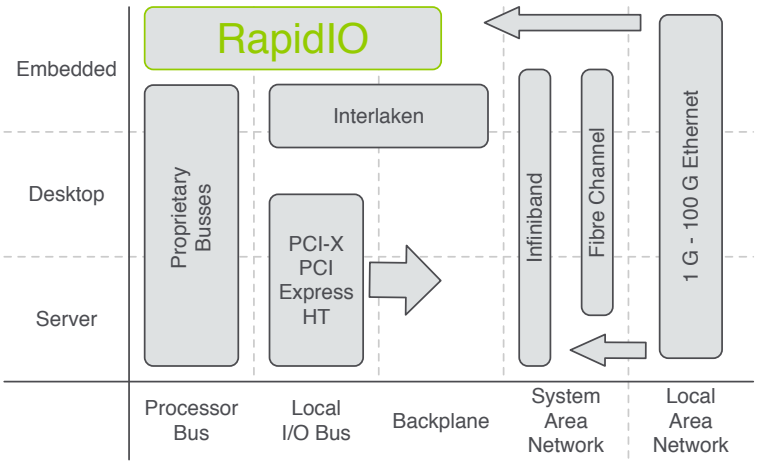


Figure 8.1.: Overview of popular interconnect standards and their specific niches.

Programmable Gate Arrays (FPGAs) have become very flexible in terms of supporting a wide range of Input/Output (I/O) interfaces due to the availability of high-speed serial Multi-Gigabit Transceivers (MGTs), the implementation of the above mentioned protocols is one of the most challenging aspects of FPGA design. Here, we use the RapidIO (RIO) protocol and provide details on how it can be used for data and information exchange between an image processing

accelerator in an FPGA-based System-on-a-Chip (SoC) design and peripheral components in a distributed embedded system.

The contributions of this chapter can be summarized as follows:

- A budget-based power management strategy is presented, that may significantly *reduce the power requirements* of Serial RapidIO (SRIO) interconnects for streaming-based applications with periodic data transmission characteristics, such as medical image processing for distributed embedded systems [SHT12*].
- An SRIO communication controller, augmented with a custom *user application* as front end, is developed. The user application autonomously generates packets for streaming-based application data and can therefore be integrated seamlessly into the AXI4S On-Chip Interconnect (OCI) for FPGA-base SoC designs.

8.1. The RapidIO Interconnect Standard

RIO is an open standard, developed to achieve high-performance, low-cost, reliable, and scalable system connectivity in embedded systems, networking applications, and communication devices. Its capability for advanced traffic management, built-in error-correction mechanisms, and provisioning for high performance and throughput have aided to position RIO as a dominant interconnect in the embedded systems market. Since RIO is not as widely recognized as Peripheral Component Interconnect Express (PCIe), this section provides a more thorough introduction to the protocol. The most recent version 3.1 of the specification was released in the second half of 2014. The work in this document is based on version 2.1 of the RIO specification [Rap09].

8.1.1. Technological Overview

The RIO standard focuses on on-board, inter-board, and back-plane connectivity. It can be implemented completely in hardware, which includes error handling and traffic management. The protocol is partitioned into a layered architecture, which is shown in Figure 8.2. RIO is based on the concept of request and response transactions. Transactions are encapsulated in packets, which are transported from the initiator to the target through the RIO network. The Physical Layer (PHY) is responsible for exchanging packets on a link-by-link basis. Data flow between RIO devices is managed by control symbols, which provide packet acknowledgement, flow control and maintenance functionality. The Transport Layer (TRA) on top of the PHY, is in charge of routing packets through the fabric from one endpoint to another, by traversing a number of

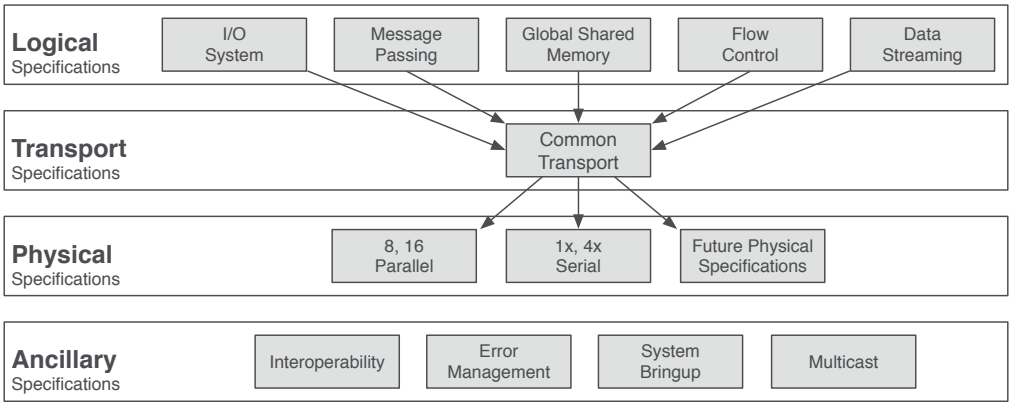


Figure 8.2.: Overview of the RIO specification hierarchy.

switches. At the top level of the specification, the Logical Layer (LOG) is in control of the transactions between the endpoints of the RIO network. An example of a RIO network is shown in Figure 8.3. Endpoints in RIO networks

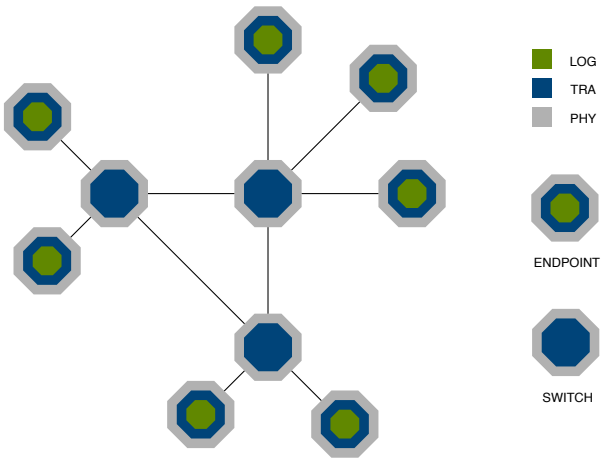


Figure 8.3.: Overview of a RIO fabric. The endpoints implement all three layers of the protocol, whereas the switches only need to implement the lower two layers.

are involved in the transaction concept and therefore have to implement all three layers of the protocol. The sole purpose of a switch is to route incoming packets to the next switch or endpoint on the route, for which it is not essential to be aware of the contents of the packet. Thus the logical layer is not needed and switches only implement the lower two layers.

To demonstrate how the transaction concept is put to work, a sample RIO transaction process is shown in Figure 8.4. Here, an initiator issues a request by transmitting the appropriate packet to the fabric. Correctly received packets

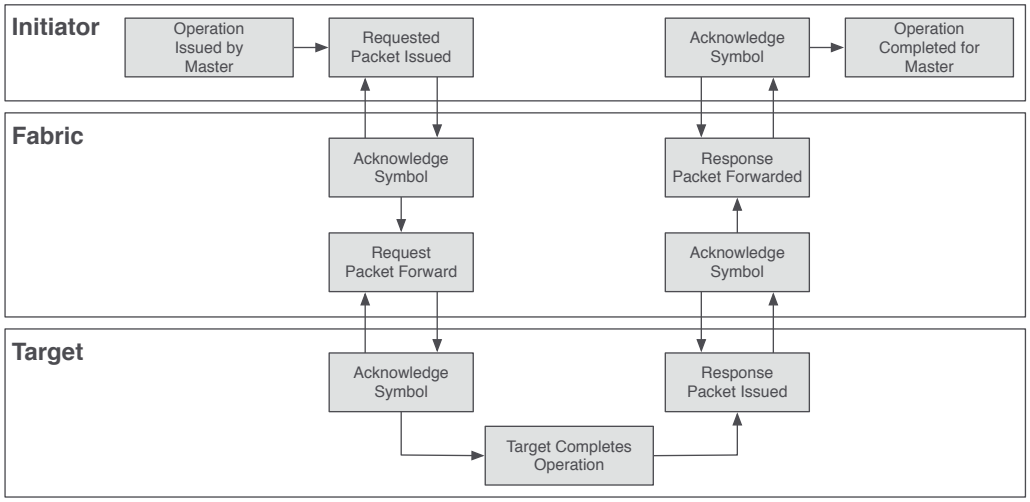


Figure 8.4.: Overview of a typical RIO transaction.

are acknowledged between directly linked devices. This approach was chosen, because of the initiator and the recipient of a transaction might be a long distance apart from each other, so that an end-to-end handshake would imply imprecise detection of errors and recovery from them. Once the packet has reached its destination, the target endpoint tries to complete the job contained in the transaction. If the transaction requires a response, the target endpoint will send the associated packet back to the initiator.

Each RIO packet carries a header, which provides information for switches and endpoints on how to handle the packet. Since each layer must take care of different tasks, each has its own header. Figure 8.5 shows an abstract example of how each layer encapsulates the Service Data Unit (SDU) passed on from above to create the layer's Protocol Data Unit (PDU) by adding its own header. On the receiving side, the layers remove their headers from the packet, and only pass on the information, that was encapsulated. The actual fields of each layer's header are covered in the proceeding sections, where appropriate.

The logical layer offers a wide variety of transaction types to support full transparency towards software. RIO supports Direct Memory Access (DMA) and message passing operations, where the chosen model depends on the memory structure of the system. As an extension, RIO can also provide globally shared memory capabilities by means of a directory-based memory coherence scheme, that involves atomic transactions.

Flow control is another very important feature of the interconnect and is used to prevent the network from becoming congested and unresponsive. In contrast to bus-based systems, where a central arbitrator can ensure progression of data, decentralized packet-switched networks have to implement rules, upon which it is possible for the nodes in the network to decide which transaction to handle

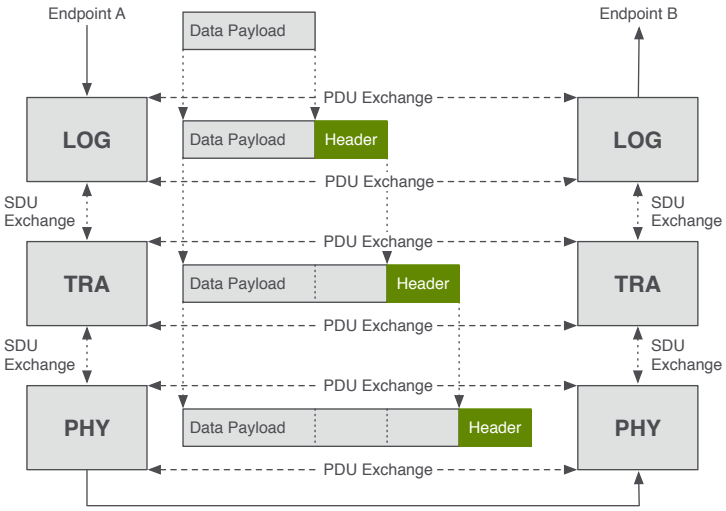


Figure 8.5.: Abstract view of the RIO layer header encapsulation.

next. RIO defines two flow control mechanisms. A link-based priority mechanism is specified on the physical layer. Transactions are assigned a priority, to be associated with one out of currently three request flows. Packets with higher priority in a certain request flow can get ahead of those with lower priorities. In this manner, it is possible for system designers to structure traffic in the system and model deterministic and isochronous behavior. The logical specification defines a second flow control mechanism on an end-to-end basis. In larger complex systems, controlling congestion between adjacent nodes might not solve the problem, which is why, RIO features a special congestion control packet. It can be used by either an endpoint or switch to shut off the source of congesting traffic for a period of time, which will decrease the overall amount of packets in the network.

RIO provides support for two different physical standards, which are implemented by the parallel and the serial physical layer. The parallel physical layer is known as the 8- or 16-bit link protocol endpoint specification with low-voltage differential signaling (8/16 LP-LVDS). It specifies a parallel, bidirectional communication of either 8 or 16 bits, including clock and frame signals. The source synchronous interface transports the clock along with the associated data, which allows for long transmission distances and frequencies up to 1 GHz. In contrast, the serial specification does not define a source synchronous interface, but rather has to be synchronized between link-partners by special frame signals. Clock information is encoded within the transmitted data, using the 8B/10B encoding scheme. The serial specification uses a Physical Coding Sublayer (PCS) and a Physical Media Attachment (PMA) sublayer to organize the packets into a serial bit stream at the transmitter, and de-serialize the bit stream at the receiver.

The parallel and serial layers both use Low Voltage Differential Signaling (LVDS) electrical interfaces, where the parallel specification follows Institute of Electrical and Electronics Engineers (IEEE)1596.3 [IEE96] and the serial specification follows the IEEE802.3XAUI [IEE15] standard.

Unalike legacy bus technologies, RIO provides *link maintenance and error management* mechanisms, which enable initial system *discovery* and *configuration*, as well as *error detection* and *recovery*. Each RIO device has a set of *capability* and *command status registers* (CAR/CSR), which specify the device's abilities and current state. By the use of *maintenance* operations, a RIO device may traverse the network and access these registers to discover and configure other devices. To avoid error situations, more than one host is allowed to configure the network. Because of RIO operates at very high frequencies, errors are likely to occur and a strong error detection and recovery mechanism is required. RIO relies on a *CCITT16* cyclic redundancy check to detect errors in packets and control symbols.

RIO offers several features to improve system performance. *Source routing* and packet *priority tagging* were implemented to reduce blocking of packets. Switch-based interconnect systems can either employ *multicasting* or source-routing to forward packets across the network. Although multicasting has the advantage of the initiator not having to know where to find the destination of the transaction, moderate traffic situations might already cause the network to become unresponsive. Source-directed routing uses the target address to route the packet on a specific path through the fabric, avoiding to burden other paths, than between the initiator and the target of a transaction, with traffic. RIO headers were carefully designed to not supply any more information than is absolutely necessary to complete the transaction. For most RIO transactions, the overhead will be around 28 bytes, including headers for request and response packets, as well as the control symbols for acknowledgement and CRC codes.

8.1.2. The Logical Layer

The logical layer of the RIO specifications provides basic support for reads and writes from and to memory. The actions necessary to perform these are encapsulated in transactions. As mentioned before, the RIO fabric does not track transactions, so only the endpoints have to implement the logical layer. Transactions are always initiated by a request made from one endpoint device to another. Depending on the type of the transaction, the receiving device might have to send a response back to the initiator.

Transactions are classified into requests and responses. They can be identified by the combination of format (FTYPE) and transaction type (TTYPE) header fields. Table 8.1 lists the defined FTYPE values for transactions.

Table 8.1.: Transaction format types for requests and responses. Not listed types are reserved for future specification.

FTYPE	Transactions	Description
0	Implementation defined	
2	NREAD	Non-coherent read from system memory
	ATOMIC	Test-and-swap operation, useful for multiprocessor semaphoring
5	NWRITE	Non-coherent write to system memory
	NWRITE_R	Non-coherent write to system memory with response
	ATOMIC	Read-modify-write operation, useful for multiprocessor semaphoring
6	SWRITE	Non-coherent write, optimized for large DMA transfers
8	MAINTENANCE	Transaction targeting RIO specific registers
13	RESPONSE	Multi-purpose response class
15	-	Implementation defined

To obtain a certain region of memory attached to another device, an endpoint can issue a read in form of the NREAD transaction. The requested amount of data can be between 1 and 256 bytes. Payload sizes above 8 bytes must be double-word (8 bytes) aligned and a multiple of 8 bytes in length. For packets, carrying a smaller payload, valid data bytes are determined by a byte mask.

Another important header field is the transaction identifier. The source assigns each transaction a currently unique ID, which can be used to match incoming responses with unanswered transactions. RIO supports three memory sizing models, where the smallest can address 4 GB. The other memory models are medium and large, which can address more than 16 GB in the first, and 32 GB in the latter case. The memory model to be used must be negotiated between endpoints at system bring-up.

A special case is the *streaming write* (SWRITE) transaction, which was specifically included for high-throughput data transfer across the RIO fabric. The header is reduced in length and the transaction does not require a response from its target. SWRITE transactions do not have data payload size information, instead, the target of the transaction must determine the length of the data itself by finding a packet delimiting control symbol. An SWRITE packet is properly delimited by either the *end-of-frame* (EOF) control symbol, or alternatively, the start of a new packet, indicated by the *start-of-frame* (SOF) control symbol.

Depending on the type of the request transaction, the target endpoint must return a response to inform the initiator of the outcome of the performed operation and deliver requested data. As it can be seen in Table 8.1. An exception is the MAINTENANCE transaction, which uses the packet for both, requests and responses. The kind of the response is identified by the transaction type field, which indicates whether the transaction carries data, and by the status field, giving information about the outcome of the transaction.

Table 8.2.: Transaction type encodings for responses. Not listed types are reserved for future specification.

TTYPE	Definition
0	RESPONSE, transaction without data payload
8	RESPONSE, transaction with data payload

Table 8.3.: Status encodings for responses. Not listed encodings are reserved for future specification.

Status	Sub-field	Definition
0	DONE	Requested transaction has been successful
8	ERROR	Unrecoverable error detected
12-15	-	Implementation specific

Values defined for the transaction type and the status header fields are shown in Tables 8.2 and 8.3, respectively. If the status is set to DONE, the transaction was completed successfully at the target, and, if the request was a read operation, the response will carry the expected data. In contrast, if an error occurred, the response never carries data, and the status is set to ERROR. The logical layer can recover from certain errors, such as uncompleted transactions that require a response. For each transaction with response, a timer is started. If either the request or the response packet is lost for any reason, a timeout will occur, and the transaction is restarted by reissuing the request. For transactions that might cause problems if they are carried out twice, such as certain atomic operations, the target must keep track of already completed transactions.

8.1.3. The Transport Layer

The RapidIO standard was crafted to support very large systems with hundreds or maybe thousands of clients. Because of it is impractical to unite such systems in a single address space, RIO nodes are uniquely identified using device identifiers

(IDs). There are two system models defined. The small model uses 8 bit device IDs and can support up to 256 individual endpoints. The large model uses 16 bit device IDs and can identify up to 65536 individual devices.

System Topology

RIO networks can support almost any topology. Since the device IDs do not contain any information where the endpoint might be located in the system, it is the responsibility of the fabric to maintain information on how to route packets to the target. RIO switches have to rely on the *discovery routines* at the system bring-up stage, to learn where to locate specific endpoints. To achieve this, RIO switches maintain a routing table, which provides information on where to forward incoming packets. Because of the device IDs are rather small, storage and search can be very efficiently implemented by hash-tables.

Packet Routing

Each directly reachable endpoint in the RIO network has at least one unique device ID, by which the packets are forwarded through the network. In its simplest form, this means that there will only be a single route between two arbitrary endpoints. More complex routing mechanism may be implemented to compensate for heavily congested routes, for example, by forcing packets onto alternative routes. Since most transactions in a RIO network only target endpoints, it is not required that switches can be explicitly targeted by device IDs. However, they must be accessible by maintenance transactions, which use a *hop count scheme* to indirectly target a specific switch on a route between two endpoints. If the hop count field is present in a packet, the switch can determine that the packet contains a maintenance transaction. If the value of the hop count field is zero, the switch performs the requested operation on its configuration registers and directs a response at the source address of the transaction. If the hop count value is different from zero, the switch decrements it, and forwards the packet to the next node using the target address. For maintenance transactions that are targeted at endpoints, it is good practice to set the hop count header field to its maximum value, so that it will not be intercepted by a switch on the route between the initiator and the target.

