Chapter 1
STHEM: Productive Implementation of High-Performance Embedded Image Processing Applications

Magnus Jahre

Abstract Developing embedded image processing systems is challenging, and one key reason is that they face a rich set of (conflicting) constraints. The challenge is further exacerbated by the need to deliver the product to the market as soon as possible. This paper presents the STHEM tool-chain which was developed during the TULIPP project to address this issue. STHEM is a collection of productivity enhancing tools that help developers rapidly implement an image processing system that satisfies all constraints. Currently, STHEM consists of six different utilities that enable extensive analysis of performance and energy consumption, provide access to highly efficient image processing functions, or help developers leverage advanced hardware features.

1.1 Introduction

Building embedded image processing systems is challenging as developers face a rich set of conflicting constraints including high performance, limited power dissipation as well as size and weight restrictions. These constraints commonly force image processing systems to become heterogeneous. More specifically, the key performance-critical parts of the application typically needs to be offloaded to specialised hardware units, commonly called accelerators [3], to enable delivering sufficient performance while staying within the power budget.

Embedded image processing application development is therefore heavily tied to the hardware platform it will be deployed on. Further, it is critically important that the developer can easily track down the root cause of performance problems, and developers typically rely on performance analysis tools to do this. To abstract away the hardware platform dependencies, we introduce the *Generic Heterogeneous*

Magnus Jahre
Norwegian University of Science and Technology, Trondheim, Norway,
e-mail: magnus.jahre@ntnu.no

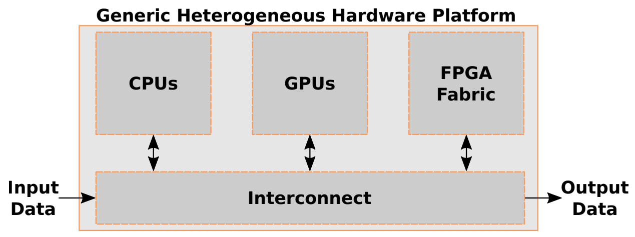

Fig. 1.1 A Generic Heterogeneous Hardware Platform (GHHP).

Hardware Platform (GHHP) (see Figure 1.1). It contains a collection of compute resources (i.e., CPUs, GPUs and an FPGA fabric) as well as an interconnection network and input/output devices. In addition, the GHHP typically contains local and global memory structures that may or may not be exposed to the developer.

At a high level, the performance analysis tools need to capture two classes of information to help the developer identify performance problems in a GHHP:

- **Inter-compute unit efficiency:** Mapping different parts of the application to the different compute units available in the hardware platform is necessary to fully leverage the capabilities of a heterogeneous platform, and performance analysis tools need to provide feedback on the quality of this mapping. Identification of bottleneck compute units is especially important. Common techniques for achieving this is to profile the application on each compute unit and present an aggregated profile to the developer.
- **Intra-compute unit efficiency:** A performance bottleneck may also be due to an inefficient implementation within a single compute unit. In this case, the developer needs to map the performance problem to the source code responsible for creating it. To achieve this, we need performance analysis tools that can pinpoint performance problems with profiling and automatically relate these to source code constructs.

One of the key objectives of the TULIPP project [12] was to contribute to making the process of developing embedded image processing systems more efficient. We addressed this problem by devising a tool-chain, called STHEM [21], that aims to reduce the time it takes to implement an image processing application that satisfies all requirements. STHEM is an acronym for Supporting uTilities for Heterogeneous EMbedded image processing platforms. At the end of the TULIPP project, we released STHEM under an open-source license on GitHub[1].

STHEM ensures that the developer can focus on core application development by automating recurring, but critical, tasks such as instrumenting code to gather performance profiles, design space exploration, and vendor tool configuration. Thus, our definition of the word tool-chain includes any tool that improves the efficiency of image processing application development. We use also use the term performance in a broad sense to cover key metrics such as runtime, energy dissipation, or power

