Pushing the Level of Abstraction of Digital System Design: a Survey on How to Program FPGAs

EMANUELE DEL SOZZO, DAVIDE CONFICCONI, ALBERTO ZENI, MIRKO SALARIS, DONATELLA SCIUTO, and MARCO D. SANTAMBROGIO, Politecnico di Milano, DEIB, Italy

Field Programmable Gate Arrays (FPGAs) are spatial architectures with a heterogeneous reconfigurable fabric. They are state-of-the-art for prototyping, telecommunications, embedded, and an emerging alternative for cloud-scale acceleration. However, FPGA adoption found limitations in their programmability and required knowledge. Therefore, researchers focused on FPGA abstractions and automation tools. Here, we survey three leading digital design abstractions: Hardware Description Languages (HDLs), High-Level Synthesis (HLS) tools, and Domain-Specific Languages (DSLs). We review these abstraction solutions, provide a timeline, and propose a taxonomy for each abstraction trend: programming models for HDLs; IP-based or System-based toolchains for HLS; application, architecture, and infrastructure domains for DSLs.

Additional Key Words and Phrases: Digital Design, Field Programmable Gate Array (FPGA), Hardware Description Languages (HDLs), High-Level Synthesis (HLS), Domain-Specific Languages (DSLs)

1 INTRODUCTION

Field Programmable Gate Arrays (FPGAs) are reconfigurable integrated circuits that contain a matrix of functional blocks interconnected together by a switching routing network [17, 29]. The central feature of an FPGA is its fabric, which is reprogrammable at the circuit level after manufacturing, giving the name *field-programmable*. Consequently, unlike Central Processing Units (CPUs) and Graphics Processing Units (GPUs), which rely on fixed data paths and topologies, an FPGA can implement different custom circuits according to the configuration and interconnection of its resources. For these reasons, in the past, FPGAs were mainly employed for Application Specific Integrated Circuit (ASIC) prototyping and networking. In the last decades, the internal complexity of FPGAs significantly increased, with single chips now containing hundreds of functionalities (both reconfigurable and hardwired), multi-die devices [23, 50, 155], different system-level interconnects [34, 164], and new device releases every year. Hence, academic researchers and companies started investing in FPGAs and adopting them also as hardware accelerators [30, 32, 44, 83, 135, 140–142, 150, 168, 175]. In this way, they can take advantage of a reconfigurable system able to efficiently implement various functionalities, providing a good trade-off between the flexibility of general-purpose CPUs and the performance and power efficiency of ASICs [29, 41, 54, 78, 168].

Despite the great opportunities, the fundamental drawback of FPGAs has always been their challenging design process, profoundly impacting their programmability and steeping the learning curve. The hardware design flow for FPGAs resembles the one available for ASICs (Physical Design block of Figure 1). Historically, the primary way to develop hardware design for FPGAs and ASICs consisted of using Hardware Description Languages (HDLs), especially Verilog and VHDL. Such low-level languages enable the description and definition of digital multi-signal circuits abstracting the behavior and structure, and they represent the standard for Register Transfer Level (RTL) design. Indeed, many industrial and commercial Electronic Design Automation (EDA) tools, like the ones by Xilinx (now part of AMD) [182], Synopsys [165], Intel [62], and Mentor Graphics [113], as well as open-source tools such as Symbiflow, take RTL description as an input and, from that, perform a sequence of steps towards the generation of the circuit. However, to efficiently leverage these

Authors' address: Emanuele Del Sozzo, emanuele.delsozzo@polimi.it; Davide Conficconi, davide.conficconi@polimi.it; Alberto Zeni, alberto.zeni@polimi.it; Mirko Salaris, mirko.salaris@polimi.it; Donatella Sciuto, donatella.sciuto@polimi.it; Marco D. Santambrogio, Politecnico di Milano, DEIB, via Ponzio 34/5, Milan, Italy, 20133, marco.santambrogio@polimi.it.

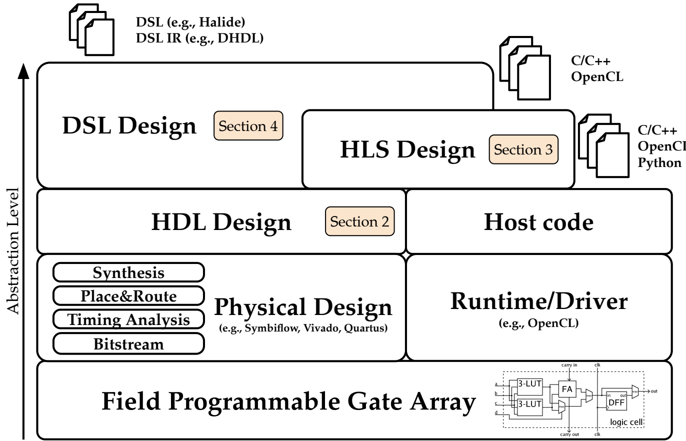

Fig. 1. FPGA Design Flow from source code down to FPGA configuration) with reference Sections numbered.

languages, the user requires significant knowledge and experience in hardware design. Besides, despite the architectural evolution of FPGAs, the features and abstractions offered by Verilog and VHDL did not evolve as fast. Consequently, even though hardware designers and EDA tools still use Verilog and VHDL, limitations like high verification effort and time-consuming/error-prone design make these languages less attractive than how they used to be. As a result, over the last years, new solutions have emerged to cope with the limitations of Verilog and VHDL.

In this scenario, we can identify three main categories of novel digital design abstractions: high-level HDLs, High-Level Synthesis (HLS) tools, and Domain-Specific Languages (DSLs), as the numbered blocks in Figure 1. Modern HDLs offer features and abstractions not available in Verilog and VHDL, like polymorphism, while still providing a design experience close to the hardware. HLS tools enable designers to rely on high-level languages, like C and OpenCL, to design hardware architectures. Finally, DSL tools represent newer trends to increase further productivity, performance, and ecosystem exploration beyond the HLS effort. Indeed, thanks to narrowing the computational domain, these tools can optimize the resulting design, while their respective languages significantly boost productivity and intuitively express the computation. Independently from the category, the goal of such tools is to: increase the level of abstraction and productivity for hardware design; enable high reuse and customization of IPs; reduce verification effort and design errors; make FPGAs accessible to a broader audience of users and developers. Consequently, these solutions are more appealing than Verilog and VHDL both to hardware designers, who want to evaluate different architectures quickly, and to application developers, who want to hardware accelerate applications, especially High-Performance Computing (HPC) ones [83, 135, 140, 191, 192]. In particular, HDLs incarnate a general-purpose solution to design whatever digital circuit[1], from processors to accelerators. On the other hand, modern HLS tools are more oriented to algorithm acceleration, but they also support the development of generic IPs, with some restrictions though. Finally, DSLs are the most specialized approach for domain-specific IPs, mainly accelerators.