8.1.4. The Physical Layer

The serial physical layer, also known as SRIO, was designed to provide connectivity on a *board to board* basis or across the *backplane* of a system. It specifies full duplex serial links, using unidirectional differential signaling in each direction. Synchronizing communication partners is achieved by embedding

the clock in the data, using 8B/10B encoding. Depending on the availability of ports and required data rate for the system, the interface can use multiple lanes in each direction. Packets can be cross-transferred between the serial and the parallel specification without any packet modification. Depending on the clocking frequency and kind of the transmitter, the physical layer can support data rates between 1.25, and 6.25 gBd.

8B/10B Encoding

The 8B/10B encoding was originally developed by IBM [WF83] and is widely used to combine data and clock information for serial communication. The encoding scheme transforms the user data by projecting every incoming 8 bits onto a 10 bit symbol for transfer. The lower 5 bits of the data are encoded into a 6-bit group and the top 3 bits are encoded into a 4-bit group. Concatenated, they form the 10-bit data symbol, that is also referred to as $Dxx.y$, where xx ranges from 0 to 31, and y from 0 to 7. The encoding scheme also defines special symbols, referred to as $Kxx.y$. These can be sent instead of the data symbols for link control. K- and D-symbols are disjunct sets of symbols. For each of the possible data-symbols, 8B/10B provides two different encodings, by which the scheme can affect long-term DC-balance in the serial data-stream, allowing links to take advantage of capacitive coupling.

Data Specification in Memory and Communication

Memory storage and communication systems show an ambiguity in how data rates and data amounts are expressed. The *raw* data rate used in communication commonly describes the amount of *user data*, that can be transported by the network. This data rate is measured in *bits per second* (bps). However, it is often necessary to encode the user data to facilitate data transfer, which mostly involves adding redundancy or other information to the user data. In this case, the data rate is measured in transported *symbols per second*, which is also known as *baud*. When describing data in computer systems, it is very common to use 1024, or 2^{10}, as order of magnitude to, for example, express 1024 bits as 1 Kbit. In communications, the order of magnitude used is not 1024, but 1000. Throughout this document, we will clarify the ambiguity by using small letter for decimal, and capital letters for binary orders of magnitude. Further, to distinguish between bits and bytes, we will use a small b for *bits*, and a capital B for bytes. The RIO physical layer specifies a maximum data rate per lane of up to 6.25 gBd per second, which means that $6250 \cdot 10^6$ symbols can be transmitted per second. User data is encoded by the 8B/10B scheme, which means, that 20% of the transmitted symbols are *overhead*, so that the actual raw data rate is 5.0 gbit per second. We can conclude from this, that a single 6.25 gBd SRIO lane can

transport approximately 4.66 Gbit raw user data per second from the computing system's point of view. In the rest of this section, we will describe data rates from the communication system's point of view, unless specified differently, and use the following abbreviations:

gBd Giga baud, meaning 10^9 symbols per second

gbps Giga bit per second, meaning 10^9 bits per second

GBps Giga bytes per second, meaning 2^{30} bytes per second

Physical Layer Packets

The physical layer is the main point of interaction for the *data link protocol* to exchange packets between RIO endpoints and switches. It is responsible for acknowledging packet transfer, the *priority mechanism* and the computation of the error correction codes (ECCs). Furthermore, it handles the *control symbols*, which are message elements, exchanged by adjacent devices, to manage all aspects of link operation. There are two functions in a control symbol, so that very common combinations, such as the end of a packet and an acknowledgement, can be combined into one control symbol to increase efficiency. The 5 bit CRC at the end of the control symbol covers all the preceding 18 bits. These 5 bits will have to be checked and computed by receivers and transmitters for all control symbols, even *idle* sequences.

Structure of an SRIO Endpoint

In Figure 8.6, the structure of a typical SRIO endpoint is shown. The serial protocol layer at the top of the illustration is responsible for generation and insertion of the control symbols for link management and control. Directly below is the PCS. Between these two layers lies also the boundary for the logical and physical clock domains of the hardware. The PCS is responsible for `IDLE` sequence generation, *lane striping*, and 8B/10B encoding. Serial interfaces must be kept synchronized and aware of each other all the time. Synchronization and clock information is embedded in data, but also the `IDLE` sequence, which is constantly exchanged between adjacent interfaces, if there is no data to send. It is also the duty of the PCS to discover by how many lanes it is connected to the other interface. SRIO currently supports multi-lane connectivity of up to ten lanes (10x) operation. In case of 1x mode, lane striping is straight forward. However, if the port runs in multi-lane mode, data is striped to the lanes on a bit-by-bit basis, meaning that bits 1 to n are distributed to lanes 1 to n, then a wraparound occurs, so that bit $n + 1$ starts out at lane 1, and so on. The PMA layer, between the PCS and the electrical layer, is the actual interface for SRIO

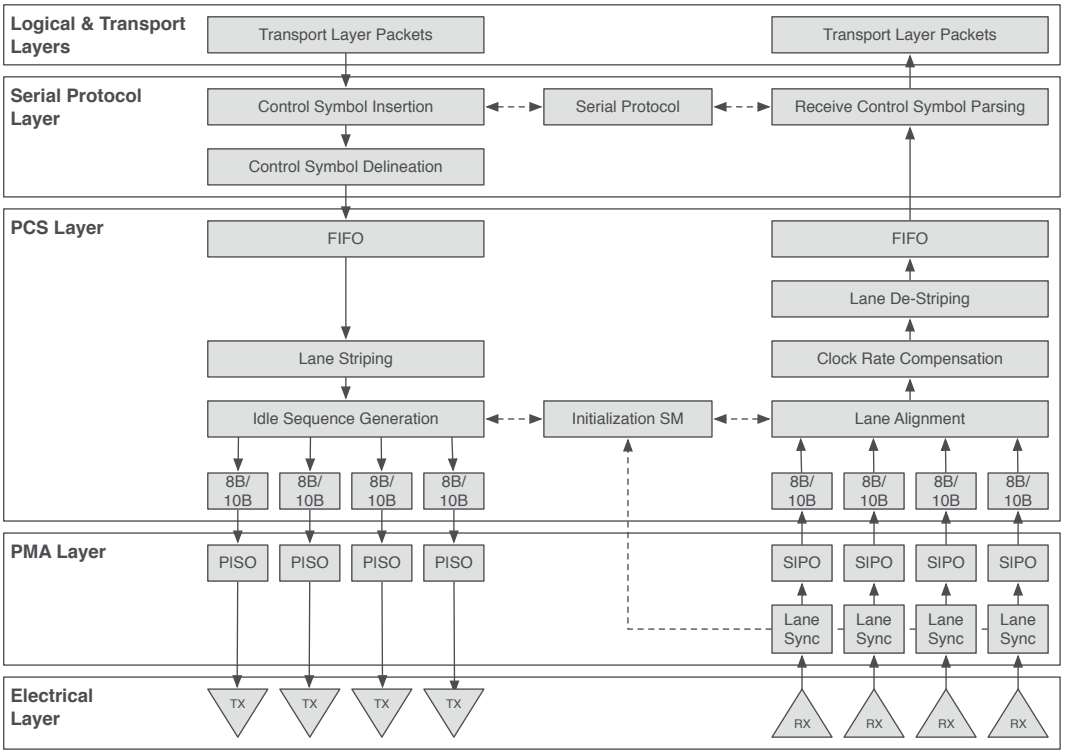


Figure 8.6.: Structure of a typical SRIO endpoint.

to the physical media. Its responsibilities include *alignment* and *serialization* of the parallel-bit code-group streams to the serial lanes. For the receivers, it is important to maintain lane alignment to ensure that there will not be too much skew between the lanes, so that the code-groups can be de-serialized correctly.

Port Initialization

The port initialization process is started by an SRIO device right after it was switched on, and is used to synchronize communication between link partners. First of all, the device must detect whether it is connected to a link partner and how many lanes there are used for this connection. If a link partner is present, the communication interfaces must be synchronized, which involves bit synchronization by code-group boundary alignment. Once the ports have been synchronized, the devices have to negotiate various parameters, such as which memory model and addressing scheme is used. The overall process is controlled by a primary state machine, which manages sub-automata for the individual tasks.

Packet Exchange

Packets can be exchanged, once the ports between adjacent link partners have been synchronized. Sending packets follows the link protocol. As mentioned before, control symbols are the key element to the link protocol and management. They are used for delimiting and acknowledging packets, as well as link management, error reporting and recovery. The control symbols are always fixed in length, so they need not be delimited. To increase efficiency and limit management jitter, control symbols can be embedded in packets, as long, as they do not delimit it inappropriately. In contrast to control symbols, packets are variable in length and must be delimited by control symbols. In normal operation, a packet is delimited by `EOF`, `SOF`, or any link-request control symbol. In case of errors it may be necessary to cancel partially transmitted packets by either the `STOMP` or the *restart-from-retry* control symbol. The link protocol requires the receiver to acknowledge each accepted packet. This is done by sending an acknowledge control symbol, that contains the acknowledge ID of the packet to be confirmed. In return, the sender must keep a copy of each packet that has not been acknowledged, yet, for the case, that an error has occurred and the packet must be resent.

Flow Control

RIO implements *link-level flow control* as a part of the physical layer specification. Its purpose is to prevent packet loss due to insufficient buffer space in the receiver. The RIO specification defines two varieties of flow control on the physical layer, *receiver* and *transmitter* controlled. Although the implementation of the transmitter controlled protocol is optional, receiver controlled flow control is mandatory for every device. In the receiver controlled version, the transmitter has no information about the available buffer space at its link partner, and simply transmits packets, until they are rejected by the receiver. The receiver starts to reject packets, if there is no more buffer space available for the packet's priority level. While the acceptance of a packet is signaled by the *acknowledge* control symbol (`ACK`), the rejection of a packet due to insufficient buffer space is indicated by the *packet-retry* control symbol, which contains the ID of the rejected packet. As soon as the transmitter receives this control symbol from its communication partner, it cancels the current transmission and restarts transmitting all the packets, starting with the rejected transaction ID. An overview of the procedure is shown in Figure 8.7. To avoid the problem of frequent retransmissions, transmitter managed flow control requires the receiving port to inform the sender about its available buffer size, which is shown in Figure 8.8. RIO implements this by including the information in acknowledge, retry and status control symbols. The sender may send more packets than there

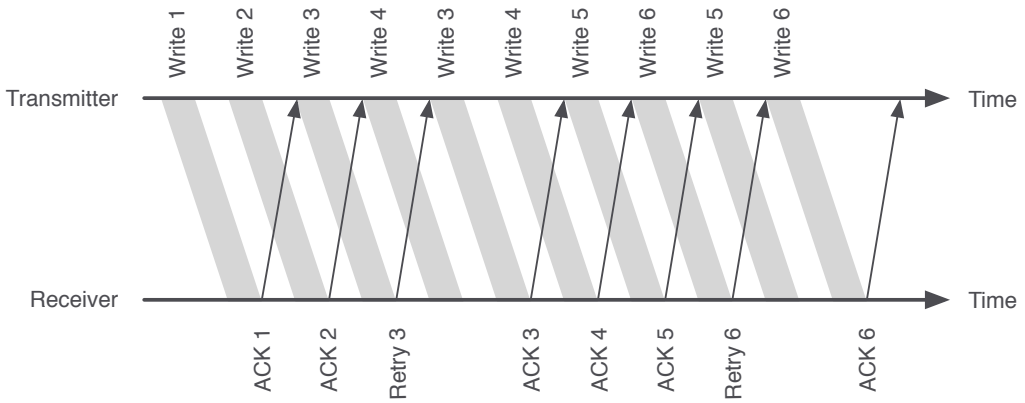


Figure 8.7.: Receiver managed flow control.

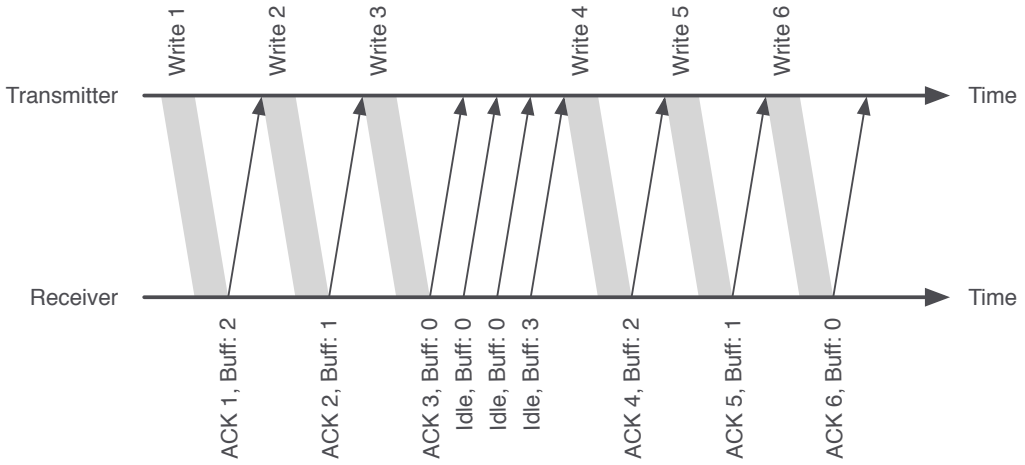


Figure 8.8.: Transmitter managed flow control.

is buffer space available on a speculative basis to maximize link utilization. The receiver then accepts or rejects a packet according to the actual buffer state. If the sender anticipated too much and exceeds the available buffer space, the protocol follows the exact same procedure, as if the flow was managed by the receiver.

Error Management

RapidIO was designed to provide highly reliable communication, which also includes the management of errors. On the physical layer, *recoverable* errors include *faulty* `IDLE` sequences and *damaged* or *lost* packets and control symbols. For every sent packet, the transmitter starts a timer to notice if a packet remains unacknowledged. Since the transmission times between adjacent link partners are relatively short, the timeout is not very long, compared to the

timers for transactions on the logical layer. The receiver is responsible to initiate the *recovery protocol*, if a packet or control symbol contains a detectable error. To start the process, the receiver sends an *error-stopped* control symbol back to the source of the faulty packet, and waits for a retransmission, while discarding all packets that do not match the expected acknowledge ID. When the transmitter receives an error-stopped control symbol, it cancels an eventually partial transmission, sends a *recovery-init* control symbol and retransmits all packets beginning with the acknowledge ID, that was contained in the error-stop control symbol.

Deadlock Avoidance

Due to the fact, that RIO can support almost any system topology, a network might become subject to *deadlocks*. A possible scenario for a deadlock is, if two communicating endpoints fill up their entire transaction buffers with requests, so that the corresponding responses cannot be transmitted any more. To prevent such situations, the RIO specification defines transaction and packet delivery ordering rules. Concerning transactions, the physical layer is required to transmit these in the exact order, as they are passed down from higher layers, except that transactions of a higher priority must be delivered before those of a lower priority. The same rule is in effect for received transactions, which must be passed on to higher layers, in the exact same order as they were received. Packets are not to be considered committed, before they were not acknowledged by the receiving link partner. The physical layer may not alter the priority of a packet, and must use a fixed route, according to the destination ID. Switches may not alter the order of packets of the same priority level and must forward packets of a higher priority before those of a lower priority. Besides the ordering rules, endpoints may not fill up their entire buffers with request and must set the priority of responses one level higher than the corresponding request.

8.1.5. Serial RapidIO on FPGAs

For use with their range of FPGAs, Xilinx offers two IP cores to implement an SRIO FPGA endpoint. The logical and transport layer, which comprise the logical layer core (`LOGIO`), are separated from the physical layer core (`PHY`). In this work, we have used the Intellectual Property (IP) cores in version 5.6, which implements version 2.1 of RIO specification [Xil11a] for a Xilinx Virtex-6 FPGA. Furthermore, we have used the updated SRIO Gen2 IP core in version 3.1.

The general structure has remained the same for both IP core versions and is illustrated in Figure 8.9.

The user interface is incorporated into the *LOGIO* IP core and provides access to data communication, as well as the SRIO configuration space. The data

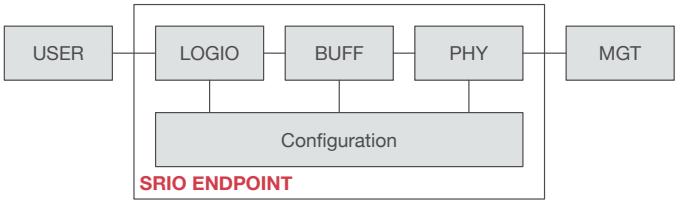


Figure 8.9.: Structural representation of the Xilinx SRIO IP core.

I/O interface allows to start request transactions and respond to transactions. Whereas the IP core in version 5.6 provided individual interfaces for initiation and response, the Gen2 IP core also offers to use a condensed interface. The user application is responsible for initiating requests, as well as process incoming requests and generate responses. The configuration space implements the endpoint's capability and status registers, which can also be accessed from other devices in the SRIO network. The serial transmission is implemented using Xilinx GTX MGTs.

8.2. Power Management Strategies for SRIO

Although SRIO is specifically targeted at embedded systems, the protocol does not consider power-efficiency. The modular structure on the FPGA, however, allows the implementation of custom means for power management, which are discussed in this section. In Figure 8.10, we depict the communication channel utilization over time in a fixed bandwidth, periodic communication scenario, where the data rate requirements are lower than the actual available data rate.

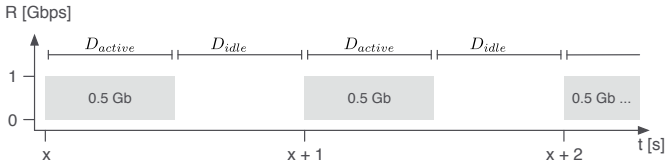


Figure 8.10.: Channel activity at a transmission rate of 1 Gbps at an available data rate of 1 Gbps and 0.5 Gbit of data.

The channel alternates between active periods (D_{active}) of data transmission and idle periods (D_{idle}), where the network adapter does not transmit any user data. Although there is no payload being transmitted during the idle periods, the link is highly active to keep the communication partners synchronized. Hence, such idle periods are prime candidates for optimization of power consumption in serial interconnects. If the link is deactivated during idle periods and reactivated for transmission or reception of new data, the link is not ready right away, but

must resynchronize before data can be transmitted. This link synchronization is referred to as the link training delay (D_{lt}) and, unfortunately, may take up between 100 and 250 ns, depending on the line rate of the interface.