[1] `https://github.com/tulipp-eu/sthem`

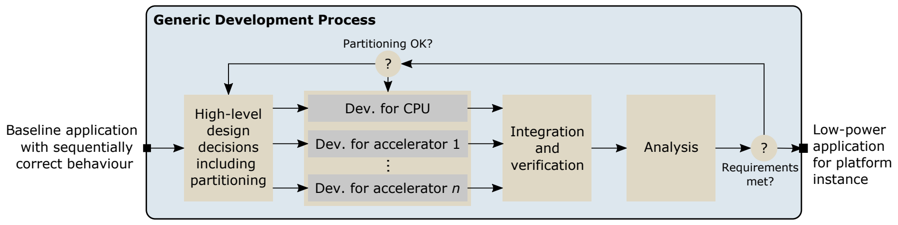
Generic Development Process
Partitioning OK?
?
Baseline application with sequentially correct behaviour
High-level design decisions including partitioning
Dev. for CPU
Dev. for accelerator 1
Dev. for accelerator n
Integration and verification
Analysis
?
Requirements met?
Low-power application for platform instance

Fig. 1.2 The Generic Development Process (GDP)

consumption. For image processing systems, requirements are often specified in terms of target frame rates or the maximum acceptable latency from frame arrival until processing is complete.

Tools aiming at improving development productivity are not the only tools necessary for supporting the full project life-cycle. In addition, tools are necessary to, for instance, support regression tests, simulation, version control, configuration handling, and bug tracking. We found that the existing state-of-the-art tools include decent support for such processes and provide options to embed third-party mechanisms for missing features. Therefore, STHEM mainly focuses on developer productivity.

1.2 The Generic Development Process (GDP)

Reaching the performance potential of the hardware platform requires adapting an image processing algorithm to leverage the characteristics of the hardware components as well as making a number of application-dependent trade-offs. To efficiently support this procedure, we propose the Generic Development Process (GDP) as shown in Figure 1.2. GDP is an iterative process that generalises the approach taken by programmers when implementing highly efficient image processing applications [21].

The starting point of GDP is the baseline application that executes with correct sequential behaviour on a modern machine with a general-purpose processor. This is the initial development step for most image processing systems – ensuring that all the functions of the system are fully understood. Although this is a critical step, substantial effort is commonly needed to move the system onto the embedded platform.

High-level partitioning decides which baseline functions should be accelerated and how. Partitioning splits off into accelerator-specific development stages that later join to produce an integrated application with the same correct behaviour as the baseline. In some cases, application behaviour can be modified compared to the baseline if this gives a substantial performance advantage on the target platform while still providing acceptable accuracy. The performance of the integrated application is checked against requirements. If found lacking, the partitioning and development stages are restarted. In this manner, programmers iteratively refine the baseline application to approach the required power consumption and performance. The purpose of the tool-chain is to minimise the number of iterations required and the time spent in each iteration before arriving at an implementation that meets requirements.

In the most basic case, GDP can be carried out manually without any software support. This will likely result in low developer productivity for complex applications since GDP is reduced to a trial-and-error process. Furthermore, the developer will typically spend considerable effort in developing support code for identifying performance problems. Thus, a better approach is to create a tool-chain – such as STHEM – that enables efficiently carrying out the iterations of GDP on the chosen hardware platform.

1.3 Realising the Generic Development Process

We have now described GDP, and we now delve into how to realise GDP for a particular embedded system application. A critical focus point will be how to enable accelerating the performance critical application regions as acceleration is commonly necessary to meet the stringent performance, energy, or power consumption requirements. Within an organisation, it is typically favourable to chose one (or a small number) of development processes to build expertise and limit the overhead of maintaining multiple tool-chains.

There are primarily three main decisions that need to be made when implementing a GDP-based process:

1. What are the key performance-relevant characteristics of the targeted hardware platform?
2. How should the application be implemented on the targeted hardware platform? (See Section 1.3.1).
3. Which tools should be used to assess the performance characteristics of the application? (See Section 1.3.2).

Since GDP assumes that development targets a known hardware platform, we focus on the two last questions in this paper. The main reason is that hardware platform selection typically needs to address the full range of system requirements including size and weight in addition the the aforementioned performance criteria.

We use the term *implementation approach* in a broad manner to capture the high-level methodology used to implement an application including choosing programming language(s) and programming model(s) as well as the degree of automation versus manual effort. Since GDP is iterative, performance analysis tools are critical as they (i) direct the developer's focus towards key performance issues, and (ii) document which aspects of the application have sufficiently high performance and hence do not require further work.

Table 1.1 Advantages and challenges of the single- and multi-language strategies.