This manuscript describes the research efforts on digital design abstractions for FPGA programming divided into *HDLs* (Section 2), *HLS* (Section 3), and *DSLs* (Section 4). Even though the literature already contains surveys on the theme of digital design abstractions for FPGAs, they

[1]We consider here neither analog nor mixed analog-digital circuit design.

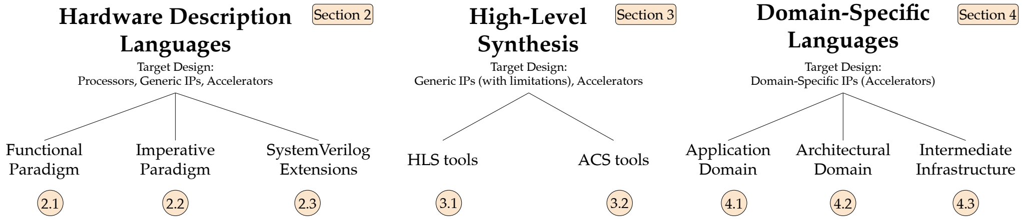

Fig. 2. Proposed taxonomy for the considered digital abstractions along with their target designs.

mainly focus on a single abstraction and its techniques [21, 36, 60, 81, 104, 108, 116, 170]. Thus, we believe that none of them comprehends an overall picture of the abstractions efforts in a unique survey proposing a taxonomy based on the trends and reporting a timeline. For other details on FPGA internal architecture and reconfigurability features, we point the interested reader to other surveys [17, 29, 79, 168, 177]. Within this survey, for each of the three abstraction efforts, we consider languages and tools that are still active or were born in the last ten years, to the best of our knowledge. Besides, we provide: a **timeline** (Figures 3, 5 and 6), reporting the first available dates of the tool appearance; a **taxonomy** (Figure 2), including the target designs and the trends we identified, such as Programming Model (Figure 3), Synthesis Target (Figure 4), or Target Domain (Figure 6); a **review of the main characteristics**[2] of each analyzed work (Tables 1 to 3). Finally, in Section 5, we give insights on possible future trends.

2 HARDWARE DESCRIPTION LANGUAGES

In this Section, we analyze various state-of-the-art HDLs. An HDL aims at describing the behavior of digital logic circuit designs both for ASICs and FPGAs. In the 1980s, VHDL and Verilog emerged as HDLs in literature introduced to help the electronic designers in the simulation and verification of Integrated Circuits (ICs), still needing the human for HDL to schematic translation [26, 67, 107, 124]. With the advent of logic synthesis and digital circuits' growth, EDA vendors pushed HDLs from just simulation and verification languages to design languages. However, for the most prominent HDLs from the 1980s (i.e., Verilog and VHDL), a significant portion of the language is not thought for synthesizing the circuit itself but for simulation purposes. Indeed, we speak of *synthesizable* or *non-synthesizable* constructs when speaking of VHDL and Verilog [26, 67, 107, 124]. Currently, VHDL and Verilog, now merged in SystemVerilog, are IEEE standards, part of commercial EDA tools, and the de facto standard for many FPGA and ASIC designers, though not the only alternative.

The impressive technology improvements increased the complexity of the hardware devices, e.g., in terms of logic gates and heterogeneity [17, 29, 107], demanding continuous research on tools able to handle such complexity and support the designers. Undoubtedly, both VHDL and SystemVerilog are evolving to keep up with technology improvements (e.g., VHDL-2008 and SystemVerilog itself). The interested reader can compare state-of-the-art surveys of the early 2000's [29, 79] with more recent ones [17, 168] to catch the device differences. Meanwhile, both academic and industrial users found limitations in the two standard HDLs and their productivity, hence proposing new programming paradigms. For this reason, this Section reviews emerging state-of-the-art HDLs.

Most of the emergent HDLs were born to increase productivity and keep up with the pace of technology development: from design to simulation and verification time, from using synthesizable HDL to complete system design, from code reuse to portability, from parametrization to generators.

[2] Appendix B and Table 4 define the technical terms reported throughout the survey.

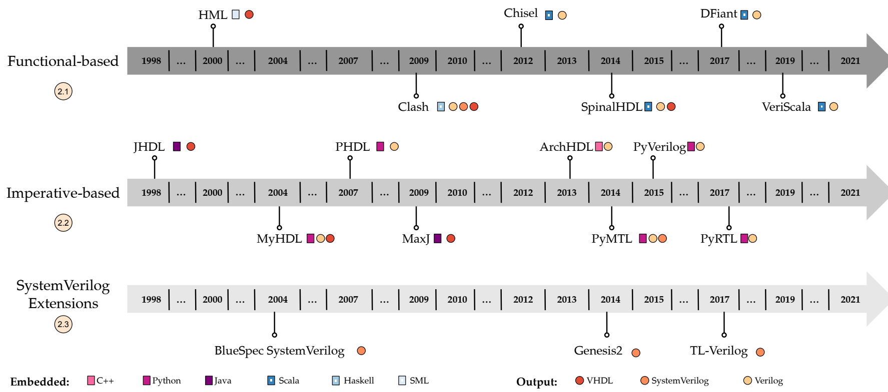

Fig. 3. HDL clustered by programming model with their embedded and output languages.

Some of these HDLs have their own syntax and constructs, while others are embedded in high-level languages like Scala and Python. Eventually, each of these HDLs translates into VHDL or (System)Verilog[3], as they remain the only languages supported by modern synthesis tools.

All these HDLs, along with HLS tools and DSLs, aim to boost the designer's productivity thanks to a higher level of abstraction, to hide complexity, and to reduce the time needed for hardware design. In this Section, we describe their main features, programming models, and their specific advantages or limitations to the best of our understanding and knowledge of each language. While programming models derive from the embedded language and advantages/limitations are language-dependant, the main features present the following structure. We highlight whether they support parametrization and polymorphism for high-level abstractions and if the HDL includes a built-in simulator. Moreover, we show other features to which an FPGA developer would have access whenever developing through VHDL or SystemVerilog, such as different clock domains, Double Data Rate (DDR) designs, digital verification (either functional or timing), and other IPs integration.

We propose an HDL taxonomy based on the characteristics of the programming model employed in the embedded languages. Our taxonomy has three main clusters: HDLs based on functional languages (such as Scala) (Section 2.1), HDLs based on imperative languages (such as C++) (Section 2.2), and SystemVerilog extensions (Section 2.3). Figure 3 shows the schema of how the HDLs analyzed fall in our taxonomy, their year of birth, and their embedded and output languages.

2.1 Functional-based HDL

The first category of HDLs derives from functional programming languages and embeds the characteristics of such a paradigm within the low-level hardware development flow. There are three different languages employed by HDL developers: SML [55], Haskell [6], and Scala [121].

While the first two languages are mainly devoted to functional features, Scala provides abstractions for object-oriented style, making it very appealing for new languages [5] and represents the current wave of functional-based HDLs. The great variety of abstractions and support from a growing community are essential factors that continuously push the development of these languages to embed low-level design features of native HDLs such as multi-clock domains.