Several possibilities exist for power-management of idle-periods. A straightforward means of reducing the power consumption of the SRIO endpoint is to increase link utilization by reducing the amount of lanes and the maximum lane rate, as depicted in Figure 8.11. Such an approach is very effective, since it does

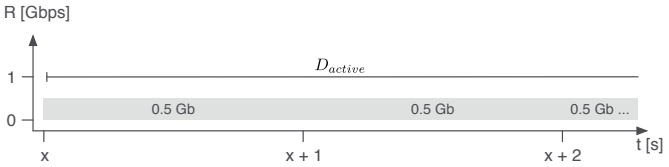


Figure 8.11.: Channel activity at a transmission rate of 0.5 Gbps at an available data rate of 2 Gbps and 0.5 Gbit of data.

not suffer from idle periods in which the link is only active for synchronization. However, it imposes a hard limit on the available data rate, so that the link is suitable only for evenly distributed traffic and cannot handle bursty traffic sources.

In contrast, a rather complex strategy is to keep the transceiver initially in the disabled state and only activate it in case of outgoing transmissions or link sensing for incoming communication. For egress data, the situation is very convenient, since the transceiver can be activated as soon as data is available for transmission. Data can then be transferred and the transceiver can be disabled afterwards, which is shown in Figure 8.12. Such an implementation is especially useful for signal processing applications, where data suffers a high latency due to processing but is delivered at a high throughput.

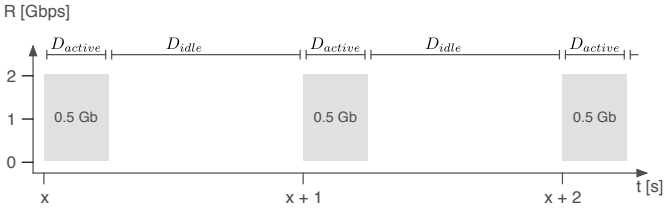


Figure 8.12.: Channel activity at a transmission rate of 2 Gbps at an available data rate of 2 Gbps and 0.5 Gbit of data.

Unfortunately, there exist several problems with such an approach. In Figure 8.13, we consider the link training delay after enabling the transceiver, which causes additional latency and reduces the link capacity.

Powering down the transceiver during idle periods can reduce the power consumption, however, incoming transmissions cannot be received. If the arrival

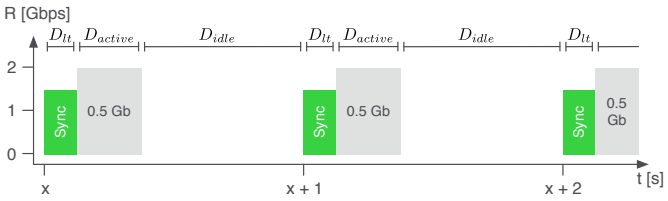


Figure 8.13.: Channel activity at a transmission rate of 2 Gbps at an available data rate of 2 Gbps and 0.5 Gbit of data. Additionally, the link training delay D_{lt} is depicted.

times of ingress transmissions are not known a priori, the transceiver must be periodically enabled to sense the link for prospective incoming data. Deciding on when to wake up and how long to listen is a complex optimization problem, since not only the power consumption of the transceiver but also that of excess memory necessary to store egress data at the sender until the link partner becomes available must be taken into account.

Embedded system applications that make use of complex serial interconnects often posses the advantage that data streams are known a priori and can therefore be scheduled. Especially if the system must fulfill real-time constraints, the exact scheduling of data streams is a common practice. In [SHT12*], we have proposed a novel budget-based power-management for a priori known data streams over SRIO communication links. In case of periodically scheduled communication streams, an advantage over the above mentioned power management strategies is that the transceivers can be activated in fixed intervals, which minimizes idle as well as link sensing periods and can therefore reach a very high level of energy efficiency.

8.2.1. Methodology

On top of the Xilinx SRIO IP core, we have integrated and implemented a Power Management Unit (PMU), to selectively disable the SRIO endpoint. The individual components of the SRIO architecture are traversed by data in sequence, according to the direction of the transmission. For example, in case of an egress transmission, a packet must first traverse the logical layer, before it is being held in the buffer between the LOGIO and the PHY until the PHY was able to successfully transmit it to the next SRIO device. This is crucial for the implementation of the PMU, which includes clock gating of the IP Core components in the order of the traversal. Another part of the SRIO, that can be disabled in idle periods are the MGTs. On a Virtex-6 LXT, which was used for this study, the GTX MGTs [Xil11b] are used as serial transceivers, which are organized in clusters of four transceivers, sharing two

differential reference clocks. Two analog supply powers are used, MGTAVCC and MGTAVTT. MGTAVCC is responsible for the internal analog circuits, including the Phase-Locked Loop (PLL) to synthesize a clock that matches the frequency of the clock that generates the incoming serial data stream, the transmitters and the receivers. MGTAVTT powers the termination circuits of the transmitters and the receivers. The GTX supports several power-down modes to facilitate the implementation of a generic power control. For both, the transmitter and the receiver lane, it is possible to put the actual transceiver as well as the PLL into a low power mode. However, it is only supported to power down the receiver in conjunction with the transmitter. At start-up or after a power-down of the GTX transceivers, the serial interfaces and the physical layer loose synchronization and must follow a protocol defined initialization routine to resynchronize. Important indicators are the link status, as well as the receive and transmit port of the module. The endpoint is functional, if all three indicators are asserted.

During the activation and deactivation process, an SRIO communication controller can be in one of five states, as depicted in Figure 8.14.

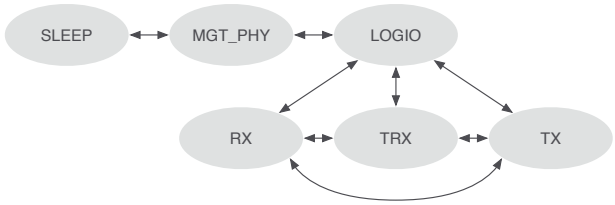


Figure 8.14.: Transceiver hardware enable states and transitions for SRIO communication events.

These are:

- SLEEP – The transceiver lanes as well as the PLL are powered down, the IP core and the user application are clock-gated.
- MGT_PHY – The transceiver are powered up and the clock to the physical layer is enabled.
- LOGIO – The logical layer and is enabled in addition to the MGT and the PHY.
- TX/RX – Either the transmitter or the receiver logic is activated in addition to the MGT, the PHY and the LOGIO.
- TRX – The complete transceiver logic is activated.

The initial state is SLEEP, in which the complete transceiver hardware is disabled. From here, to wake up the transceiver, the next state is MGT_PHY, where

the MGTs and the PHY are enabled to start link training. The transceiver will remain in this state for the duration of D_{lt} until the link is synchronized, after which the logical layer can be enabled in state LOGIO. Depending on whether only the transmitter, the receiver, or both components will be needed, the transceiver is then put into one of the corresponding states TX, RX, or TRX, respectively. The logical layer is enabled in all of theses states, however, the logic for transmitting and receiving can be selectively disabled, according to the task to fulfill.

Each of the states is associated with a power consumption P_x, where x denotes the current state of the hardware:

- P_S – Power in state SLEEP
- P_M – Power in state MGT_PHY
- P_L – Power in state LOGIO
- P_T – Power in state TX
- P_R – Power in state RX
- P_{TR} – Power in state TRX

We define the power level P_S during the state SLEEP as the *basic power reference level*. The change to other states by activating the communication controller elevates the power consumption. We denote $P'_x = P_x - P_S$ as the increase in power consumption in state x compared to the basic level of power consumption. For example, $P'_M = P_M - P_S$ expresses the increase in power consumption, if the MGTs and the PHY are activated. In contrast, if no power management is implemented, the communication controller is constantly in the state TRX and consumes an amount of energy equal to $P'_{TR} = P_{TR} - P_S$. We can achieve an improvement in power consumption if the mean value of power consumption over a certain period of time t, consisting of the sum of the power levels in the different states is smaller than the power consumption P'_{TR} over the same period of time. To elaborate on this, we examine the so far presented power management strategies.

Consider the case when there is no a priori knowledge of the duration of idle periods and data may arrive or must be transmitted at arbitrary points in time. We will first analyze the transmission process, which is depicted in Figure 8.15. Packets traverse several stages of the endpoint controller, each of which may selectively be enabled or disabled. The first step is to enable the MGTs and the PHY. Until the link is synchronized, the remaining hardware can stay in the disabled state. After synchronization, both, the logical layer and the transmit logic are activated in a single step to start the transmission.

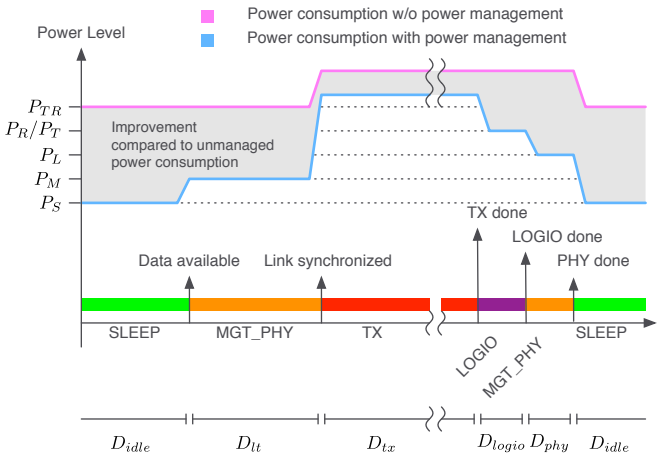


Figure 8.15.: Transmission process and power levels of managed and unmanaged SRIO endpoints. Higher power consumption is due to increased toggle activity during TX state.

When the last data packet was processed by the transmit component, it still has to traverse the logical layer and the PHY, so we sequentially disable the components, once activity has ceased. During transmission of the data, we experience an overall higher power consumption, which is due to the higher toggle rate in the individual components. We also depict the power consumption of an unmanaged endpoint controller in Figure 8.15 for comparison. Here, the power consumption is also elevated during the actual transmission. We can now evaluate the energy savings of this approach as follows: The energy $E_{unmanaged}$, consumed by the unmanaged endpoint controller is

$$E_{unmanaged} = P_{TR} \cdot (D_{idle} + D_{tl}) + P_{TX} \cdot D_{tx}, \tag{8.1}$$

whereas, the energy $E_{managed}$ consumed by the managed endpoint is

$$\begin{aligned} E_{managed} \quad = \quad & P_S \cdot D_{idle} + P_M \cdot D_{lt} + P_T \cdot D_{tx} \\ & + P_L \cdot D_{logio} + PM \cdot D_{phy}. \end{aligned}$$

The difference between both is displayed in Figure 8.15. Obviously, the longer the idle periods can be kept, the more effectively the energy can be reduced.

An advantage of this approach is that it only increases the delay by D_{tl} of a data stream, but does not decrease the maximum available line rate. The same observation holds for receiving packets, where the transceiver must be activated periodically in intervals D_{int} to probe the link for incoming transactions, as it is illustrated in Figure 8.16. However, this is only the case if the transceiver is deactivated. During active transmission periods, no extra hardware must be activated for link sensing and receiving. The worst case scenario happens if

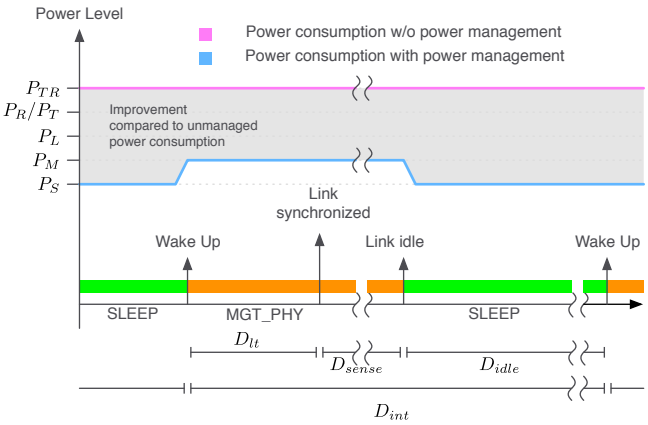


Figure 8.16.: Periodic link sensing process and power levels of managed and unmanaged SRIO endpoints.

there are no active periods due to transmissions and new ingress data becomes available right after the last sensing period D_{sense} has ended. The periodical interval comprises the idle time D_{idle}, the link training time D_{lt} and the sensing time D_{sense}. The maximum delay an ingress packet can experience is depicted in

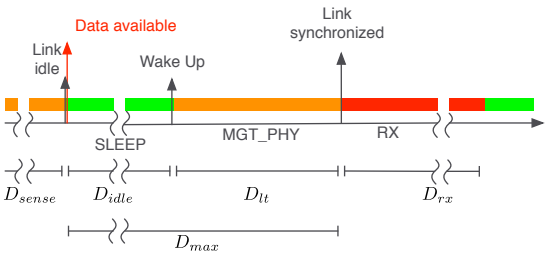


Figure 8.17.: Maximum delay D_{max} at periodic link sensing scheme.

Figure 8.17 and therefore $D_{max} = D_{idle} + D_{lt}$. The energy that can potentially be saved E_{save} in comparison to an unmanaged controller during such a cycle in the best case is the difference between the power consumed in state TRX during the time D_{max} and the power consumed in state SLEEP during D_{idle} and in state MGT_PHY during D_{lt}:

$$E_{save} = P_{TR} \cdot D_{max} - (P_S \cdot D_{idle} + P_M \cdot D_{lt}) \tag{8.2}$$

During idle periods without any incoming or outgoing transmissions, the potential saving evaluates to:

$$E_{save} = P_{TR} \cdot D_{int} - (P_S \cdot D_{idle} + P_M \cdot (D_{lt} + D_{sense})) \tag{8.3}$$

Furthermore, the sender must provide enough memory to buffer egress data while waiting for a sleeping link partner to wake up. Depending on the application,

the worst case requires a memory of size $K = D_{max} \cdot R$, where R is equal to the line rate. Providing such a memory will cause additional power consumption, which must be subtracted from the potential savings. Of course, if there are no transmissions, the memory can be disabled by clock gating, as well.

In case of a priori knowledge of the amount of data to transmit between link partners within a given time interval, we can circumvent the necessity to periodically activate the receiver for link sensing. According to a predefined schedule, the transceivers of both link partners are enabled, so that a common activity period D_{active} within an interval D_{budget} can be used to transmit and receive data between idle intervals D_{idle}. The duration of such an activity period depends on a specific data-budget between the link-partners that is defined at design time, but may also be adapted during run time. The amount of data to transmit and receive may not be equal. Thus, the active time equals to the already known delays for link training as well as IP core traversal times and maximum of the transmit time D_{tx} and the receive time D_{rx}, $D_{active} = D_{lt} + max(D_{tx}, D_{rx}) + D_{logio} + D_{phi}$. For example, in case of more data to be transmitted than received, the consumed energy during D_{active} evaluates to

$$\begin{aligned} E_{active} &= P_M \cdot D_{lt} + P_{TRX} \cdot D_{rx} + P_{TX} \cdot (D_{tx} - D_{rx}) \\ &\quad + P_L \cdot D_{logio} + PM \cdot D_{phy}. \end{aligned}$$

Apart from active periods, the controller can remain in the low-power sleep state during the idle period D_{idle}. An illustration of this case is depicted in Figure 8.18. The energy saved if the endpoint is managed by the budget-based protocol

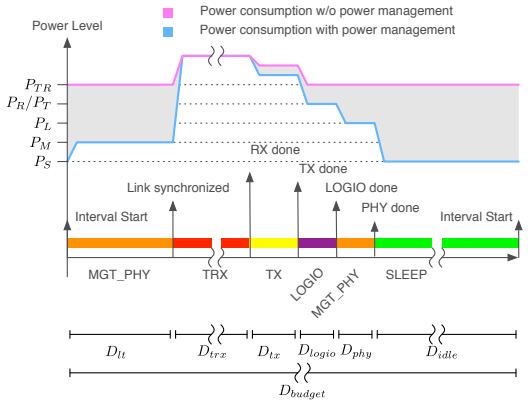


Figure 8.18.: Transmission process and power levels of budget-based power management and unmanaged SRIO endpoints. Higher power consumption is due to the increased toggle activity during TRX and TX state.

evaluates to

$$E_{save} = P_{TR} \cdot D_{budget} - (E_{active} + P_S \cdot D_{idle}). \tag{8.4}$$

8.2.2. Experimental results

In order to evaluate the proposed power management strategies, we have analyzed the PMU for the Xilinx SRIO IP core in version 5.6 using the Xilinx Virtex-6 LXT 240 FPGA on the ML605 evaluation board, which supports single lane SRIO architectures. Since the ML605 does not include an oscillator to drive the system clock, we have used an ML505 board as external source at 125 MHz. At 125 MHz, we can generate SRIO endpoints at line rates of 1.25, 2.5, 3.125 and 5 gBd. Before measuring the power consumption of the proposed power optimization strategy on actual hardware, we have performed an analysis by simulation.

Simulation-based Analysis

We have created two different versions of the SRIO endpoint controller, one with the PMU and one without power management. Without the PMU, we have synthesized the endpoint for two different optimization goals, minimum clock period (Speed) and minimum power (Power). The PMU-based endpoint design was optimized for minimum clock period (PMU). To estimate the improvement possible due to power management, the designs were analyzed using the Xilinx XPower Analyzer tool [Xil11d]. In comparison to the measurement on actual hardware, XPower offers the advantage, that it can provide an estimate of the power consumption of each individual component of the FPGA.

The results of the XPower Analysis are listed in Table 8.4 and were made under *commercial* settings for the temperature grade, as well as *typical* process settings. As expected, the largest part of the power consumption is due to leakage, which

Table 8.4.: Resource requirements and power analysis results for the SRIO endpoint enhanced by the PMU and unmanaged endpoint designs, optimized for minimum period (Speed), as well as minimum power consumption (Power), listed according to the line rate. Design names reflect the optimization goal and PMU usage.

Line Rate	1.25 gBd			2.5 gBd			3.125 gBd			5.0 gBd		
Design	Speed	Power	PMU	Speed	Power	PMU	Speed	Power	PMU	Speed	Power	PMU
CLKs	0.017	0.015	0.014	0.033	0.030	0.027	0.048	0.038	0.031	0.077	0.071	0.046
Logic	0.000	0.000	0.000	0.000	0.000	0.001	0.001	0.001	0.001	0.001	0.001	0.001
Signals	0.001	0.001	0.001	0.001	0.001	0.002	0.002	0.002	0.002	0.003	0.003	0.003
BRAM	0.005	0.005	0.005	0.011	0.010	0.010	0.014	0.011	0.013	0.022	0.018	0.021
MMCM	0.041	0.041	0.041	0.041	0.041	0.041	0.039	0.039	0.039	0.041	0.041	0.041
GTX	0.128	0.128	0.072	0.140	0.140	0.075	0.157	0.157	0.076	0.206	0.206	0.071
Leakage	3.360	3.360	3.358	3.361	3.361	3.358	3.362	3.361	3.359	3.365	3.364	3.359
Total	3.552	3.550	3.492	3.588	3.584	3.514	3.622	3.609	3.521	3.715	3.705	3.542

is an FPGA inherent problem. To visualize the remaining proportions, we have omitted the leakage part in Figure 8.19.