Single-Language Strategy	Advantages	• Setup is simpler since a single tool chain can be used for the complete application. • Application maintenance is simplified due to a single code-base and single tool-chain. • The abstractions employed to support multiple different computing units tend to result in less code being necessary to implement the application.
	Challenges	• All platform components need to support the chosen programming language. This may limit hardware platform options. • The higher level of abstraction may limit the achievable performance and energy efficiency.
Multi-Language Strategy	Advantages	• Using specialised vendor tools for each component reduces the risk of introducing performance-limiting abstractions. • Platform selection is simplified since vendors can support different programming languages.
	Challenges	• Application maintenance and integration is complicated by multiple tool-chains, especially due to upgrades. • Development is more difficult since the company needs to recruit and retain people that are experts in each programming language. • Efficient communication mechanisms and interfaces between the parts of the application that are realised in different programming languages needs to be designed, implemented, and verified.

1.3.1 Selecting the Implementation Approach

A key high-level decision when defining an implementation approach is to select the programming language(s) to use. Note that the choice of programming language is separate from that of choosing a programming model as the programming model expresses an execution model in addition to the semantics of the programming language. A straightforward example is OpenMP [6] where the programming language is typically C++ but a parallel execution model is provided in addition to the sequential execution model defined by the language's semantics.

Thus, a key high-level decision is whether to use a single programming language for the complete application (with its supported programming models) or to accept that different parts of the system is implemented with different programming languages. We refer to these strategies as a *Single-Language (SL)* strategy and a *Multi-Language* strategy, respectively. While applications for CPUs and GPUs commonly use a single-language approach (e.g., OpenMP [6] and CUDA [5]), FPGA-accelerated applications have traditionally used a multi-language approach with CPU-code implemented in C/C++ and the accelerator in Hardware Description Languages (HDLs) such as VHDL or Verilog.

Table 1.1 outlines the advantages and challenges of the SL and ML strategies. Overall, the SL-strategy simplifies the development process compared to the ML-strategy. However, the SL-strategy may limit the attainable performance and energy efficiency due to a higher abstraction level. In addition, the SL-strategy complicates platform selection since the preferred programming language needs to be efficiently supported on all platform components. Deciding which strategy to follow is a complex trade-off that depends on application requirements, hardware platform requirements as well as the expertise and strategic focus of the company.

1.3.1.1 Single-Language Approaches

Multi-threaded models. Current hardware platforms for image processing applications tend to contain multiple CPUs which means that the programmer may need to respond to the architectural challenges that can arise on multi-cores (e.g., [9, 10, 8]). Programming models for multi-cores has been studied extensively, and powerful tools such as OpenMP [6] are available to efficiently parallelise an application using task-based or data-parallel strategies. An added benefit of using tools such as OpenMP is the existence of advanced performance analysis strategies [18, 16].

SIMD-based models. In these models, the part of the application that will be offloaded to the accelerator is rewritten as a kernel that performs the desired processing on a subset of the application's input data – a Single Instruction Multiple Data (SIMD) approach. In this way, the runtime can invoke a large number of threads – some of which are executed in lock-step – to exploit parallelism at a comparatively low hardware cost. Each thread is assigned a (possibly multi-dimensional) identifier which the kernel uses to select its input data. One example of this model is OpenCL [23] which supports a range of devices – including GPUs and FPGAs. For platforms that include only GPUs and CPUs, NVIDIA CUDA [5] is another example.

The SIMD-based models are desirable because they leverage a familiar parallel programming abstraction – i.e., Single-Program Multiple Data (SPMD) – and are supported by a rich ecosystem of tools. OpenCL [23] is a standard API that enables program execution on a GHHP containing hardware components such as CPUs, GPUs, and other accelerators. It provides an abstraction layer where each computational device (e.g., a GPU) is composed of one or more compute units (e.g., processor cores). These units are again subdivided into SIMD processing elements. The task of the developer is to formulate the program in a data- or task-parallel manner to use the computational resources available in the platform.

Although OpenCL guarantees that a program will run correctly on all OpenCL-supported platforms, platform-specific optimisation is commonly necessary to achieve high performance and energy efficiency. Further, some FPGA vendors support OpenCL on selected FPGA platforms, but it can be challenging to determine the root cause of performance problems since the OpenCL compute model does not map straightforwardly to the FPGA substrate [27]. In addition, performance problems can occur when a multi-dimensional memory access pattern aligns unfavourably with the underlying hardware organisation [17].