HML: Hardware ML (HML) [95] is an HDL based on the functional language Standard ML (SML) [55]. HML heavily relies on strong type polymorphism permitting to design a function

[3]This means both Verilog and SystemVerilog

operating over several different data types. HML enriches SML syntax with specific hardware description features, including hardware function declaration, signal assignments, bit-vector operations, and extensions for describing both behavioral and structural hardware. Besides, since no clock information is required in HML, there is no sensitivity list like in VHDL. On the other hand, this design choice implies that HML supports only one clock domain. Variables in HML can be declared throughout the program, overcoming a typical restriction of traditional HDLs. In particular, designers can explicitly declare variable data types and nature (e.g., input/output) or let HML infer them. Finally, HML translates the code into VHDL.

Clash: Built on top of Haskell, Clash [4] is a functional HDL with its compiler and library. Clash is a strongly typed system with a high degree of type inference. Moreover, Clash has a fully interactive read-eval-print loop (REPL) that enables an interactive design, testing, and fast simulation. Clash does not support recursion, floating, and double, while it provides fixed-point data type. Clash applies optimizations at the top-design level, and it also provides a fine-grain control on the design. For instance, with Clash, a hardware designer may define multiple clock domains and type-safe clock domain crossing, and even define reset polarity.

Chisel: Chisel [5] is an HDL embedded in Scala, which offers a more straightforward approach to HDL design compared to Verilog. For instance, the designer can define functions using Scala conventions, build and nest data structures, design components as classes, and redefine operators. Chisel specific libraries permit the designer also to employ custom data types. A key for embedding Chisel in Scala is to support highly parametrized circuits generators, a weakness of traditional HDLs. In this way, designers can declare parameterizable classes and recursively create hardware subsystems. For instance, RocketChip System-on-Chip Generator [2] is a Chisel-based framework. As another exciting feature, Chisel abstracts the memory representation. The designers can first define it and then create ports for it. Chisel offers a fast C++ simulator for RTL debugging and a Verilog translator, which permits fine changes and integration with already designed Verilog modules as black-boxes. Additionally, Chisel supports multi-clock domain designs, and it is used for a plethora of FPGA and ASIC designs [2, 99, 151], though it lacks in verification features [99].

SpinalHDL: Started in 2014 and completely open-sourced, SpinalHDL emerges as an alternative to standard HDL languages [126]. Thought by design to be interoperable with VHDL/Verilog existing tools, SpinalHDL provides a relevant abstraction layer for hardware design and verification along with out-of-the-box IP integration. Embedded in Scala, SpinalHDL exploits a functional programming paradigm. It provides no logic overhead, generates VHDL and Verilog code completely simulatable with standard EDA tools, and preserves naming and components hierarchies. SpinalHDL provides an abstraction level close to the one of VHDL/Verilog but enables a fast and easy reuse and creation of abstraction utilities. Moreover, it supports even multiple clock domains. Indeed, SpinalHDL supports customizing a clock domain from the polarity of the signals (e.g., positive edge), to the target domain frequency, from implicit clock enable feature to simulation support and timing verification. SpinalHDL provides several built-in abstractions that remarkably ease the development. For instance, the bus slave factory and the generator framework are an easy way to provide custom protocols integration, such as the AXI4 protocol and an out of order hardware elaboration for complex systems generations, such as VexRiscv[4].

DFiant: Aiming to bridge the gap between HLS and HDLs, DFiant [130] is a dataflow hardware description language. Authors argue that on the HDLs' side, there are issues in coupling degree of functionality and timing constraints and portability, whereas on the HLS's side, concurrency and fine-grain control are lost. For these reasons, DFiant is a Scala-based HDL, inheriting a polymorphic type system and object-oriented programming, with a clock-agnostic dataflow model that generates

[4]https://github.com/SpinalHDL/VexRiscv

Verilog code. A hardware designer that exploits DFiant has to care for the dataflow model to implement. Since DFiant aims at clock-less designs, the clock is an abstract concept considered only by the compiler. For this reason, multi-clock designs are not supported, nor clock-crossing domain regions. Moreover, the designer can exploit the automatic pipelining of DFiant, which takes care of retiming and pipeline balancing in an automatic way, similar to an HLS tool.

VeriScala: Primarily focused on the cooperation of hardware and software developers, VeriScala [98] is a Scala-based HDL. Indeed, the authors propose a framework to develop hardware descriptions with software-like features, such as code-reuse, high-level abstractions, and recursion. The framework also includes a novel runtime that enables a Scala-to-accelerator direct communication on top of the Xilinx PCIe subsystem. VeriScala generates synthesizable Verilog and inherits all the Scala features, such as high parametrization, polymorphic type-system, and extensibility.

2.2 Imperative-based HDL

The second category of HDLs relies on imperative paradigms, such as those offered by Java and Python. These languages' programming model is closely related to hardware description where components are reused across hierarchical and, whenever possible, decoupled designs. Although Scala-based HDLs adopt object-oriented features, they provide many functional constructs that collide with this category. This class of HDLs mainly exploit objects and classes along with polymorphism features. These features are at the basis of an easy-to-use and extend language [27, 40, 100] and tight integration with the target host machine [10, 61].

JHDL: Just another HDL (JHDL) [10] is a language designed to integrate the host and kernel development through a set of Java libraries for both circuit simulation and hardware support. JHDL leverages Java object-oriented nature to manage circuit resources as Java objects. A single class that wraps all its components and connections represents the whole circuit defined in JHDL. Java methods describe the behavior of the hardware modules, which support parametrization. The designer can easily select the target of JHDL code execution (hardware or simulation) by just changing the class description. This permits a simple host configuration as the program can be written in one place, and requires an explicit hardware/software partitioning. The simulation at the clock level happens through the JHDL simulation kernel, where the circuit class first checks all its components and connection, then starts the simulation. Conversely, if the designer aims at running the design on an FPGA, JHDL translates a set of limited Java statements into VHDL and then produces the bitstream. Finally, a relevant feature of JHDL is the support for constructors/deconstructors to reconfigure the circuits on the host side through method calls.

MyHDL: *MyHDL* [40] is a language that exploits the Python infrastructure to implement HDL specifications to open hardware development to beginners. Its HDL description is similar to Verilog, but with a more manageable approach to verification; indeed, it is possible to convert MyHDL code into Verilog/VHDL through specific built-in libraries and use constructs to verify the designs easily. MyHDL supports waveform viewing as well. MyHDL models hardware as interactive light-weight threads that communicate with each other. In particular, MyHDL description structure is based around *generators*, namely modules that wait for a specific signal to perform specific actions, that communicate through *generator functions*. Moreover, generator functions allow to keep the state of the employed functions and resume them if needed, making them usable as ultra-light threads. In this way, it is possible to pass control information to the dedicated runtime simulator. Finally, MyHDL supports co-simulation via other HDL simulators by translating MyHDL code into Verilog.