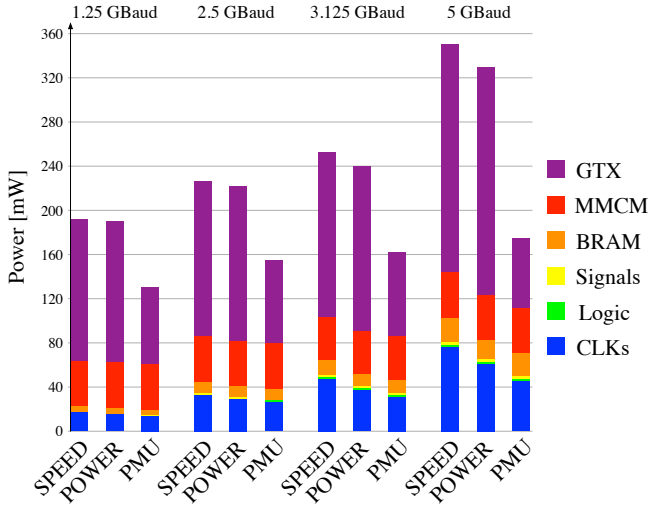


Figure 8.19.: Power requirements of individual FPGA components of the SRIO endpoint designs for multiple line rates (simulation-based).

A very large proportion of the remaining power consumption is due to the GTX transceivers, and deactivation of the transceivers in the PMU design can reduce the required energy by up to 75%. Moreover, it can be seen that the PMU designs can also reduce the energy required by the clock tree due to clock-gating, which outperforms the automatic approach of up to 35 % for the 5 gBd design. Moreover, we observe that the PMU requires extra logic, especially for the faster designs, which will also be noticeable in the hardware measurements. The downside of the XPower analysis is that the results do not very well reflect the dynamic behavior of the interconnect at different data budgets, which is why we cannot abstain from performing power measurements on the actual hardware.

Hardware-based measurements

The power supply on the Virtex-6 FPGA is controlled by a Texas Instruments (TI) UCD9240 controller. Measuring of the power consumption of the FPGA and the MGTs can be easily accomplished on the ML605 using the PMBus interface to the TI controller and the TI Fusion Digital Power Designer software package [Tex11]. For this study it is interesting to measure the power consumption on the VCCINT power rail, which drives the internal components of the FPGA, as well as the MGTAVCC and MGTAVTT power rails. The SRIO endpoint was implemented with and without the PMU. The design without the PMU

was used to generate a bit file with minimum period as design goal (Speed) and furthermore implemented with the Xilinx ISE internal power reduction option enabled (Power). The design with the PMU was optimized for minimum period (PMU). All three designs were implemented for all possible line rates at a 125 MHz system clock speed. Due to space restrictions, we only list the measurement results for 5 gBd in Table 8.5.

Table 8.5.: Power rail measurements of the 5 gBd design with and without PMU. The design without PMU was optimized for minimum period (Speed) and power reduction (Power). The PMU design (PMU) was optimized for minimum period.

Design	Speed			Power			PMU		
Rail	VCCINT	MGTAVCC	MGTAVTT	VCCINT	MGTAVCC	MGTAVTT	VCCINT	MGTAVCC	MGTAVTT
Budget	Power [W]								
0.0	1.0313	0.9421	0.9513	1.0274	0.9421	0.9513	0.9697	0.8317	0.8616
0.1	1.0350	0.9415	0.9517	1.0274	0.9415	0.9517	0.9777	0.8364	0.8681
0.3	1.0364	0.9419	0.9516	1.0289	0.9419	0.9516	0.9779	0.8446	0.8696
0.5	1.0439	0.9418	0.9515	1.0381	0.9418	0.9515	0.9997	0.8445	0.8744
0.7	1.0416	0.9418	0.9507	1.0370	0.9418	0.9507	0.9954	0.8542	0.8779
0.9	1.0472	0.9419	0.9515	1.0411	0.9419	0.9515	1.0105	0.8562	0.8836
1.1	1.0465	0.9414	0.9517	1.0470	0.9414	0.9517	1.0090	0.8612	0.8876
1.3	1.0508	0.9420	0.9512	1.0496	0.9420	0.9512	1.0237	0.8682	0.8903
1.5	1.0560	0.9417	0.9509	1.0497	0.9417	0.9509	1.0293	0.8746	0.8961
1.7	1.0574	0.9427	0.9511	1.0567	0.9427	0.9511	1.0448	0.8723	0.8940
1.9	1.0589	0.9416	0.9510	1.0573	0.9416	0.9510	1.0506	0.8850	0.8978

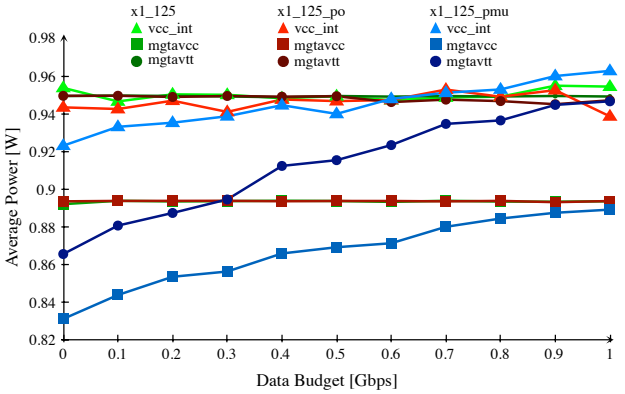


Figure 8.20.: Average power consumption of SRIO endpoint designs for 125 MHz system clock and 1.25 gBd line rate.

A plot of the measurements for the 1.25 gBd implementation is shown in Figure 8.20. The measurements were conducted up to the maximum data budget of 1 Gbps. To keep the rest of the measurements comparable, individual designs in Figures 8.21, 8.22, and 8.23 were measured up to 1.9 Gbps, although, the 3.125 and 5 gBd versions are capable of transmitting data at higher rates. The graphs show that disabling the GTX transceivers, affecting MGTAVCC and

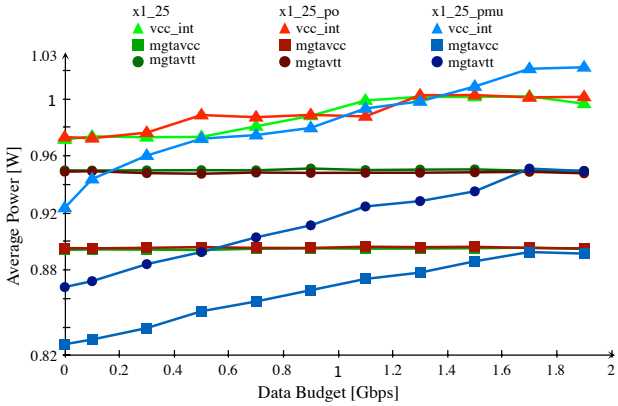


Figure 8.21.: Average power consumption of SRIO endpoint designs for 125 MHz system clock and 2.5 gBd line rate.

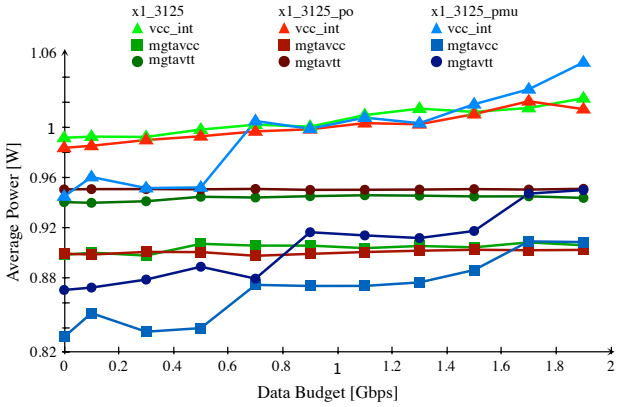


Figure 8.22.: Average power consumption of SRIO endpoint designs for 125 MHz system clock and 3.125 gBd line rate.

MGTAVTT, during idle periods is very effective, however, as the data budget approaches the maximum supported line rate, the measurements converge with those without the power management implemented. FPGA-internal clock-gating of the communication controller, which only affects VCCINT, is only marginally effective. On the contrary, once the data budget approaches the maximum achievable line rate, the power consumption is actually higher due to the extra logic of the PMU. Furthermore, the results show that although very fine-grained power reduction techniques prove to be effective, manual optimizations, such as disabling the GTX transceivers during idle periods, can outperform such techniques. The best results can of course be observed for idle periods, in which we can lower the combined power consumption of the GTX transceiver on power rails MGTAVCC and MGTAVTT by up to 200 mW. As specified earlier, we take the values at idle operation in the lowest power state as the basic reference

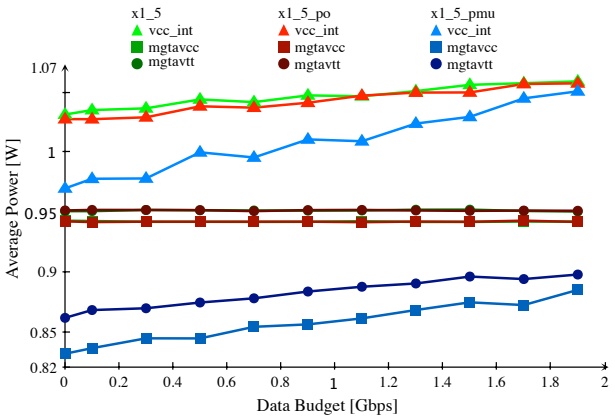


Figure 8.23.: Average power consumption of SRIO endpoint designs for 125 MHz system clock and 5.0 gBd line rate.

value. For the 5.0 gBd design, which represents the best case, we can achieve a reduction of the power consumption for the MGTs by 77 % on average, and for the internal FPGA design by 58 %. For the 1.25 gBd design, we can still achieve a reduction by 44 % on average for the GTX transceivers and by 18 % for VCCINT. Another observation is the comparison between lowering the maximum line rate for an underused link to avoid idle listening and raising the link rate to the maximum possible speed and use power management to increase idle periods, in which the controller can be deactivated by a PMU. A comparison for the

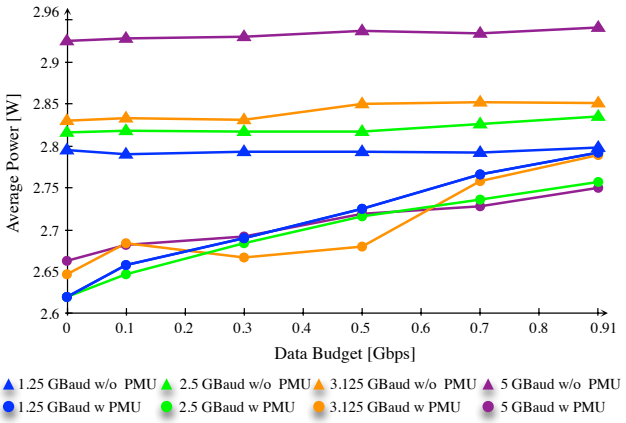


Figure 8.24.: Comparison of the average power consumption of SRIO endpoints for 125 MHz system clock and all possible line rates.

measurements up to a data budget of 0.9 Gbps is depicted in Figure 8.24. For communication controllers not involving power management, reducing the line

rate is very effective. However, using power management instead of lowering the maximum line rate can reduce the power consumption to a much higher degree.

8.3. SRIO User Application

The SRIO IP cores from Xilinx deliver an interface for the very complex SRIO standard that requires a lot of user interaction for data communication. For our purpose and applications, however, we demand for a communication interface that can initiate and receive data transfers autonomously. We therefore propose a *user application* in this section that provides the following features.

- Support of the AXI4-Stream (AXI4S) interface for data communication.
- Autonomous packet generation and transmission of user data using `SWRITE` transactions.
- Locally and remotely accessible configuration register for transaction properties, including Maximum Transfer Unit (MTU) size and destination identifier.
- Multiplexing of data streams across a single SRIO endpoint.

The *user application*, as illustrated in Figure 8.25, interfaces the *initiator/target* I/O port of the *LOGIO* IP core.

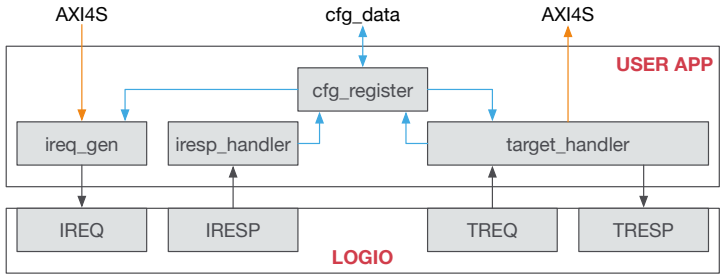


Figure 8.25.: Overview of the SRIO user application.

8.3.1. Autonomous Packet Transfer

SRIO defines an MTU of 256 bytes, which corresponds to 32 double word (8 bytes) data beats. For the first data beat of a packet, the *LOGIO* requires indication of the start of a new packet, as well as additional information for packet generation, such as data amount, transaction format and type, and target identifier. Our approach for autonomous packet generation only requires the user to submit data

to the user application using the AXI4S data port. Internally, the *initiator request generator* (ireq_gen) obtains all necessary information for packet assembly from the configuration register. According to a maximum data size, which should correspond to the data size of the images to transfer, the request generator wraps incoming data into as many full-sized `SWRITE` transaction packets as possible and eventually a smaller last packet. The packets are then sent to the target identifier specified in the configuration register. For ingress data, the *target request handler* extracts payload data from incoming `SWRITE` transactions and provides it to the egress AXI4S interface.

8.3.2. Transaction Configuration

The request generator receives all necessary information for packet generation from the configuration register. To accommodate for configuration changes at runtime, the configuration register can be accessed locally through the user application, but also remotely using `NWRITE` transactions, where the memory address in the packet describes the address in the register to modify. The local endpoint can also obtain register information from a remote register, for example, a master node, using the `NREAD` transaction. Incoming `NWRITE` transactions are forwarded to the user application through the *target request* interface of the *LOGIO* and are further processed by the *target request handler*. Remote reads must be issued by the user application's *initiator request generator* component. The response is received through the *initiator response handler*, which then updates the local configuration register with the new information.

8.3.3. Stream Multiplexing

The user application can multiplex different image streams over a single SRIO endpoint. According to the `tid`-identifier presented at the AXI4S ingress interface, the user application uses the corresponding entry of the configuration register to construct the egress transaction. Conversely, the user application applies a certain `tid`-identifier that corresponds to the *source identifier* of the ingress SRIO transaction when presenting data at the output AXI4S interface.

8.4. Integration of SRIO into the AXI4S-based FPGA Interconnect

In the same way, as we provided off-chip data to an image processing accelerator via PCIe, we can facilitate data communication via SRIO. A general system overview for integrating an SRIO endpoint is depicted in Figure 8.26. The proposed design employs the AXI4S interconnect component to periodically

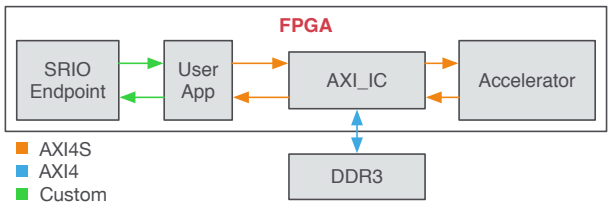


Figure 8.26.: Integration of SRIO into the AXI4S-based FPGA interconnect.

and autonomously transfer image data between the SRIO endpoint and the image processing accelerator, as well as serve as a Double Data Rate Type Three (DDR3) abstraction layer.

Data Path Adaptation

Image data in this design is received from the SRIO endpoint. Depending on the number of lanes and the lane rate, the AXI4S interconnect must convert the data path to interface the DDR3 and the image processing accelerator. For example, an SRIO endpoint using four lanes, operating at a clock frequency of up to 156.25 MHz with a line rate of 3.125 gBd provides data at a word width of 64 bits at the same clock frequency as the endpoint component. Next to several SRIO-specific protocol information, the data interface of the user front end to the SRIO endpoint uses a simple ready/valid handshaking mechanism, which seamlessly connects to the AXI4S interface of the interconnect.

8.5. Summary

In this chapter, we have introduced the serial incarnation of the RIO specification, SRIO, as a highly efficient embedded systems interconnect. A key aspect for the suitability of SRIO for board to board communication is that the protocol is highly scalable, supports a large number of communication methods and can operate autonomously in the absence of software. Moreover, the protocol is highly reliable, which makes it very attractive for harsh and critical application environments, such as medical image processing. Although RIO is a very sophisticated specification, it does not consider power efficiency. A modular FPGA implementation, however, allowed us to include a Power Management Unit (PMU) to significantly reduce the interconnects power consumption for periodic data transfers [SHT12*]. Since SRIO is a very complex protocol, its interface requires a high amount of user interaction for data communication. In order to include an SRIO communication endpoint as a system-level component in an FPGA-based SoC design, we have proposed a user application as a front

end to the Xilinx IP cores, which facilitates autonomous packet generation and data communication, as well as allows for local and remote configuration changes at runtime. We have concluded the chapter by demonstrating how an SRIO endpoint augmented with the here presented user application can be seamlessly integrated into an on-chip AXI4S interconnect as part of an FPGA-based SoC design.

9 Rapid Prototyping of FPGA Accelerators

A fundamental principle in verification is that *board-level prototyping* cannot and should not be considered as a replacement for simulation-based system verification. A major issue is *limited visibility*. Although there are a lot of options for actual chip debugging, including logic analyzers and on-chip recording, it is often only possible to record a limited period of time and only signals that were chosen for sampling can be captured. A second important point is that conditions that lead to an error must be *reproduced* exactly in order to verify that a specific problem was solved. If a problem was observed on a prototype, it might not be possible to exactly reproduce the conditions that caused it. In consequence, if the problem does not come up again while testing the prototype after a fix, it can only be asserted that the problem has not been seen for a while. The third major problem is *verification coverage*. The reason for creating a prototype is to evaluate the design's behavior under real-world conditions, avoiding design-errors. In a lab environment, however, these conditions might or might not include various corner cases that the design must handle in the field and thus, the coverage of the test scenarios might not suffice to verification.

Despite these facts, there are also ample reasons for including rapid prototyping in a design flow. For instance, in the context of system-level design, the expedited turnaround-times provided by rapid prototyping are advantageous for design-space exploration and evaluation of design decisions. Specifically in the image processing domain, testing of novel image processing accelerators is very important to validate their functionality and evaluate their performance, both, in terms of execution speed, but also with respect to accuracy. Without an available prototype, testing a hardware accelerator can only rely on logic simulation. In case of complex algorithms, simulation may take a very long time and can thus hinder productivity. For example, the multiresolution analysis algorithm used for evaluation in Chapter 6 takes over a day to simulate due to the large problem size and the off-chip memory. In contrast, the accelerator can be tested in less than a second on actual hardware.

An ideal development flow should allow the designer to evaluate the High-Level Synthesis (HLS)-generated accelerator in a familiar, software-based environment at the work place, for example, using a Field Programmable Gate Array (FPGA) card connected to the workstation using Peripheral Component Interconnect Express (PCIe). Such a development environment is possible using a hardware prototype. However, assembling such a design is beyond the scope of current HLS tools as it requires the design of a system architecture and integration of the developed accelerator into the system. It would moreover not be feasible to involve a hardware engineer in prototype building for every iteration during accelerator development, as this would take about the same time as simulating the design. A viable solution is to design a system architecture once into which an algorithm designer can then insert the prototype design without involving the hardware engineer.