High-Level Synthesis (HLS). The abstractions of OpenCL may limit implementation flexibility on hardware platforms that contain reconfigurable fabrics such as FPGAs. An alternative approach is HLS where the application is implemented in a high-level language (commonly C or C++), and an HLS-tool is used to automatically generate an accelerator for a selected code segment (e.g., a function). HLS is a viable design alternative due to the existence of multiple commercial and academic tools (e.g., Xilinx Vivado HLS [29], LegUP [4], Bambu [19], and uIR [22]).

There are two important challenges when using HLS. First, the tools only support a subset of the high-level language which commonly means that the code needs to be modified to enable HLS. Second, the relationship between the high-level code formulation and the generated hardware is not always obvious which complicates performance analysis.

Library-based acceleration. In this approach, the programmer uses standard libraries such as OpenCV [11] or OpenVX [7] to implement performance critical image processing kernels. Thus, the programmer does not need to deal with the characteristics of the accelerators and simply leverages optimised implementations provided by the libraries. However, the programmer is limited to the functionality supported by the libraries, and a high-performance implementation of the preferred library must be available on the chosen platform. Similar approaches are methodologies such as DCMI [14] which automatically generate application-specific accelerator hardware for performance-critical kernels.

A library-based strategy is a great option when the functions of the library map well to the performance critical regions of the application. If it does not, it is unlikely that a library-based approach will yield competitive performance. A common situation is that the invocation of separate kernels lead to excessive copying of data – resulting in excessive overhead that outweighs the performance improvement gained by accelerating the function.

Transparent acceleration. Transparent acceleration strategies aim at accelerating applications without programmer intervention. In other words, they aim to completely automate GDP. To achieve this, they first profile the reference application to identify the key performance-critical function(s). Then, they analyse and optimise the performance-critical functions(s) at the level of the compiler Intermediate Representation (IR) and finally map these functions to a target accelerator. Transparent acceleration approaches are currently research prototypes and not sufficiently mature to be used for industrial application development.

Although transparent acceleration is an attractive concept, it is very challenging to realise. The state-of-the-art approach is Needle [15] which identifies collections of hot program paths (called Braids) within a performance critical function and then speculatively offloads these to a reconfigurable accelerator. If the application diverts from the accelerated paths during execution, any performed changes are rolled back and the procedure is executed on the CPU. A key challenge is to achieve sufficient coverage of the application such that the benefits of offloading outweighs the overheads.

1.3.1.2 Multi-Language Approaches

While CPU and GPU programming models tend to favour a single-language approach, FPGA-targeted development has traditionally used RTL languages such as VHDL or Verilog to describe the accelerator (see e.g., [26]) and C/C++ for the CPU code. This typically results in development of the FPGA functionality being performed independently of the CPU code after an initial interface specification. Thus, the FPGA/CPU partitioning of the system is performed early in the project, based on limited performance analysis data, and typically cannot be reversed without incurring significant costs.

VHDL and Verilog require repeatedly specifying low-level implementation details such as the width of signals. Thus, development using VHDL or Verilog tends to be time-consuming. An alternative approach is to use high-productivity RTL languages such as Chisel [1]. These languages improve productivity by (i) not requiring to repeatedly specify all implementation details, and (ii) providing powerful, reusable constructs. In contrast to HLS-tools, the developer still specifies the concrete structure of the hardware. Thus, high-productivity RTL languages provide higher productivity while enabling the developer to specify most implementation details.

Overall, an SL-strategy is generally preferable compared to an ML-strategy as it enables iterative performance-data-driven acceleration. Further, it is (much) easier to modify the application (e.g., if requirements change during a project) when the application is implemented in a single language. An important exception is for (extremely) performance-sensitive components as an RTL-level implementation may be necessary to achieve sufficient performance in this case.

1.3.2 Selecting and Evaluating Performance Analysis Tools

The implementation approach and hardware platform determines an application's attainable performance and energy efficiency while the capabilities of the performance analysis tools determine how productively an application that meets requirements can be developed. In other words, the existence of efficient performance analysis tools is a secondary concern. It is not useful to quickly develop a solution with an implementation approach that cannot meet performance and energy efficiency requirements.