PHDL: Python HDL (PHDL) [105] is a Python framework that aims to increase the level of abstraction of hardware design, making designers more aware of what they are doing. The PHDL framework has two main components: framework classes and a component library, which contains pre-made descriptions for the low-level components. Designers build components and systems using

mainly three types of objects: connectors, components, and connections. Connectors represent actual wires and special collections of them. PHDL components may be either meta- or vanilla components: meta-components are in charge of choosing the best component to employ for a target design, vanilla ones implement the actual logic. Connection objects take care of tying connectors together. Designers implement components and wires as classes through templates provided by PHDL and connections as functions. Finally, PHDL enables the parametrization of hardware modules and integration with existing libraries through a wrapper.

MaxJ: MaxJ [61, 96] is an HDL language derivative of Java devised to describe spatial and parallel computations targeting Maxeler's Dataflow Engines (DFEs). MaxJ offers a remarkable level of abstraction for the designer, also providing easy integration between host and FPGA. The designer employs Java constructs and features (e.g., parametrization and polymorphism) to design the kernel and define components and signals, while the syntax remains close to the one of HDL languages. On the other hand, the designer may write the host code in different languages, like C/C++, MATLAB, etc. MaxCompiler translates MaxJ code into output compatible with the FPGA synthesis tools. MaxCompiler schedules the design into a pipelined dataflow architecture and connects components through FIFOs, whose size is inferred implicitly. In this way, MaxCompiler can synchronize different paths with different latencies. Finally, starting from the kernel description, MaxCompiler automatically generates the interfaces to allow the communication between host and kernel, synthesizes the kernel, and produces the bitstream.

ArchHDL: ArchHDL [148] is an HDL for RTL modeling based on C++ that focuses on intuitive module descriptions and flexible testbench description. ArchHDL's primary objective is to speed up hardware simulation and provide easier access to hardware implementation. Designers can easily compile code written using the ArchHDL library (which offers features like non-blocking assignments and all Verilog constructs) with a C++ compiler, parallelize it using OpenMP, and simulate the hardware design executing the resulting binary. These factors enable to achieve significant simulation speedup over Synopsys VCS simulator. Despite having all the benefits of C++, ArchHDL has some limitations. Since ArchHDL has to be translated into Verilog, the ArchHDL library only supports Verilog's data types. In particular, the implementation of some ArchHDL data types directly derives from C++ ones, e.g., wire and integer types come from C++ integer, and thus requires the introduction of some restrictions (e.g., arrays are at most bi-dimensional). Finally, ArchHDL supports only one clock signal and assigns variables only at the positive edge of the clock, thus limiting the possible designs.

PyMTL: PyMTL [100] is a highly productive language for Cycle-Level (CL), Functional-Level (FL), and RTL modeling. In PyMTL, the module construction follows a bottom-up approach. At first, the designer focuses on the functionality of the algorithm, thus trying multiple implementations of it. PyMTL permits rapid prototyping due to the language's nature and provides optimizations libraries for this step. The next step analyzes the operations done within a cycle. The designer can tweak this step to achieve the desired performance and enforce some optimizations provided by PyMTL libraries. At last, PyMTL implements the RTL model and generates Verilog. In terms of language features, PyMTL supports dynamic types, providing more flexibility to the possible implementations and a highly parametrizable behavioral and structural components. On the other hand, PyMTL handles the simulation using SimJIT. This custom JIT specialization engine leverages the LLVM compiler and Verilator [158] to automatically generate C++ for CL and RTL models speeding up the simulation. Besides, PyMTL permits the reuse of testbenches created for the CL and FL also for the RTL step. The current third version of PyMTL [9, 68] includes new features such as SystemVerilog generation, component parametrization, and improved simulation speed [68].

PyVerilog: PyVerilog [166] is an open-source, Python-based toolkit for analysis and code generation of RTL designs. The various tools available permit a fine grained analysis of the design, thus

providing the developer with numerous ways to optimize the code, but with additional complexity. PyVerilog offers a code parser, dataflow analyzer, control-flow analyzer, visualizer, and code generator for Verilog. The code parser analyzes PyVerilog code and generates an Abstract Syntax Tree (AST) based on the preprocessing of the code. From that, the dataflow analyzer constructs a dataflow graph that represents relationships among signals. The dataflow analyzer first passes the AST, builds a list of all the modules within it, classifies signals, and checks modules' connections. After that, the dataflow analyzer checks signal assignments and constructs a dataflow graph for each signal. The control-flow analyzer generates a graph representation of Finite State Machines (FSMs) in hardware, exploring the previously generated graph. This tool infers the values of candidate conditions from the assignment conditions and identifies the assertion conditions of the signals for state transitions. Finally, the code generator of PyVerilog generates a source code in Verilog from the intermediate representation of a PyVerilog AST.

PyRTL: Embedded in Python, PyRTL [27] is an HDL that aims to lower the barrier to digital design, promote hardware co-design, and allow complex hardware design patterns to be easily reused. PyRTL is designed for simplicity, usability, clarity, and extensibility rather than optimizations and performance. Indeed, PyRTL provides a comprehensive infrastructure in addition to the language per se. In this way, authors want to enable fast prototyping solutions in analysis and simulation other than design only. For instance, PyRTL infrastructure enables direct code instrumentation in the overall process from the intermediate representation to the code transformation applied.

2.3 SystemVerilog Extension HDL

First appeared in 2002 [144], SystemVerilog represents a significant improvement over his predecessor Verilog. Thanks to many features borrowed from the object-oriented world, SystemVerilog increases the abstractions for hardware developers (both design and verification people). The third category of HDLs based its power on extending SystemVerilog syntax for different purposes: from a new design experience to highly customizable hardware generators [152], and new design paradigms, such as the transaction-level paradigm.

BlueSpec SystemVerilog: Bluespec SystemVerilog (BSV) [16, 119] is an HDL that aims to provide a general-purpose language for hardware design using *atomic transactions*. Atomic transactions are rules that dictate the behavior of the described hardware to enable a high level of parallelism and smoothly refinable designs. The designer develops modules in BSV and implements, for each module, both methods and rules. The modules represent the outwards interfaces, while the rules update and modify the module's internal state. Both rules and methods have guards, and they can fire only if the guards are true and there are no conflicts concerning the considered rule, preserving atomicity. The code in BSV heavily relies on these transactions to deliver concurrent execution and easy reconfigurability. The designer can change the application order of the rules without modifying the rules themselves, differently from SystemVerilog. The BSV synthesis tool compiles parallel hardware for the rules, which is always logically equivalent to a serialized execution. Module interfaces are components of atomic transactions and derive from C++ and Haskell interfaces. BSV permits polymorphism to easily create complex, overloaded, and fully type-checked interfaces in a bottom-up approach by constructing templates. Designers can easily design reusable components to build a more complex architecture. Indeed, the generation mechanism of micro-architectures supports conditionals, parametrization, loops, and even recursion, making the design process more comfortable and more customizable. Moreover, BSV modules can coexist with SystemVerilog blocks, thus giving the developer the possibility to use already existing ones. Finally, BSV supports design with positive clock edge and reset asserted low by default, multiple clock domains, and DDR designs, i.e., logic active on both positive and negative clock edges. Despite this possibility, BSV does not support timing verification, which requires a standard Verilog simulator.