To facilitate such an environment, this chapter makes the following contributions:

- *FPGA support designs* that allow for a holistic and fully automated system integration and project compilation.
- A *prototyping design flow* that facilitates (a) automatic hardware generation, and system integration for (b) software-based verification and (c) real-world context system-evaluation.

9.1. FPGA Support Systems

FPGA support systems are modular FPGA-based System-on-a-Chip (SoC) designs that make use of a standard On-Chip Interconnect (OCI) to (a) provide data and information exchange between the individual modules of the system architecture and (b) support connectivity to off-chip peripherals, such as external memory, but also board to host or board to board communication. An appropriate OCI is the AXI4-Stream (AXI4S) interconnect, introduced in Section 6.2.4 to facilitate connectivity between a HLS-generated image processing Intellectual Property (IP) core and other elements of the SoC architecture. The integration of board to host communication using PCIe into the OCI was presented in Section 7.2. Combining these results, a system architecture to support a software-controlled test environment for HLS-generated IP cores may be automatically generated as shown in Figure 9.1.

A similar concept can be used by replacing the PCIe endpoint with an Serial RapidIO (SRIO) endpoint. A vital component is the *user application* to autonomously handle packet generation and transmission, introduced in Section 8.3, as *glue logic* between the SRIO endpoint and the OCI, as proposed in Section 8.4.

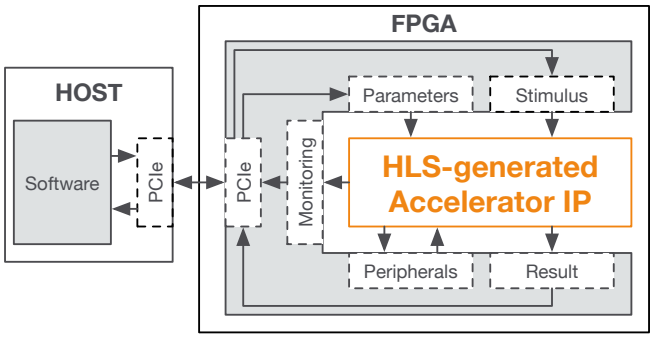


Figure 9.1.: Conceptual representation of the FPGA support design for holistic software-controlled prototyping of HLS-generated IP cores for image processing.

A conceptual illustration of the support design for system deployment is depicted in Figure 9.2.

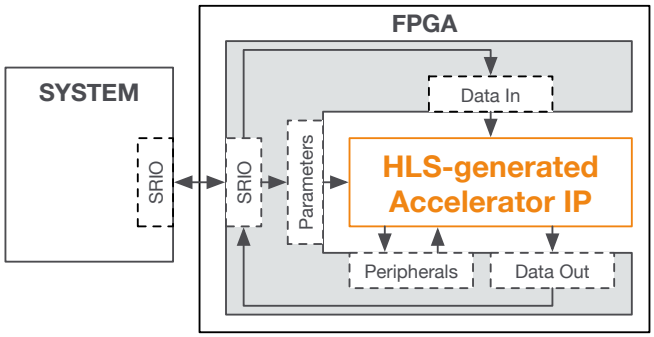


Figure 9.2.: Conceptual representation of the FPGA support design for system deployment of HLS-generated IP cores for image processing.

Although the integration of the accelerator in the form of a HLS-generated IP core into the FPGA support systems only requires interconnecting some signals on the top-level of the design, achieving timing closure during implementation might pose a significant challenge, as the communication controllers and the interface to external memory require a specific clock frequency. A strategy to avoid such problems is *incremental compilation*.

9.1.1. Incremental Compilation

Incremental compilation is a design technique initially proposed to handle very complex FPGA-based SoC designs in team-based development. Complex FPGA

designs may take a very long time to compile and although timing closure might have already been achieved, a small change to one of the system components might cause timing violations. Incremental compilation aims at avoiding such situations by partitioning the design, so that each component can be implemented individually without affecting other parts. A separate compilation may not only dramatically reduce the compilation time, it may also significantly reduce the effort to achieve timing closure, as components that have already been timing closed can be locked down to preserve their routing. When additional components are added at a later time or existing ones are modified, their compilation does not affect locked components. The methodology can also be applied to components that do not change to reuse their implementation results for multiple design iterations.

Fortunately, the timing critical elements of an FPGA support design are also those components that remain unchanged during prototyping. These components include PCIe communication, the interface to external DDR3 memory, as well as the AXI4S OCI. Although timing closure is also important during the implementation of an accelerator, the situation is more relaxed for prototyping than for the final deployment as part of a complete system. Instead of targeting a specific clock frequency, the situation can be turned around by supplying the maximum supported clock frequency to the accelerator. In this area, we propose to include a programmable clock generator in the design and supply the generated clock to the accelerator. The OCI is moreover designed to support an individual clock domain for each connected component and handles the transition to its frequency domain internally.

9.1.2. Out-of-Context System Design

In Xilinx terminology, this top-down incremental design methodology is named *top-down reuse* and the implementation of the components is referred to as *Out-Of-Context* (OOC) implementation. The methodology also has several peculiarities in comparison to the normal compilation flow of complete design. The key factor to achieve acceptable results is floor planning and constraining the design components to specific areas of the FPGA (called *partition blocks*, or short *pblocks*), which may not overlap or contain nested partitions. This prevents optimization across design blocks and reduces device utilization. The locations of clock buffers must be known beforehand and locked down by hand. Buffers inside components must also be locked and those in the top-level design must be marked for implementation. All module pins that connect to the outside of the component must be identified and cross-component timing constraints must be specified. Dedicated connections to Input/Output (I/O) components, such as I/O buffers must be moved from the top-level into the modules. Also, modules to be reused do not support generics or synthesis parameters at the top-level, which means

that these must be made static and moved inside of the component. Furthermore, the necessary logic for using Multi-Gigabit Transceivers (MGTs) is not supported inside of OOC-components and must be moved to the top-level. The same applies to I/O delay primitives, which are used in the PCIe and Double Data Rate Type Three (DDR3) controllers. The top-down flow requires an initial compilation of the complete design to evaluate timing requirements. For this, the OOC components are included as black boxes that span the allocated pblocks. This initial compilation usually takes a substantially long time. Figure 9.3 illustrates the architecture of the FPGA support design for incremental compilation using the Xilinx design flow.

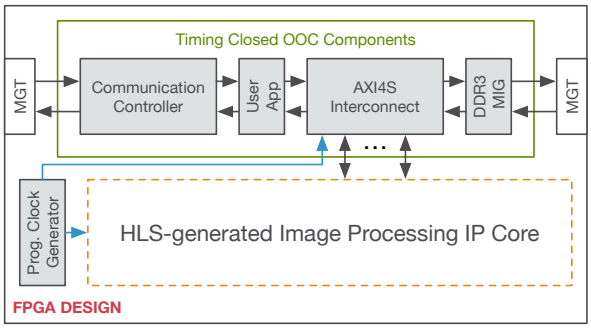


Figure 9.3.: OOC system architecture of the FPGA support designs.

9.2. A Fully Automated Rapid Prototyping Design Flow

As stated in the beginning of this thesis, the overall goal is to make FPGA design more approachable for software programmers with only limited hardware design experience. For this, we propose a three-step design and prototyping flow, as shown in Figure 9.4.

9.2.1. Develop

The first step is to develop the function to be accelerated and synthesize it using HLS. To support developers in obtaining highly efficient and high-performance results we have proposed a library of code templates, classes, and functions for the design of image processing algorithms in Chapter 4. If the performance of the synthesized accelerator is not sufficient, the design can additionally exploit Data-Level Parallelism (DLP) using *loop tiling* or *loop coarsening*, as explained in Chapter 5. The two parallelization techniques are included in the library and

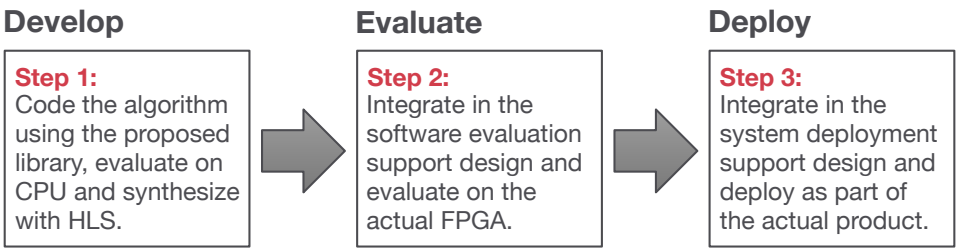


Figure 9.4.: Fully automated three-step design and prototyping flow for FPGA-based accelerator designs.

can be applied with only minor changes in the accelerator source code (refer to Section 5.2.5 for loop tiling and Section 5.3.4 for loop coarsening). As the library components are fully functional C++ code, the implementation can be tested and verified on the CPU using the GNU Compiler Collection (gcc) tool chain. Once the synthesis of the design delivers satisfying results, the designer can export the design as an IP core for system integration and move on to evaluate the design on the actual hardware.

9.2.2. Evaluate

To evaluate the design on actual hardware in a software-based environment, the IP core obtained from HLS is integrated into the SoC-like FPGA support design for software-controlled evaluation, as shown in Figure 9.1. Our library is designed to use the AXI4S protocol as interface between the hardware accelerator and the AXI4S-based OCI. System integration can therefore be automated by connecting the ports of the accelerator to the corresponding ports of the support design. The support design makes use of incremental compilation, where the system architecture is implemented and timing-closed independently of the accelerator IP core. To implement the final FPGA design, it is only necessary to compile the IP core and program the clock generator with the appropriate design frequency. To support enumeration during the boot procedure of the PCIe root complex (refer to Section 7.1.1), the design must be stored in the flash memory of the FPGA before it can be used for prototyping. In addition to the FPGA support design, we also supply the necessary device driver to use the FPGA in a Linux operating system. The device driver provides a *character device* interface for moving large amounts of data between a software program for evaluation and the FPGA. A very simple test program is shown in Listing 7.1 in Section 7.3. To monitor the FPGA device, the support design may, for example, collect information of the power requirements and device temperature, the device driver contains an *I/O control* interface (*ioctl*), that can be accessed independently of

the character interface to obtain the necessary information from the device. An example for obtaining power and temperature information is shown in Listing 7.2 in Section 7.3. The ioctl-interface can also be used to communicate information to the device, for example, filter parameters for the hardware accelerator. Once the evaluation of the design on the FPGA could determine that the accelerator performs satisfactorily, the designer can move on to the deployment step.

9.2.3. Deploy

The same IP core that was used for evaluation can be also used for deployment by integrating it into the FPGA support design for system deployment, which is shown in Figure 9.2. The system integration procedure is similar to that explained in Section 9.2.2. In our example, we use the SRIO protocol to establish the required board to board communication for connecting the hardware accelerator to an actual product in form of a distributed embedded system (refer to Figure 5.26). In principle, the SRIO protocol can be replaced by any embedded systems communication protocol that is supported by the FPGA and used in the system, for example, Interlaken or Ethernet.

9.3. Summary

In this chapter, we have explained how the contributions of the previous Chapters 6 to 8 are combined to assemble SoC-like FPGA support systems to allow a simplified system integration that can be used to evaluate a HLS-generated IP core on actual FPGA hardware using rapid prototyping. We apply incremental design compilation to implement the components of the system architecture of the FPGA support designs, which may comprise modules for evaluation and monitoring of the IP core, as well as communication endpoints and off-chip peripheral controllers. In this way, we may achieve timing closure for the system architecture components independently from the IP core to be tested, which can therefore be reused for multiple design iterations. When the designer adds the accelerator IP core to the support design at a later time, only the IP core must be compiled. To avoid any complications during timing closure, we add a programmable clock generator to the design that can be configured to deliver the maximum clock frequency supported by the accelerator IP core. We have furthermore explained how the contributions of the individual chapters of this thesis are used to obtain a holistic and fully automized design flow for rapid prototyping of image processing accelerators on FPGAs to make developing hardware accelerators approachable also for designers who do not have any prior experience with FPGAs. The design flow is structured into a three-step procedure. In the first step, the designer develops an FPGA accelerator using

the classes and functions of the library introduced in Chapters 4 and 5 and evaluates the design using the gcc tool chain. Once the synthesis of the accelerator using HLS delivers satisfactory results, the design is exported as an IP core and integrated into the FPGA support design for software-controlled evaluation. After configuring the FPGA with the design, it can be evaluated in a familiar, software-controlled environment. Finally, once the designer has verified that the design performs as desired, the same IP core that was generated for software-based evaluation can be inserted into the support design for system deployment.

10 Conclusions and Future Work

The design of dedicated hardware accelerators is gaining more and more importance as the demand for efficient computing is burgeoning. Field Programmable Gate Arrays (FPGAs) are a particularly interesting platform to accelerate compute-intensive applications, since they offer many of the advantages of a full custom Application-Specific Integrated Circuit (ASIC), however, at considerably lower development costs. Although the synthesis steps at the lower abstraction levels of the design cycle have been successfully automated in recent years, a significant gap remains at the system-level, where many tasks must still largely be carried out by hand. The approaches for domain-specific High-Level Synthesis (HLS) and interface synthesis, proposed in this work, present necessary preparations to close this gap at the system-level. In combination, they facilitate a holistic and fully automated approach to make the FPGA platform more accessible for software developers who might lack the necessary hardware design experience.

10.1. Summary

In this thesis, we have foremost concentrated on (a) domain-specific HLS, and (b) interface synthesis for FPGA-based System-on-a-Chip (SoC) integration of image processing Intellectual Property (IP) cores. As current HLS tools still need al lot of domain-specific experience, we have proposed a lightweight library of generic classes, methods, and functions for basic image processing operations. The library can be used as a high-level abstraction to assemble even complex pyramidal algorithms in a very intuitive way for the automatic generation of image processing IP cores using HLS. We have shown that the resulting IP cores may achieve a high throughput, use the resources of an FPGA efficiently, and provide a more energy-efficient solution than software-based accelerators, e. g., Graphics Processing Units (GPUs). In terms of system-level design, the use of the library facilitates (a) software-based localization and elimination of programming errors, (b) rapid design space exploration, (c) optimization with respect to specific design goals, and may thus remarkably expedite the design cycle.

As an FPGA is initially unprogrammed, actually employing a HLS-generated image processing IP core requires to integrate it into its on- and off-chip environment. A vital part to automate this step is *interface synthesis* for FPGA-based SoC integration of accelerator IP cores. The contributions in this area allow (a) software-based board-level evaluation and verification of a HLS-generated IP core as part of a hardware/software co-design, and (b) system-level evaluation to rapidly validate the design decisions made at the system-level. In order to automate the procedure, we propose the use of so-called FPGA support designs. A support design provides a timing-closed SoC, consisting of communication controllers to implement the allocated communication channels, as well as an On-Chip Interconnect (OCI) to facilitate the integration of the IP core. In this way, we contribute a holistic and fully automized approach for rapid prototyping of hardware accelerators that may decisively increase productivity and puts the FPGA platform within reach of software developers. Although we concentrate on medical image processing in this work, the contributions are also applicable to many other application areas.

10.2. Future Work Directions

To extend the work discussed in this thesis, the proposed library was employed to create an FPGA back end for the open source code generation framework HIPAcc [Mem+15b], as shown in Figure 10.1. Built upon a Domain-Specific

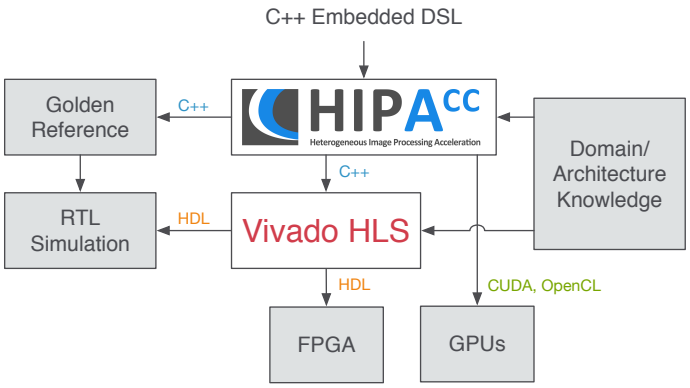


Figure 10.1.: HIPAcc design flow for generating FPGA accelerators.

Language (DSL) for the image processing domain, HIPAcc can generate highly efficient algorithm implementations for heterogeneous accelerators, such as GPUs, embedded GPU (eGPU), and multi-core processors. Code generation, such as offered by HIPAcc, provides true portability of programs and performance across different platforms. Moreover, it delivers increased productivity, as developers

need not be concerned about implementation details, but can focus on functionality. In combination with general purpose HLS frameworks, such as Vivado HLS, support of a wide range of applications can be accomplished efficiently. As presented in [Rei+14*], the FPGA designs generated by HIPAcc using the library presented in Chapter 4 as an abstraction for code generation, may deliver a higher performance than eGPUs and are significantly more energy efficient than server-grade GPUs. The library templates for Multiresolution Analysis (MRA), proposed in Section 4.7, are moreover suitable for scientific computing. This was shown in [Sch+14c*] as an extension for HIPAcc, and in [Sch+15b*] for the DSL ExaSlang. In [Sch+15a*], we have augmented the FPGA back end of HIPAcc to also support loop coarsening, as explained in Section 5.3. The experimental results show that the generated FPGA accelerators can also deliver a higher throughput than high-performance GPUs if loop coarsening is applied.

The work presented in Part II of this thesis could also be extended, so that HIPAcc would be able to generate appropriate FPGA support designs, as well as software applications and device drivers to facilitate board-level evaluation and system deployment. A possible design flow is shown in Figure 10.2.

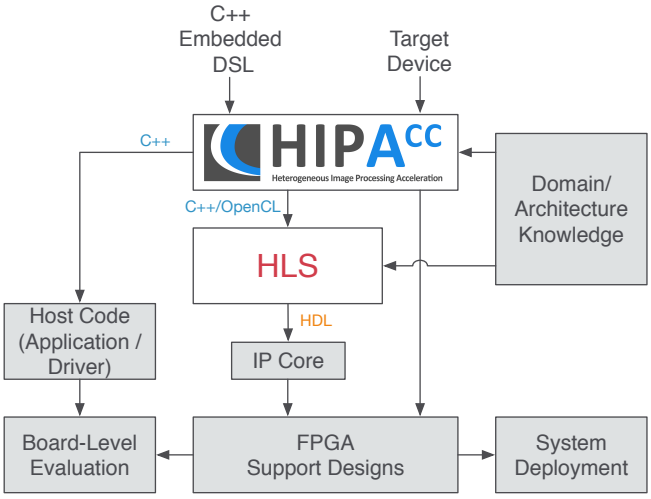


Figure 10.2.: HIPAcc design flow for board-level evaluation and system deployment.