The performance analysis tool availability can only impact the choice of implementation approach when there are multiple options that can meet requirements. In this case, the performance analysis tools can be evaluated on their ability to (i) efficiently detect performance problems, (ii) relate the performance problem to source code construct that caused it, and (iii) provide suggestions or solutions to how the performance problem can be alleviated.

Efficient performance problem detection tends to require some form of application profiling combined with high-level visualisations such as Gantt charts or Grain Graphs [18]. With appropriate mechanisms, the visualisations can automatically zoom in on problematic sections and thereby significantly simplify performance problem detection.

By leveraging the debug information available in the application binary, it is possible to map a performance problem to a specific source code location. By externally sampling the CPU program counter, it is possible to implement a similar strategy to relate instantaneous power measurements to source code constructs (see Chapter **??**).

Providing analysis functions that can automatically solve performance problems is a challenging research problem. Thus, solving problems tends to be the responsibility of the application developer. A different approach is restricting the formulation of programs such that performance problems are less likely to occur (e.g., [13, 20]). Another class of approaches can avoid some platform-specific performance issues by conducting extensive design-space exploration to ensure that implementation details are chosen to arrive at a high-performance design point (e.g., [30]). An interesting compromise is to explore semi-automatic approaches where a tool provides suggestions on how a performance problem can be dealt with and the developer leverages domain knowledge to choose the exact strategy.

1.4 STHEM: The TULIPP Tool-Chain

The previous sections discussed development and analysis of embedded system applications in a general sense. In this section, we provide a concrete tool-chain example by introducing the collection of performance- and productivity-enhancing utilities that have been developed during the TULIPP project. More specifically, we explain the choices made to support GDP on the TULIPP hardware platforms [24, 25]. Overall, we follow an HLS-based Single-Language strategy and use a combination of state-of-the-art vendor tools and novel research-based utilities. We also support library-based acceleration since this strategy is very efficient when the functionality of the library is a good fit for the needs of the application.

STHEM [21] contains two types of tool packages:

- **Vendor Tools (VTs):** VTs are existing, industry-grade tool packages that are critical to enable GDP on a particular platform instance, and they are commonly supplied by platform vendors or third-party companies. For the TULIPP platforms, examples of VTs are the Xilinx SDSoC development [28] environment and the HIPPEROS real-time operating system. VT are commonly large and complex, and it is both infeasible and inefficient to not use them when they are available.
- **Utilities:** Utilities are smaller tool packages provided by the TULIPP project that are designed to resolve limitations that hamper developer productivity on a particular platform. To facilitate reuse across platforms, the utilities are designed to be as independent of the VT as possible and are often stand-alone. A utility may consist of a single hardware or software tool, or a collection of several tools working together to provide a certain functionality.

Table 1.2 The STHEM utilities and their key benefits.

Utility	Key Benefits
PMU	• Reduces the time spent establishing system power and energy consumption by providing power profiles with high temporal and high spatial resolution. • Automatically correlates power samples with program counter values non-intrusively retrieved from the platform CPUs.
AU	• Reduces the time it takes the developer to identify performance and power consumption issues by visualising the performance and power profiles collected by the PMU. • Reduces development time with automatic Design Space Exploration (DSE) of HLS configurations – in contrast to time-consuming manual exploration of configuration options.
DPRU	• Reduces development time by enabling using HLS in systems that require dynamically reconfiguring the FPGA. Concretely, it enables using dynamic partial reconfiguration with SDSoC.
HiFlipVX	• Reduces development time by adding optimised FPGA-enabled implementations of commonly used image processing functions.
IIU	• Reduces development time by including support for cameras that support the CameraLink interface. • Reduces development time by readily supporting HDMI input and output.
FDU	• Provides lossless stream of signal values to facilitate on-FPGA accelerator debugging.

The objective of STHEM is to provide efficient support for GDP on the Tulipp platform. To reach this objective, we leverage VTs when they are available to ensure that the baseline tool-chain is comparable to current state-of-the-art tool-chains. Within the Tulipp project, we identified cases where executing GDP is unnecessarily cumbersome and developed utilities to address these productivity issues.

The core of STHEM is the Xilinx SDSoC development environment [28] which provides support for accelerating specific functions within an application with Xilinx Vivado HLS [29]. However, it is generally challenging to manually use HLS to accelerate applications functions. First, it is difficult – and thereby time-consuming – to establish which functions to accelerate. Second, it is also difficult to develop an HLS-based implementation that has sufficient performance, acceptable power consumption, and does not need more computational resources than are available in the chosen FPGA [2]. Third, developing high-performance I/O-controllers for commonly used high resolution camera and display interfaces can be challenging.