Table 1. Comparison table of the presented HDLs.

	HDL Language	Embedded Language	Output Language	Parametrization	Polymorphism	Simulation	Other Relevant Features
Functional	HML	SML	VHDL	Supported	Supported	Supported 3rd Party	-
	Clash†‡★	Haskell	VHDL, (System)Verilog	Supported	Supported	Supported	Multi-Clock and Reset Polarity
	Chisel†‡★	Scala	Verilog	Supported	Supported	Supported	Multi-Clock Design, BlackBox IPs, Functional Verification
	SpinalHDL†‡★	Scala	VHDL/Verilog	Supported	Supported	Supported	Multi-Clock Design, DDR Design, BlackBox IPs, Timing Verification
	DFiant†	Scala	Verilog	Supported	Supported	Supported	Dataflow, Clock-less
	VeriScala†	Scala	Verilog	Supported	Supported	Supported	Recursion, Scala-FPGA Runtime
Imperative	JHDL	Java	VHDL	Supported	Not Supported	Supported	Host Design, Partial Reconfiguration
	MyHDL†‡★	Python	VHDL/Verilog	Supported	Not Supported	Supported	Verification
	PHDL	Python	Verilog	Supported	Not Supported	Not Supported	-
	MaxJ★	Java	VHDL	Supported	Supported	Supported	DataFlow Optimizations Host Design, Multi-Clock Design
	ArchHDL	C++	Verilog	Not Supported	Not Supported	Supported	-
	PyMTL†‡★	Python	(System)Verilog	Supported	Supported	Supported	-
	PyVerilog†‡★	Python	Verilog	Not Supported	Not Supported	Not Supported	-
	PyRTL†‡★	Python	Verilog	Supported	Supported	Supported	Instrumentation
SystemVerilog Extensions	BlueSpec†‡★ SystemVerilog	Extension of SystemVerilog	SystemVerilog	Supported	Supported	Supported	Multi-Clock Design, DDR Design, Functional Verification
	Genesis2†	Extension of SystemVerilog	SystemVerilog	Supported	Not Supported	Supported	-
	TL-Verilog★	Extension of SystemVerilog	SystemVerilog	Not Supported	Supported	Supported	Timing Verification

† Open-source ‡ Last update in the last years (2020-2021) ★ Maintained, to the best of authors' knowledge

Genesis2: Genesis2 [152] provides an extension of SystemVerilog functionalities without modifying its formal syntax. Genesis2 provides hardware designers with a rich software language for writing instructions that specify how to generate modules from a set of input parameters. The behavioral description is still in SystemVerilog. Even though it requires SystemVerilog to describe the hardware modules, Genesis2 uses PERL to express module instantiation. To allow polymorphism, Genesis2 enables designers to define and give default values to parameters and then provides a simple mechanism for overriding these values from external configuration files. Genesis2 permits implementing custom types for the parameters, increasing the flexibility of the modules. Every time the Genesis2 compiler runs, it generates code and extracts the entire parametrization space hierarchically into an XML-formatted description file, thus reading out the machine configuration.

TL-Verilog: Transaction-Level Verilog, or TL-Verilog, is an extension of SystemVerilog to support a design methodology based on the concept of *transaction* [59]. A transaction represents a general entity that might be a machine instruction, a packet flow control unit, or a memory read/write, flowing through generic structures, such as pipelines, arbiters, and queues. TL-Verilog currently outputs SystemVerilog code, has a timing-abstract design methodology, and shortly will support parametrizable code. Despite the novel paradigm, it lacks characteristics that designers expect to have (such as parametrization) and presents many experimental features not yet mature enough.

2.4 Summary

Table 1 summarizes the presented HDLs, clustered by programming model (functional, imperative, SystemVerilog-based) and then ordered by date of appearance. The first two HDL clusters present many languages in contrast to the reduced number of SystemVerilog extensions. Although the semantics of these HDLs remains declarative, functional and imperative constructs and formalisms enhance their expressive power to design parallel hardware architectures. Conversely, the third cluster, which is smaller, extends an HDL such as SystemVerilog through language features that further ease the hardware development, e.g., BSV rules or TL-Verilog transactions.

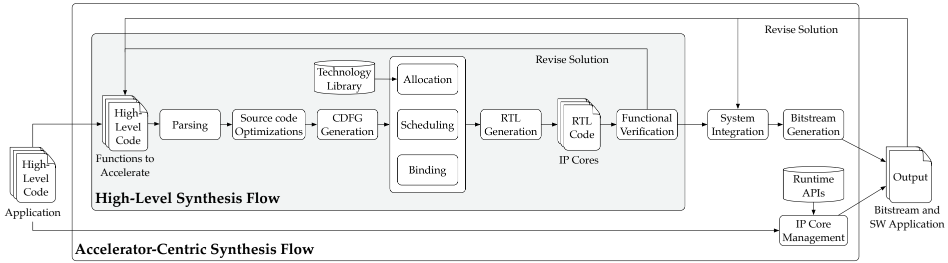

Fig. 4. High-Level Synthesis and Accelerator-Centric Synthesis flows and their integration.

All three clusters contain open-source languages, and their majority reports updates in the last two years (2020-2021). This fact introduces a community building around languages that come even before 2010 (e.g., Clash) and interests beyond the single research manuscript. Each of these languages outputs standard HDL language, either VHDL or (System)Verilog. However, the majority of the considered HDLs exploit only one of the two standards as the output language, especially Verilog. Indeed, only Clash, SpinalHDL, and MyHDL support both VHDL and Verilog outputs.

As HDL relevant characteristics, we highlight if the target language supports parametrization and polymorphism, two essential features for abstraction improvements. Moreover, functionally verifying and simulating the stimuli response of a target design give the designers hints on the process outcome; therefore, Table 1 displays the simulation support (if any) with a custom tool or if they leverage third-party tools. Moreover, we report other features we believe relevant in the last column of Table 1. Although VHDL and (System)Verilog are evolving, increasing abstractions are essential features for both software development and the hardware ecosystem. For instance, traditional hardware designers and verifiers demand support of standard HDL features, such as multiple clock domains, IP integrations, timing closures. Without some of these features, these HDLs are not considered ready for production purposes [99]. However, few HDLs offer features to design and verify architectures with multiple clock domains.

Finally, among the presented HDLs, we see a promising direction from novel languages, such as TL-Verilog. However, we believe that the most promising and mature languages are the ones that provide standard HDL features (e.g., BlackBox IP, Verification), additional ones (e.g., higher abstraction layers and constructs), a continuous development throughout the years, and a consistent ecosystem. Examples are Chisel, SpinalHDL, and BlueSpec SystemVerilog.