German Part

Schnelle Prototyperstellung von Hardware-Beschleunigern für die medizinische Bildverarbeitung

Zusammenfassung

Aufgrund des enormen technischen Fortschritts der vergangenen Jahre verfügen rekonfigurierbare Logikbausteine (engl. *Field Programmable Gate Array* (FPGA)) mittlerweile über eine ausreichend große Menge an Logikressourcen, um auch sehr komplexe Hardware-Systeme, zum Beispiel in Form einer Ein-Chip-Realisierung (engl. *System-on-a-Chip* (SoC)), implementieren zu können. Im Vergleich zu anderen oftmals verwendeten Beschleunigern, wie zum Beispiel Grafikkarten (engl. *Graphics Processing Unit* (GPU)), kann mit FPGAs eine vergleichbare Verarbeitungsgeschwindigkeit bei jedoch wesentlich höherer Energieeffizienz erreicht werden. Obwohl in der Entwurfsmethodik für Hardware-Systeme enorme Fortschritte hinsichtlich der Entwurfsautomatisierung erreicht werden konnten, existieren beim Entwurf von Schaltungen für FPGAs immer noch genügend schwierige Teilschritte die unerfahrene EntwicklerInnen davon abhalten sich bei der Konzipierung eines Systems für die FPGA-Plattform zu entscheiden. Um den Entwurf von FPGA-basierten Hardwarebeschleunigern in Form von Funktionsblöcken, sogenannten IP-Cores, auch für EntwicklerInnen mit nur wenig bzw. ohne Erfahrung im Hardware-Entwurf zu ermöglichen, werden in dieser Arbeit relevante Beiträge für den Entwurf auf der Systemebene vorgestellt. Insbesondere werden hierbei die Bereiche der

(a) domänenspezifischen Architektursynthese von IP-Cores und der

(b) Schnittstellensynthese für die Integration von IP-Cores in FPGA-basierte SoCs behandelt.

Viele der heute verfügbaren Werkzeuge für die Architektursynthese (engl. *High-Level Synthesis* (HLS)) erlauben die Verwendung von höheren Programmiersprachen, wie zum Beispiel C++, für die Verhaltensbeschreibung. Jedoch erfordert die Angabe einer geeigneten Spezifikation nach wie vor fundierte Kenntnisse aus dem Bereich des FPGA-Entwurfs um als Ergebnis der Architektursynthese auch eine hinreichend effiziente Hardware-Realisierung zu erhalten. Oftmals ist es auch nötig mit weiteren Prinzipien, wie z. B. der Parallelisierung, vertraut zu sein, um durch Angabe geeigneter Synthesedirektiven gezielt Einfluss auf die einzelnen Schritte der Synthese, bestehend aus Allokation, Ablaufplanung

und Bindung, nehmen zu können. Aus diesen Gründen können viele der heute verfügbaren HLS-Werkzeuge noch nicht ohne entsprechende Erfahrung eingesetzt werden.

Eine Verbesserung dieser Situation kann durch die Verwendung einer in dieser Arbeit vorgeschlagenen Bibliothek erreicht werden. Mit deren Hilfe ist es möglich, auch ohne relevante Entwurfserfahrung für die Architektursynthese geeignete Verhaltensbeschreibungen zu erstellen. Die Bibliothek wurde in der Programmiersprache C++ entworfen und ist besonders für den Einsatz im Bereich der medizinischen Bildverarbeitung ausgelegt. Algorithmen dieser Domäne können häufig als Komposition von grundlegenden Bildverarbeitungsoperationen, wie Punkt- und lokalen Operatoren, realisiert werden. Daher werden durch die Bibliothek keine komplexen Algorithmen sondern grundlegende, generische Klassen und Funktionen für diese Operatoren, als auch zur Unterstützung deren Kombination als Bildverarbeitungskette bereitgestellt. Synthese-Direktiven sind ebenfalls bereits im Quellcode der Bibliothek enthalten und können auf einfache Weise modifiziert werden indem sie jeweils an zentralen Stellen zusammengefasst aufgeführt sind. Dies ist insbesondere für die Entwurfsraumexploration auf Systemebene von Vorteil, da sich die mit Hilfe der Bibliothek erstellten Spezifikationen hinsichtlich unterschiedlichen Entwurfszielen, wie besonders hoher Durchsatz oder sehr niedriger Ressourcenverbrauch, optimieren lassen. Darüber hinaus ermöglicht die modulare Struktur der Bibliothek deren Erweiterung um neue Funktionalität auf besonders einfache Weise. Da die Bibliothek in der Programmiersprache C++ geschrieben wurde, können die mit Hilfe dieser Programmierabstraktion erstellten Verhaltensbeschreibungen wie normale Software-Programme ausgeführt und dabei evaluiert und verifiziert werden. Dies kann in einer üblichen Software-Umgebung geschehen und erfordert keine besonderen, rechnergestützten Entwurfswerkzeuge oder Kenntnis des Hardware-Entwurfs. Hierdurch wird auch eine Reduzierung der Entwurfszeit erreicht, da die funktionale Verifikation der Verhaltensbeschreibung als Software-Programm erheblich schneller ausgeführt werden kann als die Simulation einer vergleichbaren Schaltung auf der Register-Transfer-Ebene.

Zur Bewertung der Qualität der unter Verwendung der Bibliothek erreichbaren Syntheseergebnisse, wurden für mehrere typische Algorithmen aus dem Bereich der medizinischen Bildverarbeitung mit Hilfe der Bibliothek Spezifikationen erstellt. Unter Zuhilfenahme des HLS-Werkzeugs Vivado HLS, wurden anschließend entsprechende IP-Cores synthetisiert. Hierbei konnte gezeigt werden, dass die mit Hilfe unserer Bibliothek synthetisierten IP-Cores einen höheren Durchsatz und eine effizientere Ausnutzung der auf dem FPGA verfügbaren Hardware-Ressourcen aufweisen, als die mit Hilfe einer vergleichbaren Bibliothek erzielten Ergebnisse. Des Weiteren wurden die Bildverarbeitungsalgorithmen auch für die Ausführung auf unterschiedlichen GPUs implementiert, sowohl aus dem Hochleistungsbereich als auch für eingebettete Systeme. Im direkten

Vergleich können die für den Einsatz im Rahmen eines FPGA-basierten SoCs synthetisierten IP-Cores einen höheren Durchsatz sowie eine signifikant höhere Energieeffizienz als die GPU-Implementierungen erreichen.

FPGAs sind anfangs nicht konfiguriert, so dass für den Einsatz einer auf dem FPGA zu verwendenden Hardware-Komponente auch deren Einbettung in eine entsprechende Systemarchitektur zur Kommunikation mit weiteren auf dem FPGA implementierten Komponenten als auch mit der Außenwelt erstellt werden muss. Eine solche Architektur kann zum Beispiel als SoC realisiert sein. Um die nötigen Voraussetzungen für die Automatisierung dieses Entwurfsschritts zu schaffen, beschäftigen wir uns in dieser Arbeit weiterführend mit der Schnittstellensynthese (engl. *interface synthesis*) für die Integration von automatisch synthetisierten Bilverarbeitungs-IP-Cores in FPGA-basierten SoCs.

Für die Bereitstellung der nötigen Kommunikationskanäle zwischen den allozierten Systemkomponenten schlagen wir die Verwendung eines im FPGA implementierten Verbindungsnetzwerks (engl. *On-Chip Interconnect* (OCI)) vor. Dieses basiert auf einer standardisierten Datenbusspezifikation die den Transport von Daten sowohl als Datenstrom als auch über speicherbezogene Abbildung (engl. *memory-mapped*) unterstützt. Jegliche Art von Anpassung, z.B. der Bitbreite oder der Taktrate, zwischen den Schnittstellen verbundener Komponenten ist hierbei durch das OCI implementiert. Auf diese Weise müssen bei der Synthese der Bildverarbeitungskomponenten keine besonderen Anforderungen anderer Systemkomponenten berücksichtigt werden. Dies fördert die Interoperabilität mit einer großen Menge an verfügbaren IP-Cores und unterstützt den Entwurf von anwendungsspezifischen IP-Cores und Verbindungsprotokollen.

Ein weiterer Schwerpunkt besteht in der Anbindung der Bildverarbeitungs-IP-Cores an die Außenwelt, zum Beispiel an einen Host-Computer, oder an weitere SoCs im Rahmen eines verteilt implementierten eingebetteten Systems. Die Kopplung der auf dem FPGA implementierten IP-Cores mit einem auf dem Mikroprozessor eines Host-Computers ausgeführten Anwendungsprogramm geschieht mit Hilfe des Verbindungsstandards *Peripheral Component Interconnect Express* (PCIe). Für die Realisierung als Hardware/Software-System wurde auf der Hardware-Seite hierfür eine Systemkomponente zur Implementierung der PCIe-Kommunikationsschnittstelle über das OCI mit dem Bildverarbeitungs-IP-Core verbunden. Auf der Software-Seite wurden für den Datenaustausch geeignete Software-Schnittstellen als auch ein entsprechender Gerätetreiber für das Linux-Betriebssystem entworfen. Die Integration eines Bildverarbeitungs-IP-Cores in ein verteilt implementiertes eingebettetes System wurde hingegen unter Verwendung eines sogenannten *Serialisierer/Deserialisierer*-Protokolls (abgek. SerDes) für die serielle Datenübertragung zwischen parallelen Endpunkten implementiert. Als Beispiel dient der Verbindungsstandard *Serial RapidIO* (SRIO), der speziell für die Kopplung von Hardware-Systemen konzipiert wurde und vollständig als Hardware-Komponente realisiert werden kann. Hervorzuheben ist der Entwurf

einer anwendungsspezifischen Systemkomponente zur autonomen Generierung von SRIO-Paketen für Datenstrom-basierte Anwendungen. Diese dient hierbei als Front-End und ermöglicht somit die Integration der SRIO-Schnittstelle in das OCI eines FPGA-basierten SoC.

Basierend auf den entwickelten Grundlagen für die Entwurfsautomatisierung auf Systemebene, wird ein ganzheitlicher, vollständig automatisierbarer Ansatz zur schnellen Prototyperstellung von IP-Cores für die Implementierung von medizinischen Bildverarbeitungsalgorithmen als Teil eines FPGA-basierten SoC vorgestellt. Der Ansatz besteht aus den folgenden drei Teilschritten:

1. Mit Hilfe der vorgeschlagenen Bibliothek als Programmierabstraktion wird eine Verhaltensbeschreibung in einer höheren Programmiersprache erstellt. Diese kann zunächst als Software-Programm ausgeführt und hinsichtlich der Zielvorgaben bewertet und optimiert werden. Sobald die Verhaltensbeschreibung die Vorgabe erfüllt, wird mit Hilfe der Architektursynthese ein Hardware-Entwurf in Form eines IP-Cores generiert.

2. Der generierte IP-Core wird in ein FPGA-basiertes SoC integriert und auf einer FPGA-Entwicklungsplattform die über PCIe mit einem Host-Computer verbunden ist implementiert. Als Teil eines Hardware/Software-Systems kann der generierte IP-Core mit Hilfe eines Software-Programms evaluiert und verifiziert werden.

3. Abschließend erfolgt der Test der generierten Hardware-Komponente im Feld, zum Beispiel als Teil eine Produkts. Hierfür wird der IP-Core in ein FPGA-basiertes SoC integriert, das die Verbindung zum Zielsystem über eine geeignete Kommunikationsschnittstelle, zum Beispiel SRIO herstellt.

Die in dieser Arbeit vorgestellten Ansätze für die domänenspezifische Architektur- und Schnittstellensynthese leisten einen wesentlichen Beitrag zur Entwurfsautomatisierung von FPGA-basierten Systemen und ermöglichen hierdurch die Erschließung der FPGA-Technologie für Software-EntwicklerInnen. Die vorgeschlagene Programmierabstraktion in Form einer Bibliothek erlaubt einen besonders intuitiven Entwurf von Verhaltensbeschreibungen für die Architektursynthese in der Programmiersprache C++. Dies ermöglicht (a) die Software-gestützte Lokalisierung und frühzeitige Behebung von Entwurfsfehlern, (b) eine schnelle Entwurfsraumexploration, als auch (c) eine gezielte Optimierung zur Einhaltung von Zielvorgaben und somit eine signifikante Verkürzung der Entwurfszeit. Die Beiträge im Bereich der Schnittstellensynthese erlauben die automatisierte Integration von generierten Hardware-Realisierungen in FPGA-basierte SoCs für (a) die rechnergestützte Bewertung und Verifikation der generierten IP-Cores als Teil eines Hardware/Software-Systems als auch (b) den Test der Hardware–Komponenten als Teil des Zielsystems, um auf Systemebene getroffene Entwurfsentscheidungen zu validieren. Die Beiträge dieser Arbeit bieten in diesem Sinne

einen ganzheitlichen Ansatz für die schnelle Prototyperstellung wodurch von einem erheblichen Produktivitätsgewinn ausgegangen werden kann. Obwohl sich die Arbeit auf den Bereich der medizinischen Bildverarbeitung konzentriert, können die Beiträge jedoch auch auf viele weitere Anwendungsbereiche übertragen werden.

Bibliography

[ACD74] T. Adam, K. M. Chandy, and J. R. Dickson. "A comparison of list schedules for parallel processing systems". In: *Communications of the ACM* 17.12 (Dec. 1974), pp. 685–690. ISSN: 0001-0782.

[Alt15a] Altera Corp. *Altera's User-Customizable ARM-based SoC*. 2015. URL: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/br/br-soc-fpga.pdf (visited on 05/03/2015).

[Alt15b] Altera Corp. *Nios II Processor*. 2015. URL: http://www.altera.com/products/processors/overview.html (visited on 05/03/2015).

[Alt15c] Altera, Corp. *PCI Express*. 2015. URL: https://www.altera.com/solutions/technology/transceiver/protocols/pro-pci_exp.html (visited on 03/19/2015).

[Alt13] Altium Ltd. *C-to-Hardware Compiler*. 2013. URL: http://techdocs.altium.com/display/AEE/Introduction+to+C-to-Hardware+Compilation+Technology+in+Altium+Designer (visited on 01/20/2015).

[Amd67] G. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the Spring Joint Computer Conference (AFIPS)*. (Atlantic City, NJ, USA). Apr. 18, 1967, pp. 483–485.

[ARM14] ARM, Ltd. *AMBA Specifications*. 2014. URL: http://www.arm.com/products/system-ip/amba-specifications.php (visited on 03/12/2015).

[AN08] Arvind and R. Nikhil. "Hands-on Introduction to Bluespec System Verilog (BSV)". In: *Proceedings of the 6th International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*. (Anaheim, CA, USA). June 5–7, 2008, pp. 205–206.

[Asa+06] L. Asanovic, R. Bodik, B. Catancaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick. *The Landscape of Parallel Computing Research: A view from Berkely*. Tech. rep. UCB/EECS-2006-183. University of California at Berkeley, Dec. 2006.

[AL96] M. Aubury and W. Luk. "Binomial filters". In: *Journal of VLSI signal processing systems for signal, image and video technology* 12.1 (Jan. 1996), pp. 35–50. ISSN: 0922-5773.

[BMP07] B. Bailey, G. Martin, and A. Piziali. *ESL Design and Verification*. Elsevier, Inc., 2007. ISBN: 0-12-373551-5.

[Bai11] D. Bailey. *Design for embedded image processing on FPGAs*. John Wiley & Sons, Inc., 2011. ISBN: 978-0470828496.

[BS73] M. Barbacci and D. Siewiorek. "Automated Exploration of the Design Space for Register Transfer (RT) Systems". In: *SIGARCH Computer Architecture News* 2.4 (Dec. 1973), pp. 101–106. ISSN: 0163-5964.

[BDQ98] F. Bassetti, K. Davis, and D. Quinlan. "Optimizing Transformations of Stencil Operations for Parallel Object-Oriented Scientific Frameworks on Cache-Based Architectures". In: *Computing in Object-Oriented Parallel Environments*. Vol. 1505. Lecture Notes in Computer Science (LNCS). Springer Berlin Heidelberg, 1998, pp. 107–118. ISBN: 978-3-540-65387-5.

[Bas04] C. Bastoul. "Code Generation in the Polyhedral Model Is Easier Than You Think". In: *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT)*. (Antibes Juan-les-Pins, France). Sept. 29, 2004–Oct. 3, 2003, pp. 7–16.

[Bau+03] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt. "PACT XPP—A Self-Reconfigurable Data Processing Architecture". In: *The Journal of Supercomputing* 26.2 (2003), pp. 167–184. ISSN: 0920-8542.

[Ber+95] R. Bergamaschi, R. O'Connor, L. Stok, M. Moricz, S. Prakash, A. Kuehlmann, and D. Rao. "High-level synthesis in an industrial environment". In: *IBM Journal of Research and Development* 39.1.2 (Jan. 1995), pp. 131–148. ISSN: 0018-8646.

[Bie+93] J. Biesenack, M. Koster, A. Langmaier, S. Ledeux, S. Marz, M. Payer, M. Pilsl, S. Rumler, H. Soukup, N. Wehn, and P. Duzy. "The Siemens high-level synthesis system CALLAS". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems (TVLSI)* 1.3 (Sept. 1993), pp. 244–253. ISSN: 1063-8210.

[BHT14] S. Boppu, F. Hannig, and J. Teich. "Compact Code Generation for Tightly-Coupled Processor Arrays". In: *Journal of Signal Processing Systems* 77.1–2 (Oct. 2014), pp. 5–29. ISSN: 1939-8018.

[Bre72] M. Breuer. "Recent Developments in Design Automation". In: *Computer* 5.3 (May 1972), pp. 23–35. ISSN: 0018-9162.

[BR96] S. Brown and J. Rose. "FPGA and CPLD architectures: A tutorial". In: *IEEE Design & Test of Computers* 13.2 (Summer 1996), pp. 42–57. ISSN: 0740-7475.

[BAS03] R. Budruk, D. Anderson, and T. Shanley. *PCI Express System Architecture*. MindShare, Inc., 2003. ISBN: 0-321156307.

[Cad11] Cadence Design Systems. *C-to-Silicon Compiler High-Level Synthesis*. 2011. URL: http://www.cadence.com/rl/Resources/datasheets/C2Silicon_ds.pdf (visited on 01/20/2015).

[Cal14] Calypto Design. *Catapult: Product Family Overview*. 2014. URL: http://calypto.com/en/products/catapult/overview (visited on 01/20/2015).

[Can+11] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. Anderson, S. Brown, and T. Czajkowski. "LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems". In: *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*. (Monterey, CA, USA). Feb. 27, 2011, pp. 33–36.

[Cas+13] V. G. Castellana, V. G. Castellana, F. Ferrandi, and F. Ferrandi. "An automated flow for the High Level Synthesis of coarse grained parallel applications". In: *Proceedings of the International Conference on Field-Programmable Technology (FPT)*. (Kyoto, Japan). Dec. 9–11, 2013, pp. 294–301.

[Cha11] V. A. Chandrasetty. *VLSI Design – A Practical Guide for FPGA and ASIC Implementations*. Springer Science+Business Media, 2011. ISBN: 978-1461411208.

[CBA13] J. Choi, S. Brown, and J. Anderson. "From software threads to parallel hardware in high-level synthesis for FPGAs". In: *Proceedings of the International Conference on Field-Programmable Technology (FPT)*. (Kyoto, Japan). Dec. 9–11, 2013, pp. 270–277.