STHEM contains the following utilities that help alleviate these challenges:

- **The Power Measurement Utility (PMU)** gathers power consumption samples with high spatial (i.e., high sample rate) and temporal resolution (up to seven concurrent inputs) and relates them to application behaviour through non-intrusively sampling the program counter of the CPUs.
- **The Analysis Utility (AU)** enables visual and automatic analysis of application performance and power consumption based on the profiles collected by the PMU.

- **The Dynamic Partial Reconfiguration Utility (DPRU)** enables using dynamic partial reconfiguration within SDSoC.
- **HiFlipVX** includes highly optimized image processing functions that developers can use to transparently accelerate their image processing application.
- **The I/O IP Utility (IIU)** includes IP-cores for camera input over the Camera Link interface as well as for HDMI input and output.
- **The FPGA Debug Utility (FDU)** provides a lossless stream of signal data to simplify online FPGA debugging.

Collectively, the STHEM utilities improve support for GDP on the TULIPP platforms by streamlining the process of developing an image processing system that meets performance and power constraints, and Table 1.2 summarises the key benefits of each utility. We showcase the capabilities of selected STHEM utilities later in this book. More specifically, selected features of the PMU and AU are described in Chapter **??** while Chapter **??** covers HiFlipVX.

1.5 Conclusion

We have now presented the STHEM tool-chain developed during the TULIPP project. STHEM is a collection of utilities that work alongside vendor tools with the overall objective of reducing the time it takes a developer to implement an embedded image processing application that satisfies all constraints. Although we have had a particular focus on the TULIPP platforms, we have taken care to keep the utilities as general as possible to simplify porting to other platforms.

Acknowledgements We would like to thank all of the people that worked on STHEM during the TULIPP project as STHEM is indeed a highly collaborative effort. This work has been funded in part by the European Horizon 2020 project TULIPP (grant agreement #688403).

References

1. Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., Asanović, K.: Chisel: Constructing hardware in a Scala embedded language. In: Proceedings of the Annual Design Automation Conference (DAC), pp. 1216–1225 (2012)
2. Bacon, D.F., Rabbah, R., Shukla, S.: FPGA programming for the masses. Commun. ACM **56**(4), 56–63 (2013)
3. Borkar, S., Chien, A.A.: The future of microprocessors. Communications of the ACM **54**(5), 67 (2011)
4. Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Czajkowski, T., Brown, S.D., Anderson, J.H.: LegUp: An Open-source High-level Synthesis Tool for FPGA-based Processor/Accelerator Systems. ACM Trans. Embed. Comput. Syst. **13**(2), 24:1–24:27 (2013)
5. Cook, S.: CUDA programming: A developer's guide to parallel computing with GPUs. Newnes (2012)