3 HIGH-LEVEL SYNTHESIS

After discussing relevant HDLs in literature, we now focus on HLS tools [108]. HLS [36] represents a step forward in the digital design flow, for it increases the level of abstraction of such a process. HLS aims to enable users, not necessarily expert in the hardware domain, to develop a digital custom architecture for FPGAs or ASICs starting from a high-level language. More specifically, HLS alleviates the designer's work and automates several tasks. Given an algorithmic/functional specification (untimed description) of a design (often decorated with directives/pragmas), the HLS tool translates it into an intermediate representation (usually a control and data flow graph). From this point, the HLS tool determines the types of operators and memory elements the specification needs and *allocates the resources*. Then, the next step *schedules* the operations within the specification to clock cycles. Later on, the tool *binds* each operation and variable to a specific functional unit and a memory element, respectively, and circuit interfaces (control and data signals) to peripherals (such as memory interfaces). Finally, the HLS tool *generates a fully timed RTL design*. Thanks to

this approach, HLS improves the design productivity and facilitates the exploration of the design space through source code and directives tweaking. Besides, HLS tools may reduce the verification time. Indeed, they can automatically generate testbench and functionally validate the RTL design with the very same test vectors used for the source code.

Despite the advantages, an HLS-based design flow has some relevant flaws. First of all, the user has less control over the resulting RTL design [81]. Hence, the RTL quality significantly depends on the tool internals and optimizations and may be lower than a handmade HDL design. Moreover, the majority of HLS tools offer little to no support for specific kinds of RTL designs, e.g., the ones including clock domain crossing or DDR transfers. Finally, the current generation of HLS tools mainly focuses on datapath applications [104]; thus, it may not be the best choice for control ones.

According to Martin et al. [104], we are currently in the third generation of HLS tools. While the first two generations failed for several reasons (e.g., immaturity of tools, poor quality of results, improper input languages, and wrong target users), the third generation succeeded thanks to sound design choices that fixed the past mistakes. One of these is the input language. Indeed, most HLS tool vendors require designers to use languages they are already familiar with, like C variants (C/C++/SystemC) or MATLAB, instead of specific languages or unfamiliar HDLs. Thanks to their features, modern HLS tools significantly boosted FPGA programmability and helped reduce the steepness of the FPGA learning curve, especially when the user's goal is the acceleration of a given application or part of it. Consequently, FPGAs gained increasing attention as a viable alternative to CPUs and GPUs to speed up computations while maintaining a relatively low power profile.

The efficient design of an accelerator and its RTL generation is only a part of the whole FPGA design process. Indeed, the next step, usually called *system-level design*, involves integrating the resulting IP within a more extensive system. Usually, an FPGA is not the only on-board component, but rather it is part of an ecosystem including off-chip memories, buses like PCIe, network interfaces, to name a few. Therefore, the proper connection of the produced IP with such components is paramount to deploy the accelerator, allowing the user to interact with the IP from the host processor. However, HLS tools focus on designing an IP and do not cover/automatize the system-level design step. Thus, this step involves different tools like Quartus or Vivado and is usually up to the user, requiring solid hardware knowledge to properly and effectively perform it. For this reason, in recent years, FPGA vendors started developing toolchains embedding both the HLS step and automatic system-level integration of the resulting IP oriented to the hardware accelerator design. We name such toolchains *Accelerator-Centric Synthesis (ACS)* tools: they represent a further step in the FPGA development panorama and offer a CPU/GPU-like development experience. In particular, to better resemble such an experience, these toolchains also began to support OpenCL as input language. We categorize them as the fourth and latest generation of HLS, extending the previous classification [104]. Figure 4 depicts the tight integration between HLS and ACS flows.

Throughout this Section, our analysis focuses on the most relevant HLS tools available in the literature, whether they just perform the RTL synthesis step (Section 3.1) or the entire flow towards the bitstream generation (Section 3.2). In particular, our study only examines tools that directly implement the HLS/ACS flow. Consequently, we exclude solutions implementing similar high-level and automated toolchains as they build upon the considered tools [30, 137, 156, 179].

3.1 High-Level Synthesis Tools

This Section describes the current status of the most relevant HLS tools belonging to the third generation, reported in chronological order as in Figure 5. We only consider *"pure"* HLS tools, i.e., tools that do perform the HLS process without delegating it to third-party software [30, 137, 179]. For each HLS tool, we provide an overview of their features, supported optimizations, and peculiarities.

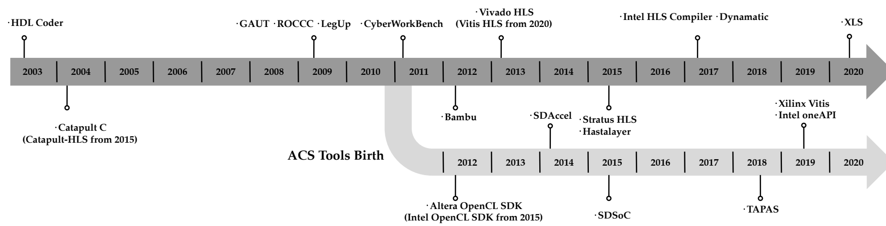

Fig. 5. Timeline of High-Level and Accelerator-Centric Synthesis Tools (first release/published paper).

HDL Coder: HDL Coder [106] is a tool available within the MATLAB suite by Mathworks that generates synthesizable Verilog/VHDL from MATLAB functions, Simulink models, and Stateflow diagrams. The designer can employ the resulting RTL to target both FPGAs (Xilinx, Intel) and ASICs (Microsemi SoC). Given a MATLAB function, HDL Coder first automatically converts floating-point data types to fixed-point, then generates the RTL. The designer can tune the design via optimization features, including loop optimizations, array mapping to on-chip memory, and area optimizations. Similarly, given a Simulink model, HDL Coder generates RTL from the employed library blocks. HDL Coder provides a workflow advisor integrated with both Xilinx and Intel synthesis tools. The workflow advisor supports the designer during various phases, from synthesis to FPGA programming. For instance, the designer can exploit the advisor to analyze the resource usage or highlight the critical design tasks and annotate the Simulink model with such information to identify the main bottlenecks. In terms of verification, the HDL Verifier tool allows the designer to verify the generated RTL behavior functionally via either a MATLAB or Simulink testbench. Likewise, the designer can test and debug FPGA implementations on Xilinx and Intel boards.

Catapult-HLS: Catapult-HLS (previously Catapult-C) [14, 112] is a commercial platform by Mentor Graphics for both FPGAs and ASICs. During the design process, the user can target one or more technology libraries and generate hardware designs for multiple FPGAs and ASICs starting from the very same algorithmic description. In terms of input/output code, Catapult-HLS takes a C++ or SystemC input and outputs RTL design in VHDL/Verilog. In particular, Catapult-HLS completely supports both syntax and semantics of C++, as well as all basic statements, functions, pointers, and templates. Only two restrictions apply and involve dynamic memory management and code properties (e.g., array dimensions), as the input code must be statically determinable. In addition to standard libraries, Catapult-HLS supports arbitrary precision for both integer and fixed-point data types. Moreover, it enables the optimization of the resulting design at various levels via directives/pragmas. For instance, the user may apply basic loop optimizations, *hierarchical synthesis*, specific resource allocation, and scheduling optimizations. Finally, Catapult-HLS GUI offers a set of built-in graphical analysis tools, such as the *Gantt Chart Viewer* for code profiling, *Resource Viewer* for HLS results visualization, and a tool for quality coverage metrics.