[Chu06] P. Chu. *RTL Hardware Design using VHDL*. John Wiley & Sons, Inc., 2006. ISBN: 0-471-72092-5.

[Com90] R. Composano. "Path-based scheduling for synthesis". In: *Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences*. (Kailua-Kona, HI, USA). Jan. 2–5, 1990, 348–355 vol.1.

[Cou+08] P. Coussy, C. Chavet, P. Bomel, D. Heller, E. Senn, and E. Martin. "GAUT: A High-Level Synthesis Tool for DSP Applications". In: *High-Level Synthesis*. Springer Netherlands, 2008, pp. 147–169. ISBN: 978-1-4020-8587-1.

[DJ80] J. Darringer and W. Joyner. "A New Look at Logic Synthesis". In: *Proceedings of the 17th Design Automation Conference (DAC)*. (San Francisco, CA, USA). June 23, 1980, pp. 543–549.

[De +86] H. De Man, J. Rabaey, P. Six, and L. Claesen. "Cathedral-II: A Silicon Compiler for Digital Signal Processing". In: *IEEE Design & Test of Computers* 3.6 (Dec. 1986), pp. 13–25. ISSN: 0740-7475.

[De 94] G. De Micheli. *Syntesis and Optimization of Digital Circuits*. McGraw-Hill Education, 1994. ISBN: 978-0071132718.

[DL95] E. Dougherty and P. Laplante. *Introduction to Real-Time Imaging*. Wiley-IEEE Press, 1995. ISBN: 978-0819417893.

[DC98] A. Downton and D. Crookes. "Parallel architectures for image processing". In: *Electronics & Communication Engineering Journal* 10.3 (June 1998), pp. 139–151. ISSN: 0954-0695.

[Du+13] Z. Du, X. Li, X. Yang, and K. Shen. "A Parallel Multigrid Poisson PDE Solver for Gigapixel Image Editing". In: *High Performance Computing*. Vol. 207. Communications in Computer and Information Science. Springer Berlin Heidelberg, 2013, pp. 89–98. ISBN: 978-3-642-41590-6.

[Edw06] S. Edwards. "The Challenges of Synthesizing Hardware from C-Like Languages". In: *IEEE Design & Test of Computers* 23.5 (May 2006), pp. 375–386. ISSN: 0740-7475.

[Ell99] J. Elliott. *Understanding Behavioral Synthesis: A Practical Guide to High-Level Design*. Springer US, 1999. ISBN: 978-1-4615-5059-4.

[Fib14] Fibre Channel Industry Association. *GEN 6 - Sixth Generation Fibre Channel*. 2014. URL: http://fibrechannel.org/library/2014/03/gen-6---sixth-generation-fibre-channel.html (visited on 03/19/2015).

[For15] Forte Designs Systems. *Cynthesizer*. 2015. URL: http://www.fortends.com/products/cynthesizer.asp (visited on 01/20/2015).

[FVP06] F. Franchetti, Y. Voronenko, and M. Püschel. "A Rewriting System for the Vectorization of Signal Transforms". In: *High Performance Computing for Computational Science (VECPAR)*. (Rio de Janeiro, Brazil). June 10–13, 2006, pp. 363–377.

[Fre96] M. Freidlin. "Diffusion Processes and PDE's in Narrow Branching Tubes". In: *Markov Processes and Differential Equations*. Lectures in Mathematics ETH Zürich. Birkhäuser Basel, 1996, pp. 79–89. ISBN: 978-3-7643-5392-6.

[FY69] T. Friedman and S.-C. Yang. "Methods Used in an Automatic Logic Design Generator (ALERT)". In: *IEEE Transactions on Computers* C-18.7 (July 1969), pp. 593–614. ISSN: 0018-9340.

[FY70] T. Friedman and S.-C. Yang. "Quality of designs from an automatic logic generator (ALERT)". In: *Proceedings of the 7th Design Automation Workshop (DAC)*. (San Francisco, California, USA). June 22–25, 1970, pp. 71–89.

[Ful05] S. Fuller. *RapidIO The Embedded System Interconnect*. John Wiley & Sons, Ltd., 2005. ISBN: 0-470092912.

[GK83] D. Gajski and R. Kuhn. "Guest Editors' Introduction: New VLSI Tools". In: *Computer* 16.12 (Dec. 1983), pp. 11–14. ISSN: 0018-9162.

[Geo+13] N. George, D. Novo, T. Rompf, M. Odersky, and P. Ienne. "Making domain-specific hardware synthesis tools cost-efficient". In: *Proceedings of the International Conference on Field-Programmable Technology (FPT)*. (Kyoto, Japan). Dec. 9–11, 2013, pp. 120–127.

[Gos05] A. Goshtasby. *2-D and 3-D Image Registration: for Medical, Remote Sensing, and Industrial Applications*. John Wiley & Sons, Inc., 2005. ISBN: 978-0471649540.

[Gup+03] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. "SPARK: A high-level synthesis framework for applying parallelizing compiler transformations". In: *Proceedings of the 16th International Conference on VLSI Design (VLSID)*. (Washington, DC, USA). Jan. 4–8, 2003, pp. 461–466.

[GL91] B. Gustafsson and P. Lötstedt. "Analysis of multigrid methods for general systems of PDE". In: *Multigrid Methods III*. Vol. 98. International Series of Numerical Mathematics / Internationale Schriftenreihe zur Numerischen Mathematik / Série Internationale d'Analyse Numérique. Birkhäuser Basel, 1991, pp. 223–234. ISBN: 978-3-0348-5714-7.

[GOW07] M. Gustlin, F. Olsson, and M. Weber. *Interlaken Technology: New-Generation Packet Interconnect*. 2007. URL: https://www.cortina-systems.com/images/documents/400023_Interlaken_Technology_White_Paper.pdf (visited on 05/03/2015).

[HKZ14] F. Hannig, D. Koch, and D. Ziener, eds. *Proceeding of First International Workshop on FPGAs for Software Programmers (FSP 2014)*. Sept. 1, 2014. arXiv: 1408.4423 [cs.AR].

[Han+14] F. Hannig, V. Lari, S. Boppu, A. Tanase, and O. Reiche. "Invasive Tightly-Coupled Processor Arrays: A Domain-Specific Architecture/Compiler Co-Design Approach". In: *ACM Transactions on Embedded Computing Systems (TECS)* 13.4s (Apr. 2014), 133:1–133:29.

[Han+08] F. Hannig, H. Ruckdeschel, H. Dutta, and J. Teich. "PARO: Synthesis of Hardware Accelerators for Multi-dimensional Dataflow-Intensive Applications". In: *Reconfigurable Computing: Architectures, Tools and Applications*. Lecture Notes in Computer Science (LNCS). Springer Berlin Heidelberg, Jan. 2008, pp. 287–293. ISBN: 978-3-540-78609-2.

[Han09] F. Hannig. "Scheduling Techniques for High-Throughput Loop Accelerators". Verlag Dr. Hut, Munich, Germany. Dissertation. University of Erlangen-Nuremberg, Germany, Aug. 11, 2009. 307 pp. ISBN: 978-3-86853-220-3.

[HS88] C. Harris and M. Stephens. "A Combined Corner and Edge Detector". In: *Proceedings of the 4th Alvey Vision Conference*. (London, United Kingdom). Aug. 31–Sept. 2, 1988, pp. 147–151.

[Hau+07] C. Haubelt, J. Falk, J. Keinert, T. Schlichter, M. Streubühr, A. Deyhle, A. Hadert, and J. Teich. "A SystemC-based Design Methodology for Digital Signal Processing Systems". In: *EURASIP Journal on Embedded Systems* 2007.1 (Jan. 2007), pp. 15–37. ISSN: 1687-3955.

[Hem+94] A. Hemani, B. Karlsson, M. Fredriksson, K. Nordqvist, and B. Fjellborg. "Application of high-level synthesis in an industrial project". In: *Proceedings of the Seventh International Conference on VLSI Design*. (Calcutta, India). Jan. 5–8, 1994, pp. 5–10.

[Hen03] J. Henkel. "Closing the SoC design gap". In: *Computer* 36.9 (Sept. 2003), pp. 119–121. ISSN: 0018-9162.

[HP14] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. 5th. Morgan Kaufmann Publishers, Inc., 2014. ISBN: 978-0124077270.

[HTA08] B. Holden, J. Trodden, and D. Anderson. *HyperTransport 3.1 Interconnect Technology*. MindShare, Inc., 2008. ISBN: 978-0977087822.

[Hor+08] A. Hormati, M. Kudlur, S. Mahlke, D. Bacon, and R. Rabbah. "Optimus: Efficient Realization of Streaming Applications on FPGAs". In: *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*. (Atlanta, GA, USA). Oct. 19, 2008, pp. 41–50.

[Hum77] R. Hummel. "Image enhancement by histogram transformation". In: *Computer Graphics and Image Processing* 6.2 (Apr. 1977), pp. 184–195. ISSN: 0146-664X.

[Hwa+93] D. Hwang, S. Cho, Y. Kim, and S. Han. "Exploiting Spatial and Temporal Parallelism in the Multithreaded Node Architecture Implemented on Superscalar RISC Processors". In: *Proceedings of the International Conference on Parallel Processing (ICPP)*. (Syracuse, NY, USA). Aug. 16–20, 1993, pp. 51–54.

[IEE04] IEEE and IEC. "IEC/IEEE Behavioural Languages - Part 4: Verilog Hardware Description Language (Adoption of IEEE Std 1364-2001)". In: *IEC 61691-4 First edition 2004-10; IEEE 1364* (2004), 0_1–860.

[IEE15] IEEE Computer Society. *IEEE 802.3: ETHERNET*. 2015. URL: http://standards.ieee.org/about/get/802/802.3.html (visited on 05/03/2015).

[IEE96] IEEE Computer Society. *IEEE Std. 1596.3-1996, IEEE Standard for Low-Voltage Differential Signals (LVDS) for Scalable Coherent Interface (SCI)*. 1996. URL: http://standards.ieee.org/findstds/standard/1596.3-1996.html (visited on 05/03/2015).

[IEE09] IEEE Computer Society and Design Automation Standards Subcommittee and Institute of Electrical and Electronics Engineers and IEEE Standards Board and IEEE Standards Association. "IEEE Standard VHDL Language Reference Manual". In: *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)* (Jan. 2009), pp. c1–626.

[ISO14] ISO. *ISO/IEC 14882:2014: Information technology — Programming languages — C++*. International Organization for Standardization, 2014.

[ISO11] ISO. *ISO/IEC 9899:2011: Information technology — Programming languages — C*. International Organization for Standardization, 2011.

[Jai+89] R. Jain, K. Kucukcakar, M. Mlinar, and A. Parker. "Experience with the ADAM Synthesis System". In: *Proceedings of the 26th ACM/IEEE Design Automation Conference (DAC)*. (Las Vegas, NV, USA). June 1, 1989, pp. 56–61.

[JG93] A. W. Johnson and M. Graham. *High-Speed Digital Design – A Handbook of Black Magic*. Prenctice Hall, 1993. ISBN: 978-0133957242.

[KM12] N. Kavvadias and K. Masselos. "Automated Synthesis of FSMD-Based Accelerators for Hardware Compilation". In: *Proceedings of the 23rd International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*. (Delft, The Netherlands). July 9–11, 2012, pp. 157–160.

[Keh10] N. Kehtarnavaz. *Real-Time Image and Video Processing*. SPIE Press, 2010. ISBN: 978-0819481979.

[Kna96] D. Knapp. *Behavioral Synthesis: Digital System Design Using the Synopsys Behavioral Compiler*. Prentice-Hall, Inc., 1996. ISBN: 0-13-569252-0.

[Knu98] D. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. 2nd. Addison Wesley Longman Publishing Co., Inc., 1998. ISBN: 0-201-89685-0.

[KT11] D. Koch and J. Torresen. "FPGASort: A High Performance Sorting Architecture Exploiting Run-time Reconfiguration on FPGAs for Large Problem Sorting". In: *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*. (Monterey, CA, USA). Feb. 27, 2011, pp. 45–54.

[Kro98] S. Krolikoski. "The V-synth system". In: *Digest of Papers of the Thirty-Third IEEE Computer Society International Conference (CompCon)*. (San Francisco, CA, USA). Feb. 29–Mar. 3, 1998, pp. 328–331.

[KD90a] D. Ku and G. De Micheli. "High-level synthesis and optimization strategies in Hercules and Hebe". In: *Proceedings of Euro ASIC '90*. (Paris, France). May 29–June 1, 1990, pp. 124–129.

[KD90b] D. Ku and G. De Micheli. "Relative scheduling under timing constraints". In: *Proceedings of the 27th ACM/IEEE Design Automation Conference (DAC)*. (Orlando, FL, USA). June 24–28, 1990, pp. 59–64.

[Kuc+98] K. Kucukcakar, C.-T. Chen, J. Gong, W. Philipsen, and T. Tkacik. "Matisse: An architectural design tool for commodity ICs". In: *IEEE Design & Test of Computers* 15.2 (Apr. 1998), pp. 22–33. ISSN: 0740-7475.

[KP87] F. Kurdahi and A. Parker. "REAL: A Program for REgister ALlocation". In: *Proceedings of the 24th ACM/IEEE Design Automation Conference (DAC)*. (Miami Beach, Florida, USA). May 28–June 1, 1987, pp. 210–215.

[KC05] P. Kwan and C. Clarke. "FPGAs for Improved Energy Efficiency in Processor Based Systems". In: *Advances in Computer Systems Architecture*. Vol. 3740. Lecture Notes in Computer Science (LNCS). Springer Berlin Heidelberg, 2005, pp. 440–449. ISBN: 978-3-540-29643-0.

[KSP15] N. Kyrtatas, D. Spampinato, and M. Püschel. "A Basic Linear Algebra Compiler for Embedded Processors". In: *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*. (Grenoble, France). Mar. 9–12, 2015, p. 6.

[Lai+93] A. Laine, S. Song, J. Fan, W. Huda, J. Honeyman, and B. Steinbach. "Adaptive multiscale processing for contrast enhancement". In: *Proceedings SPIE1905, Biomedical Image Processing and Biomedical Visualization)*. (San Jose, CA, USA). Jan. 31, 1993.

[Lam88] M. Lam. "Software Pipelining: An Effective Scheduling Technique for VLIW Machines". In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI)*. (Atlanta, GA, USA). June 1, 1988, pp. 318–328.

[Lip+91] P. Lippens, J. Van Meerbergen, A. van der Werf, W. Verhaegh, B. McSweeney, J. Huisken, and O. McArdle. "PHIDEO: A silicon compiler for high speed algorithms". In: *Proceedings of the European Conference on Design Automation (EDAC)*. (Amsterdam, The Netherlands). Feb. 25, 1991–Feb. 28, 1992, pp. 436–441.

[Mei+03] S. Mei B.and Vernalde, D. Verkest, H. De Man, and R. Lauwereins. "ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix". In: *Field Programmable Logic and Application*. Vol. 2778. Lecture Notes in Computer Science (LNCS). Springer Berlin Heidelberg, 2003, pp. 61–70. ISBN: 978-3540408222.

[Mem+14] R. Membarth, O. Reiche, F. Hannig, and J. Teich. "Code generation for embedded heterogeneous architectures on android". In: *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*. (Dresden, Germany). Mar. 24–28, 2014, pp. 1–6.

[Mem+15a] R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Körner, and W. Eckert. "HIPAcc: A Domain-Specific Language and Compiler for Image Processing". In: *IEEE Transactions on Parallel and Distributed Systems (TPDS)* PP.99 (Jan. 2015), p. 14. ISSN: 1045-9219.

[Mem+15b] R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Körner, and W. Eckert. "Hipacc: A Domain-Specific Language and Compiler for Image Processing". In: *IEEE Transactions on Parallel and Distributed Systems (TPDS)* PP.99 (2015), pp. 1–14. ISSN: 1045-9219.

[Men+09] R. Menotti, J. Cardoso, M. Fernandes, and E. Marques. "LALP: A Novel Language to Program Custom FPGA-Based Architectures". In: *Proceedings of the 21st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. (Sao Paulo, Brazil). Oct. 28–31, 2009, pp. 3–10.

[Men15] Mentor Graphics. *Functional Verification – Modelsim*. 2015. URL: http://www.mentor.com/products/fv/modelsim/ (visited on 03/27/2015).

[MD96] E. Mirsky and A. DeHon. "MATRIX: A reconfigurable computing architecture with configurable instruction distribution and deployable resources". In: *Proceedings of the 1996 IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*. (Napa Valley, CA, USA). Apr. 17–19, 1996, pp. 157–166.

[Moo65] G. Moore. "Cramming More Components onto Integrated Circuits". In: *Proceedings of the IEEE* 38.8 (Apr. 1965), pp. 82–85. ISSN: 0018-9219.

[Ney93] F. Neycenssaci. "Contrast Enhancement Using the Laplacian-of-a-Gaussian Filter". In: *Graphical Models and Image Processing (CVGIP)* 55.6 (Nov. 1993), pp. 447–463. ISSN: 1049-9652.

[Ofe+13] G. Ofenbeck, T. Rompf, A. Stojanov, M. Odersky, and M. Püschel. "Spiral in Scala: Towards the Systematic Construction of Generators for Performance Libraries". In: *Proceedings of the 12th International Conference on Generative Programming: Concepts &*

Experiences (GPCE). (Indianapolis, Indiana, USA). Oct. 27, 2013, pp. 125–134.

[Pan+05] P. Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh. "Performance evaluation and design trade-offs for network-on-chip interconnect architectures". In: *IEEE Transactions on Computers* 54.8 (Aug. 2005), pp. 1025–1040. ISSN: 0018-9340.

[PP88] N. Park and A. Parker. "Sehwa: A software package for synthesis of pipelines from behavioral specifications". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 7.3 (Mar. 1988), pp. 356–370. ISSN: 0278-0070.

[PPM86] A. Parker, J. Pizarro, and M. Mlinar. "MAHA: A Program for Datapath Synthesis". In: *Proceedings of the 23rd ACM/IEEE Design Automation Conference (DAC)*. (Las Vegas, NV, USA). June 29–July 2, 1986, pp. 461–466.

[PK89] P. Paulin and J. Knight. "Force-directed scheduling for the behavioral synthesis of ASICs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 8.6 (June 1989), pp. 661–679. ISSN: 0278-0070.

[PKG86] P. Paulin, J. Knight, and E. Girczyc. "HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis". In: *Proceedings of the 23rd ACM/IEEE Design Automation Conference (DAC)*. (Las Vegas, NV, USA). June 29–July 2, 1986, pp. 263–270.

[Piz+87] S. Pizer, E. Amburn, J. Austin, R. Cromartie, A. Geselowitz, T. Greer, B. Romeny, and J. Zimmerman. "Adaptive Histogram Equalization and Its Variations". In: *Computer Vision, Graphics, and Image Processing* 39.3 (Sept. 1987), pp. 355–368. ISSN: 0734-189X.

[Pre63] R. Press. "R63-37 A Logic Design Translator". In: *IEEE Transactions on Electronic Computers* EC-12.2 (Apr. 1963), pp. 157–157. ISSN: 0367-7508.