6. Dagum, L., Menon, R.: OpenMP: An industry standard API for shared-memory programming. IEEE Computational Science and Engineering **5**(1), 46–55 (1998)
7. Giduthuri, R., Pulli, K.: OpenVX: A framework for accelerating computer vision. In: SIGGRAPH ASIA 2016 Courses, SA '16, pp. 1–50 (2016)
8. Jahre, M., Eeckhout, L.: GDP: Using dataflow properties to accurately estimate interference-free performance at runtime. In: International Symposium on High Performance Computer Architecture (HPCA), pp. 296–309 (2018)
9. Jahre, M., Grannaes, M., Natvig, L.: A quantitative study of memory system interference in chip multiprocessor architectures. In: 11th IEEE International Conference on High Performance Computing and Communications (HPCC) (2009)
10. Jahre, M., Natvig, L.: A high performance adaptive miss handling architecture for chip multiprocessors. Transactions on High-Performance Embedded Architectures and Compilers IV **6760** (2011)
11. Kaehler, A., Bradski, G.: Learning OpenCV 3: Computer vision in C++ with the OpenCV library. "O'Reilly Media, Inc." (2016)
12. Kalb, T., Kalms, L., Göhringer, D., Pons, C., Marty, F., Muddukrishna, A., Jahre, M., Kjeldsberg, P.G., Ruf, B., Schuchert, T., Tchouchenkov, I., Ehrenstrahle, C., Christensen, F., Paolillo, A., Lemer, C., Bernard, G., Duhem, F., Millet, P.: TULIPP: Towards ubiquitous low-power image processing platforms. In: Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), pp. 306–311 (2016)
13. Koeplinger, D., Delimitrou, C., Prabhakar, R., Kozyrakis, C., Zhang, Y., Olukotun, K.: Automatic Generation of Efficient Accelerators for Reconfigurable Hardware. In: Proceedings of the International Symposium on Computer Architecture (ISCA), pp. 115–127 (2016)
14. Koraei, M., Fatemi, O., Jahre, M.: DCMI: A scalable strategy for accelerating iterative stencil loops on FPGAs. ACM Transactions on Architecture and Code Optimization **16**(4), 36:1–36:24 (2019)
15. Kumar, S., Sumner, N., Srinivasan, V., Magrem, S., Shriraman, A.: Needle: Leveraging program analysis to analyze and extract accelerators from whole programs. In: Proceedings of the International Symposium on High Performance Computer Architecture (HPCA) (2017)
16. Langdal, P.V., Jahre, M., Muddukrishna, A.: Extending OMPT to Support Grain Graphs. In: International Workshop on OpenMP (IWOMP), Lecture Notes in Computer Science, pp. 141–155 (2017)
17. Liu, Y., Zhao, X., Jahre, M., Wang, Z., Wang, X., Luo, Y., Eeckhout, L.: Get out of the valley: Power-efficient address mapping for GPUs. In: Proceedings of the International Symposium on Computer Architecture (ISCA) (2018)
18. Muddukrishna, A., Jonsson, P.A., Podobas, A., Brorsson, M.: Grain Graphs: OpenMP Performance Analysis Made Easy. In: Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP), pp. 1–13 (2016)
19. Pilato, C., Ferrandi, F.: Bambu: A modular framework for the high level synthesis of memory-intensive applications. In: International Conference on Field programmable Logic and Applications (FPL), pp. 1–4 (2013)
20. Prabhakar, R., Koeplinger, D., Brown, K.J., Lee, H., De Sa, C., Kozyrakis, C., Olukotun, K.: Generating Configurable Hardware from Parallel Patterns. In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 651–665 (2016)
21. Sadek, A., Muddukrishna, A., Kalms, L., Djupdal, A., Podlubne, A., Paolillo, A., Goehringer, D., Jahre, M.: Supporting utilities for heterogeneous embedded image processing platforms (STHEM): An overview. In: Applied Reconfigurable Computing (ARC) (2018)
22. Sharifian, A., Hojabr, R., Rahimi, N., Liu, S., Guha, A., Nowatzki, T., Shriraman, A.: uIR - An intermediate representation for transforming and optimizing the microarchitecture of application accelerators. In: Proceedings of the International Symposium on Microarchitecture (MICRO) (2019)
23. Stone, J.E., Gohara, D., Shi, G.: OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. Computing in Science & Engineering **12**(3), 66–73 (2010)

24. Sundance: PC/104 OneBank Board w. Xilinx Zynq Z7030 SoC FPGA. https://www.sundance.technology/som-cariers/pc104-boards/emc2-z7030/ (2018)
25. Sundance: PC/104 OneBank Board w. Zynq ZU4EV MPSoC FPGA. https://www.sundance.technology/som-cariers/pc104-boards/emc2-zu4ev/ (2018)
26. Umuroglu, Y., Jahre, M.: An energy efficient column-major backend for FPGA SpMV accelerators. In: Proceedings of the International Conference on Computer Design (ICCD), pp. 432–439 (2014)
27. Wang, Z., He, B., Zhang, W., Jiang, S.: A performance analysis framework for optimizing OpenCL applications on FPGAs. In: International Symposium on High Performance Computer Architecture (HPCA), pp. 114–125 (2016)
28. Xilinx: SDSoC development environment (2018). URL https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html
29. Xilinx: Vivado High-Level Synthesis (2018). URL https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html
30. Zhong, G., Prakash, A., Wang, S., Liang, Y., Mitra, T., Niar, S.: Design Space exploration of FPGA-based accelerators with multi-level parallelism. In: Design, Automation Test in Europe Conference Exhibition (DATE), pp. 1141–1146 (2017)