GAUT: GAUT is an academic and open-source HLS tool dedicated to digital signal processing (DSP) applications [35, 173]. GAUT GCC-based compiler accepts in input C/C++ code and generates VHDL synthesizable by Xilinx, Intel, and Synopsys tools. It can also produce SystemC for simulation, visual prototyping, or Design Space Exploration (DSE) purposes. In particular, the generated SystemC models are cycle- and bit-accurate and employable in external platforms, like SocLib [35]. Thanks to the support for the *Algorithmic CTM* library from Mentor, GAUT supports bit-accurate integer and fixed-point variables. Besides, the designer can specify constraints on the throughput and the clock period, while the memory mapping and the I/O timing diagram are optional. The

compiler lowers the input code to an annotated dataflow graph and then to the GIMPLE-IR, where most of the analyses and optimizations happen. Like most of the HLS tools, GAUT supports loops, memory, and scheduling optimizations. After performing the standard HLS steps, GAUT generates an RTL architecture that comprises: a processing unit, a memory unity, and a communication and interface unit. Finally, GAUT automatically generates a test bench for architecture validation.

ROCCC: Riverside Optimizing Compiler for Configurable Computing (ROCCC) [174, 176] is a C-to-VHDL HLS compilation toolchain initially developed by the University of California Riverside and then by Jacquard Computing Inc., the current maintainer. ROCCC is particularly suitable for streaming applications and applies several restrictions on the input C code. For instance, since loops are the main target of ROCCC optimizations, it requires perfectly nested loops with a fixed stride and forbids the while loop construct. ROCCC provides an integrated control on code transformations and optimizations (to improve throughput, memory accesses, and resources), manages the modules' and cores' instantiation into C code, defines the platforms' interfaces, and generates the verification test benches. Besides, the designer can manually tune the same code for different FPGAs by varying the compiler optimizations/parameters, e.g., the channels' number and bitwidth, without altering the original code. ROCCC offers support for floating-point and integer operations, while it precludes resource sharing in favor of better performance. Finally, ROCCC provides peculiar features like native support for triple modular redundancy and *smart buffers*, i.e., on-chip buffers enabling data reuse through loop iterations according to the data access pattern.

LegUp: LegUp is an HLS tool developed and released by LegUp Computing [90], which Microchip Technology Inc. acquired in 2020. Up to version 4.0, LegUp was open-source, developed by the University of Toronto [20], and available for non-commercial use [46]. LegUp can synthesize the same design for Xilinx, Intel, Lattice, Microsemi, PolarFire, and Achronix FPGAs. It offers an Eclipse-based IDE and accepts C/C++ as input languages with restrictions on dynamic memory allocation and recursion. The tool exposes two design flows to the designer: *only-hardware* and *software-hardware*. In the former, LegUp synthesizes the entire input C/C++ code to Verilog. In the latter, LegUp profiles the software code and identifies the best candidates for hardware acceleration. After the designer's manual partitioning, LegUp generates a hybrid system comprising a processor (either a MIPS softcore or a hard ARM processor) and one or more accelerators, which communicate via a memory-mapped interface. LegUp supports standard HLS optimizations like loop transformations, pthreads, OpenMP, and synthesizes the multi-threaded software into parallel hardware blocks. Besides, LegUp can automatically apply bitwidth reduction via a static variable range and bitmask analysis, multi-cycle path analysis, and register removal. Finally, the designer can simulate the resulting RTL with ModelSim by Mentor Graphics.

CyberWorkBench: CyberWorkBench (CWB) [117] by NEC integrates an HLS design process, comprising synthesis, simulation, and verification for both FPGAs (Xilinx and Intel) and ASICs. The core idea of CWB's suite is the "All-in-C" approach, which comprises *All-in-C Synthesis* (i.e., all modules described in C), and *All-in-C Verification* (i.e., all the verification tasks are at the C level). CWB generates Verilog/VHDL from an extended ANSI-C, called *BDL* or *Cyber-C*, or SystemC along with a set of design constraints, which may refer to clock frequency, resource kind, and number. Moreover, CWB accepts RTL and netlists as black boxes, insertions of assertions/properties, directives, and clock boundary areas, which help describe complex timing behavior concisely and fix their scheduling. CWB is one of the few HLS tools able to handle clock domain crossing and clock gating. Then, CWB provides a configurable and extensible base processor fully described in BDL. Finally, CWB supplies a DSE tool, that given a set of constraints (e.g., area and latency), produces multiple architectures with multiple trade-off charts. Through CWB, it is possible to verify the hardware at behavioral and cycle-accurate levels, also thanks to the automatic hardware testbench generation and the interaction with third-party tools.

Bambu: Bambu [129] is an HLS tool developed at the Politecnico di Milano as part of the Panda framework [125]. Bambu takes as input a behavioral description written in C and outputs a synthesizable RTL implementation in VHDL/Verilog and a testbench for simulation and functional verification. Bambu targets multiple FPGA vendors, namely Xilinx, Altera/Intel, and Lattice, and ASICs. Similarly, various simulators are integrable into Bambu, such as Mentor Modelsim, Xilinx ISIM, and Verilator. Bambu supports most C constructs (e.g., function calls, multidimensional arrays) and benefits from all the target-independent GCC-based optimizations. Currently, Bambu does not support recursion, but, if necessary, GCC can automatically convert recursive forms into non-recursive ones. Moreover, Bambu provides an additional speculative scheduling algorithm and allows the possibility to specify fixed scheduling (as an XML file) as input. Furthermore, the designer can activate the post-rescheduling option to distribute resources better. In terms of HLS optimizations, Bambu supports operation chaining, resource sharing, and pipelining. It also offers HW/SW partitioning features and provides various Pareto-optimal implementations *(trading off latency and resources)*. Finally, Bambu modular organization makes it is easily extensible.

Vivado/Vitis HLS: Vivado HLS [183], formerly AutoPilot by AutoESL and then acquired by Xilinx in 2011, is a design suite for HLS that allows converting high-level languages in HDL. Vivado HLS accepts C, C++, or SystemC as input specification languages and can generate Verilog or VHDL hardware descriptions. The designer can specify the target FPGA and provide constraints on the clock period, clock uncertainty and optimization directives to better control the HLS process. Vivado HLS accepts most of the constructs of C/C++ while applying the usual restrictions, such as recursion and dynamic memory allocation. Besides, Vivado HLS provides various built-in libraries that range from mathematical operations to arbitrary precision for integer and fixed-point data types. The designer can leverage multiple directives to improve the final design, such as loop transformations, binding to specific resources, hardware interfaces definition, and dataflow execution model. Vivado HLS supplies tools for functional verification of the resulting design at both software and hardware level with the same software testbench. Moreover, Vivado HLS produces various reports about circuit timing, resource usage, and scheduling. Finally, thanks to the integration with the other Xilinx tools, the designer can invoke synthesis and place & route steps within Vivado HLS to assess the design quality. From the 2020 release of its developer tools, Xilinx substituted Vivado HLS with Vitis HLS [187], which automatically applies more optimizations and relies on AMBA AXI4 interface protocol to communicate with the off-chip memory by default.