[PM92] J. Prince and E. McVeigh. "Motion estimation from tagged MR image sequences". In: *IEEE Transactions on Medical Imaging (T-MI)* 11.2 (June 1992), pp. 238–249. ISSN: 0278-0062.

[Pul+12] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov. "Real-time Computer Vision with OpenCV". In: *Communications of the ACM* 55.6 (June 2012), pp. 61–69. ISSN: 0001-0782.

[Püs+05] M. Püschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo. "SPIRAL: Code Generation for DSP Transforms". In: *Proceedings of the IEEE* 93.2 (Feb. 2005), pp. 232–275. ISSN: 0018-9219.

[Rap09] RapidIO Trade Association. *RapidIO Specification Revision 2.1*. 2009. URL: http://www.rapidio.org/files/rev2.1_spec_stack.zip (visited on 05/03/2015).

[Rap14] RapidIO Trade Association. *RapidIO Specifications*. 2014. URL: http://www.rapidio.org/rapidio-specifications (visited on 03/19/2015).

[RJ99] N. Ratha and A. Jain. "Computer vision algorithms on reconfigurable logic arrays". In: *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 10.1 (Jan. 1999), pp. 29–43. ISSN: 1045-9219.

[Sch64] H. Schorr. "Computer-Aided Digital System Design and Analysis Using a Register Transfer Language". In: *IEEE Transactions on Electronic Computers* EC-13.6 (Dec. 1964), pp. 730–737. ISSN: 0367-7508.

[Shi83] S. Shiva. "Automatic hardware synthesis". In: *Proceedings of the IEEE* 71.1 (Jan. 1983), pp. 76–87. ISSN: 0018-9219.

[SB76] D. Siewiorek and M. Barbacci. "The CMU RT-CAD System: An Innovative Approach to Computer Aided Design". In: *Proceedings of the National Computer Conference (NCC)*. (New York, NY, USA). June 7–10, 1976, pp. 643–655.

[ST82] K. Stüben and U. Trottenberg. "Multigrid methods: Fundamental algorithms, model problem analysis and applications". In: *Multigrid Methods*. Vol. 960. Lecture Notes in Mathematics. Springer Berlin Heidelberg, 1982, pp. 1–176. ISBN: 978-3-540-11955-5.

[Syn15] Synopsis Inc. *Synphony Model Compiler*. 2015. URL: http://www.synopsys.com/Tools/Implementation/RTLSynthesis/Pages/SynphonyC-Compiler.aspx (visited on 01/20/2015).

[Tei97] J. Teich. *Digitale Hardware/Software Systeme*. 1st. Springer-Verlag Berlin Heidelberg, 1997. ISBN: 3-540624333.

[Tei00] J. Teich. "Embedded System Synthesis and Optimization". In: *Proceedings of the Workshop on System Design Automation (SDA)*. (Rathen, Germany). Mar. 13–14, 2000, pp. 9–22.

[TH07] J. Teich and C. Haubelt. *Digitale Hardware/Software Systeme: Synthese und Optimierung*. 2nd. Springer-Verlag Berlin Heidelberg, 2007. ISBN: 978-3540468226.

[THM06] M. Telgarsky, J. Hoe, and J. Moura. "Spiral: Joint Runtime and Energy Optimization of Linear Transforms". In: *Proceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. (Toulouse, France). May 14, 2006–June 19, 2005, pages.

[Tex11] Texas Instruments, Inc. *Using the UCD92xx Digital Point-of-Load Controller*. 2011. URL: http://www.ti.com/lit/ug/sluu490/sluu490.pdf (visited on 05/03/2015).

[The15] The MathWorks, Inc. *MATLAB*. 2015. URL: http://www.mathworks.com/products/matlab (visited on 05/03/2015).

[TBU00] P. Thevenaz, T. Blu, and M. Unser. "Interpolation revisited". In: *IEEE Transactions on Medical Imaging* 19.7 (July 2000), pp. 739–758. ISSN: 0278-0062.

[TKA02] W. Thies, M. Karczmarek, and S. Amarasinghe. "StreamIt: A Language for Streaming Applications". In: *Compiler Construction*. Vol. 2304. Lecture Notes in Computer Science (LNCS). Springer Berlin Heidelberg, Mar. 2002, pp. 179–196. ISBN: 978-3-540-43369-9.

[Tho+88] D. E. Thomas, E. M. Dirkes, R. A. Walker, J. V. Rajan, J. A. Nestor, and R. L. Blackburn. "The System Architect's Workbench". In: *Proceedings of the 25th ACM/IEEE Design Automation Conference (DAC)*. (Atlantic City, New Jersey, USA). June 1, 1988, pp. 337–343.

[TGP07] J. Tripp, M. Gokhale, and K. Peterson. "Trident: From High-Level Language to Hardware Circuitry". In: *Computer* 40.3 (Mar. 2007), pp. 28–37. ISSN: 0018-9162.

[Van94] E. Van de Velde. "Poisson Solvers". In: *Concurrent Scientific Computing*. Vol. 16. Texts in Applied Mathematics. Springer New York, 1994, pp. 183–216. ISBN: 978-1-4612-6921-2.

[Vil+10] J. Villarreal, A. Park, W. Najjar, and R. Halstead. "Designing Modular Hardware Accelerators in C with ROCCC 2.0". In: *Proceedings of the 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. (Charlotte, NC, USA). May 2–4, 2010, pp. 127–134.

[WO00] K. Wakabayashi and T. Okamoto. “C-based SoC design flow and EDA tools: an ASIC and system vendor perspective”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 19.12 (Dec. 2000), pp. 1507–1522. ISSN: 0278-0070.

[WC91a] R. Walker and R. Camposano. “Honeywell’s V-Synth System”. In: *A Survey of High-Level Synthesis Systems*. Vol. 135. The Springer International Series in Engineering and Computer Science. Springer US, 1991, pp. 89–91. ISBN: 978-1-4613-6772-7.

[WC91b] R. Walker and R. Camposano. “IBM’s Yorktown Silicon Compiler”. In: *A Survey of High-Level Synthesis Systems*. Vol. 135. The Springer International Series in Engineering and Computer Science. Springer US, 1991, pp. 100–105. ISBN: 978-1-4613-6772-7.

[Wan+14] C. Wang, F.-L. Yuan, T.-H. Yu, and D. Markovic. “27.5 A multi-granularity FPGA with hierarchical interconnects for efficient and flexible mobile computing”. In: *Proceedings of the IEEE International Solid-State Circuits Conference - Digest of Technical Papers*. (San Francisco, CA, USA). Feb. 9–13, 2014, pp. 460–461.

[WF83] A. Widmer and P. Franaszek. “A DC-Balanced, Partitioned-Block, 8B/10B Transmission Code”. In: *IBM Journal of Research and Development* 27.5 (Sept. 1983), pp. 440–451. ISSN: 0018-8646.

[Wol89] M. Wolfe. “More Iteration Space Tiling”. In: *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*. (Reno, NV, USA). Jan. 1, 1989, pp. 655–664.

[Xil14a] Xilinx, Inc. *7 Series FPGAs Integrated Block for PCI Express*. PG054. Nov. 2014.

[Xil14b] Xilinx, Inc. *7 Series FPGAs Overview*. 2014. URL: http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf (visited on 05/03/2015).

[Xil13] Xilinx, Inc. *Accelerating OpenCV Application with Zynq-7000 All Programmable SoC using Vivado HLS Video Libraries*. 2013. URL: http://www.xilinx.com/support/documentation/application_notes/xapp1167.pdf (visited on 05/03/2015).

[Xil15a] Xilinx, Inc. *AMBA AXI4 Interface Protocol*. 2015. URL: http://www.xilinx.com/ipcenter/axi4.htm (visited on 03/12/2015).

[Xil11a] Xilinx, Inc. *LogiCORE IP Serial RapidIO v5.6 User Guide*. 2011. URL: http://www.xilinx.com/support/documentation/ip_documentation/srio_ug503.pdf (visited on 05/03/2015).

[Xil15b] Xilinx, Inc. *MicroBlaze Soft Processor Core*. 2015. URL: http://www.xilinx.com/tools/microblaze.htm (visited on 05/03/2015).

[Xil11b] Xilinx, Inc. *Virtex-6 FPGA GTX Transceivers User Guide*. 2011. URL: http://www.xilinx.com/support/documentation/user_guides/ug366.pdf (visited on 05/03/2015).

[Xil11c] Xilinx, Inc. *Virtex-II Pro and Virtex-II Pro X Platfrom FPGAs: Complete Data Sheet*. 2011. URL: http://www.xilinx.com/support/documentation/data_sheets/ds083.pdf (visited on 05/03/2015).

[Xil15c] Xilinx, Inc. *Vivado High-Level Synthesis*. 2015. URL: http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html (visited on 01/20/2015).

[Xil11d] Xilinx, Inc. *XPower Estimator User Guide*. 2011. URL: http://www.xilinx.com/support/documentation/user_guides/ug440.pdf (visited on 05/03/2015).

[Xil14c] Xilinx, Inc. *Zynq-7000 All Programmable SoC Overview*. 2014. URL: http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf (visited on 05/03/2015).

[Yas+87] F. Yassa, J. Jasica, R. Hartley, and S. Noujaim. "A silicon compiler for digital signal processing: Methodology, implementation, and applications". In: *Proceedings of the IEEE* 74.9 (Sept. 1987), pp. 1272–1282. ISSN: 0018-9219.

[Zha+08] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong. "AutoPilot: A Platform-Based ESL Synthesis System". In: *High-Level Synthesis*. Springer Netherlands, 2008, pp. 99–112. ISBN: 978-1-4020-8587-1.

[Zim77] G. Zimmermann. *Report on the computer architecture design language MIMOLA: Machine Independent Microprogramming Language*. Tech. rep. 4/77. Universität Kiel, Apr. 1977.

[Zuo+13] W. Zuo, P. Li, D. Chen, L.-N. Pouchet, S. Zhong, and J. Cong. "Improving Polyhedral Code Generation for High-level Synthesis". In: *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. (Montreal,Canada). Sept. 29–Oct. 4, 2013.

Author's Own Publications

[Sch+15a*] M. Schmid, O. Reiche, F. Hannig, and J. Teich. "Loop Coarsening in C-based HLS". In: *Proceedings of the 26th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. (Toronto, Canada). July 27–29, 2015, pp. 166–173.

[Sch+15b*] C. Schmitt, M. Schmid, F. Hannig, J. Teich, S. Kuckuk, and H. Köstler. "Generation of Multigrid-based Numerical Solvers for FPGA Accelerators". In: *Proceedings of the 2nd International Workshop on High-Performance Stencil Computations (HiStencils)*. (Amsterdam, The Netherlands). Ed. by A. Größlinger and H. Köstler. Jan. 20, 2015, pp. 9–15.

[Rei+14*] O. Reiche, M. Schmid, F. Hannig, R. Membarth, and J. Teich. "Code Generation from a Domain-specific Language for C-based HLS of Hardware Accelerators". In: *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. (New Dehli, India). Oct. 12–17, 2014.

[Sch+14a*] M. Schmid, N. Apelt, F. Hannig, and J. Teich. "An Image Processing Library for C-based High-Level Synthesis". In: *Proceedings of the 24th International Conference on Field Programmable Logic and Applications (FPL)*. (Munich, Germany). Sept. 2–4, 2014.

[Sch+14c*] M. Schmid, O. Reiche, C. Schmitt, F. Hannig, and J. Teich. "Code Generation for High-Level Synthesis of Multiresolution Applications on FPGAs". In: *Proceedings of the First International Workshop on FPGAs for Software Programmers (FSP)*. (Munich, Germany). Sept. 1, 2014, pp. 21–26. arXiv: 1408.4721 [cs.CV].

[Sch+14d*] M. Schmid, A. Tanase, V. Bhadouria, F. Hannig, J. Teich, and D. Ghoshal. "Domain-Specific Augmentations for High-Level Synthesis". In: *Proceedings of the 25th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. (Zurich, Switzerland). June 18–20, 2014.

[Sch+14b*] M. Schmid, F. Hannig, A. Tanase, and J. Teich. "High-Level Synthesis Revised – Generation of FPGA Accelerators from a Domain-Specific Language using the Polyhedron Model". In: *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*. Vol. 25. Advances in Parallel Computing. Amsterdam, The Netherlands: IOS Press, Apr. 2014, pp. 497–506. ISBN: 978-1-61499-380-3.

[Sch+13*] M. Schmid, M. Blocherer, F. Hannig, and J. Teich. "Real-Time Range Image Preprocessing on FPGAs". In: *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig)*. (Cancun, Mexico). Dec. 9–11, 2013.

[Han+13*] F. Hannig, M. Schmid, V. Lari, S. Boppu, and J. Teich. "System Integration of Tightly-Coupled Processor Arrays using Reconfigurable Buffer Structures". In: *Proceedings of the ACM International Conference on Computing Frontiers (CF)*. (Ischia, Italy). May 14–16, 2013.

[Lar+12*] V. Lari, S. Muddasani, S. Boppu, F. Hannig, M. Schmid, and J. Teich. "Hierarchical Power Management for Adaptive Tightly-Coupled Processor Arrays". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 18.1 (Dec. 2012), 2:1–2:25. ISSN: 1084-4309.

[SHT12*] M. Schmid, F. Hannig, and J. Teich. "Power Management Strategies for Serial RapidIO Endpoints in FPGAs". In: *Proceedings of the 20th Annual IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. (Toronto, Canada). Apr. 29–May 1, 2012, pp. 101–108.

[Was+11*] J. Wasza, S. Bauer, S. Haase, M. Schmid, S. Reichert, and J. Hornegger. "RITK: The Range Imaging Toolkit - A Framework for 3-D Range Image Stream Processing". In: *VMV*. Ed. by P. Eisert, J. Hornegger, and K. Polthier. Eurographics Association, 2011, pp. 57–64. ISBN: 978-3-905673-85-2.

[Han+10*] F. Hannig, M. Schmid, J. Teich, and H. Hornegger. "A Deeply Pipelined and Parallel Architecture for Denoising Medical Images". In: *Proceedings of the IEEE International Conference on Field Programmable Technology (FPT)*. Beijing, China: IEEE, Dec. 8–10, 2010, pp. 485–490.

[Dut+10*] H. Dutta, F. Hannig, M. Schmid, and J. Keinert. "Modeling and Synthesis of Communication Subsystems for Loop Accelerator Pipelines". In: *Proceedings of the 21st IEEE International Conference on Application-specific Systems, Architectures, and Processors*

(ASAP). Rennes, France: IEEE Computer Society, July 7–9, 2010, pp. 125–132.

[Sch+10*] M. Schmid, F. Hannig, J. Teich, R. Diefenbach, H. Pettendorf, and H. Hornegger. "Discourse on Extending Embedded Medical Image Processing Systems Using the High Speed Serial RapidIO Interconnect". In: *Proceedings of the Embedded World Conference*. Nuremberg, Germany, Mar. 2010, pp. 1–9.

[ZST10*] D. Ziener, M. Schmid, and J. Teich. "Robustness Analysis of Watermark Verification Techniques for FPGA Netlist Cores". In: *Design Methodologies for Secure Embedded Systems*. Ed. by A. Biedermann and H. G. Molter. Vol. 78. Lecture Notes in Electrical Engineering. Springer, Berlin, 2010, pp. 105–127.

[SZT08*] M. Schmid, D. Ziener, and J. Teich. "Netlist-Level IP Protection by Watermarking for LUT-Based FPGAs". In: *Proceedings of IEEE International Conference on Field-Programmable Technology (FPT)*. Taipei, Taiwan, Dec. 8–10, 2008, pp. 209–216.

List of Acronyms

AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
AMBA	Advanced Microcontroller Bus Architecture
ASIC	Application-Specific Integrated Circuit
ASP	Application-Specific Processor
AXI4S	AXI4-Stream
AXI4	Advanced eXtensible Interface 4
BAR	Base Address Register
BRAM	Block Random Access Memory
CAD	Computer Aided Design
CDC	Clock Domain Crossing
CGRA	Coarse-Grained Reconfigurable Array
CLB	Configurable Logic Block
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DAG	Directed Acyclic Graph
DA	Design Automation
DDR3	Double Data Rate Type Three
DFG	Data Flow Graph
DLL	Data Link Layer
DLP	Data-Level Parallelism
DMA	Direct Memory Access
DPRAM	Dual-Ported Random Access Memory
DSE	Design Space Exploration
DSL	Domain-Specific Language
DSP	Digital Signal Processor
EDA	Electronic Design Automation
eGPU	embedded GPU

ESL	Electronic System-Level
FD	finite differences
FF	Flipflop
FIFO	First In, First Out
FMA	Fused Multiply-Add
FPGA	Field Programmable Gate Array
FSMD	Finite State Machine with Data Path
FSM	Finite State Machine
gcc	GNU Compiler Collection
GPGPU	General Purpose Computation on Graphics Processing Unit
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HLL	High-Level Programming Language
HLS	High-Level Synthesis
HPC	High-Performance Computing
HSCD	Hardware/Software Co-Design
IC	Integrated Circuit
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
II	Initiation Interval
ILP	Instruction-Level Parallelism
I/O	Input/Output
IP	Intellectual Property
LMS	Lightweight Modular Staging
LoC	Lines of Code
LOG	Logical Layer
LSI	Large Scale Integration
LUT	Lookup Table
LVDS	Low Voltage Differential Signaling
MAC	Multiplier-Accumulator
MGT	Multi-Gigabit Transceiver
MIMO	Multiple Input, Multiple Output
MMCM	Mixed-Mode Clock Manager
MPSoC	Multi-Processor System-on-a-Chip
MRA	Multiresolution Analysis
MSI	Medium Scale Integration
MTU	Maximum Transfer Unit

NoC	Network-on-Chip
NRE	Non-Recurring Engineering
OCI	On-Chip Interconnect
OEM	Original Equipment Manufacturer
OOC	Out-Of-Context
OpenCL	Open Computing Language
OpenCV	Open Source Computer Vision
PCIe	Peripheral Component Interconnect Express
PCS	Physical Coding Sublayer
PDE	Partial Differential Equation
PDU	Protocol Data Unit
PMU	Power Management Unit
PHY	Physical Layer
PLL	Phase-Locked Loop
PMA	Physical Media Attachment
PPnR	Post Place and Route
QoR	Quality of Results
RAM	Random Access Memory
RGBA	Red Green Blue Alpha
RHS	Right-Hand Side
RIO	RapidIO
ROM	Read Only Memory
RTL	Register Transfer Level
RT	Register Transfer
SDF	Synchronous Data Flow
SRAM	Static Random Access Memory
SDRAM	Synchronous Dynamic Random Access Memory
SDU	Service Data Unit
SerDes	Serializer/Deserializer
SIMD	Single Instruction, Multiple Data
SoC	System-on-a-Chip
SRIO	Serial RapidIO
SSI	Small Scale Integration
TAL	Transaction Layer
TCPA	Tightly-Coupled Processor Array
TLP	Transaction Layer Packet
TRA	Transport Layer

TTL	Transistor-Transistor Level
VHDL	Very high speed Hardware Description Language
VLSI	Very Large Scale Integration