Stratus HLS: Released by Cadence, Stratus HLS [19, 134] is a commercial HLS platform that targets ASICs, SoCs, and FPGAs. After acquiring Forte Design Systems in 2014, Cadence created Stratus HLS out of Forte's Cynthesizer and its own C-to-Silicon Compiler. This HLS tool takes C, C++, or SystemC descriptions and creates RTL (Verilog) implementations. In addition to the input code, the designer can provide high-level implementation constraints, which are beneficial to tailor the design to various target architectures without changing the input specification. Stratus HLS provides an IDE that allows the designer to to actively trade off power, area, and performance. This tool accurately identifies hotspots in the RTL, both in time and space, supports power-aware scheduling, and offers many other low-power optimizations. After applying various optimizations, the designer can exploit control and dataflow graph schematic viewer and pipeline analysis and visualization tools to evaluate the impact of such optimizations. Finally, Stratus HLS helps automate the design and verification flow of hundreds of blocks from transaction-level modeling (TLM) to gates, providing tools for RTL verification, debugging, power analysis, and design exploration.

Hastlayer: Hastlayer is a free open-source HLS tool developed by Lombiq Technologies [101]. The goal of Hastlayer to enable software developers of the .NET platform to target FPGAs easily and accelerate their applications. Since the .NET platform supports multiple popular programming languages, Hastlayer supports languages such as Python, PHP, C++, JavaScript, to name a few.

However, it applies restrictions on the synthesizable code. For instance, Hastlayer does not support floating-point data types (`float` and `double`) or types wider than 64 bits. After the HLS process, the tool integrates the resulting VHDL within the Hastlayer hardware framework and invokes existing vendor toolchains to produce the bitstream file. Currently, Hastlayer offers two flows: one for Xilinx FPGAs and one for Intel FPGAs available on the Microsoft Catapult cloud [135]. Hastlayer targets both FPGA-experts and developers with no hardware design knowledge. On the one hand, with Hastlayer, a .NET software developer can select a compute-bound part of their application. Then, the tool automatically swaps out the software implementation with the FPGA one, hiding and abstracting the interaction with the hardware. On the other hand, Hastlayer assists FPGA-experts by offering options to fine-tune their designs. Besides, the designers can leverage both task-level and operation-level parallelism through standard .NET constructs.

Intel HLS Compiler: Intel HLS Compiler [64] takes untimed ANSI C/C++ as input and generates RTL optimized for Intel FPGAs. It is part of Intel Quartus Prime Design Software for FPGA design. Intel HLS Compiler has native support for fixed-point and floating-point data types and supports arbitrary width integers. Like other HLS tools, the compiler has some limitations regarding the supported subset of C99 and C++. For instance, the Intel HLS compiler does not support dynamic memory allocation, virtual functions, function pointers, and C/C++ library functions except for a few math functions. On the other hand, the Intel HLS compiler provides the designer with a design exploration tool and high-level constraints and directives. In this way, the designer can annotate the code and specify the optimizations the compiler has to apply (e.g., loop pipelining and unrolling). Similarly, the Intel HLS Compiler can also perform device-specific optimization and technology mapping for Intel FPGAs. Finally, the tool offers software testbench verification features and generates interactive analysis reports with cross-probing support.

Dynamatic: Dynamatic [70] is an academic open-source HLS framework [69] developed at EPFL. This tool converts C/C++ code into synchronous dataflow circuits. The main feature of Dynamatic is its ability to schedule the resulting circuit dynamically. Usually, most HLS tools tend to schedule the output circuit statically, forcing worst-case assumptions and, consequently, reaching suboptimal results. This scenario particularly applies when the target application contains memory/control dependencies that the HLS tool cannot identify at compile time. Conversely, Dynamatic's approach supports the design of dataflow circuits able to adjust the schedule at runtime. Besides, Dynamatic leverages performance modeling to optimize the throughput through optimal buffers placement and sizing. Finally, thanks to its open-source nature, developers can extend Dynamatic by adding custom features, pragmas, and optimization passes.

XLS: XLS [53], or Accelerated HW Synthesis, is a project by Google for rapid hardware IP development through HLS. Although not officially supported by Google, and still experimental, and in rapid growth, XLS was born to make DSLX generate SystemVerilog synthesizable code and target both ASICs and FPGAs. DSLX is a functional language that mimics Rust, able to express dataflow computations. A developer describes through DSLX both hardware and host code. Currently, the support for C++ code is under development. A peculiar feature of XLS is a fuzzer component for random program generation. Indeed, fuzzing is a valuable tool for bug discovery in compilers, though a successful fuzzer is highly costly due to the language support required [37]. Another feature of XLS is its pure dataflow IR named XLS IR. This IR is specialized for generating circuitry and embeds high-level parallel patterns to improve the final design effectiveness.

3.2 Accelerator-Centric Synthesis Tools

This Section analyzes ACS tools in chronological order, as depicted in Figure 5. These tools provide a unified environment where the user can design both the host and accelerator code and integrate them via high-level APIs. The ACS tool takes care of both the HLS process and the system-level

integration according to the target scenario, i.e., host-to-IP communication via shared memory (embedded) or PCIe (high-end/cloud). Once the code is ready, the user can "compile" it similarly to a software flow and then deploy the output files on the target platform.

Intel FPGA SDK for OpenCL: Intel FPGA SDK for OpenCL [63], previously known as *Altera SDK for OpenCL* (AOCL) [38], is a development environment that enables software developers to accelerate their applications targeting heterogeneous platforms with Intel CPUs and Intel FPGAs. The designer develops both the kernel and host code within the same environment. In particular, the designer relies on the OpenCL computational paradigm to implement the accelerator and its APIs to manage the interaction between host and kernel. Intel FPGA SDK for OpenCL offers multiple tools to evaluate the quality of the current implementation. The tool inserts performance counters in the FPGA design, and the result obtained can then be reviewed by the designer using the Dynamic Profiler tool. Moreover, it provides analysis on the resources and performance, a fast FPGA-based emulation, *what-if kernel* performance analysis, and support for symbolic debugging. Once the host application and the kernel match the expected performance, Intel FPGA SDK for OpenCL performs a complete compilation towards the bitstream.

SDAccel: SDAccel [185] is a Development Environment by Xilinx aimed at accelerating computational kernels in data centers (such as encryption, search, speech recognition, image recognition) through FPGA resources. The SDAccel IDE provides functionalities for code development, debugging, area reports, and profiling of kernel performance and memory transfers. It also offers coding templates and software/hardware emulation to verify the kernel functional correctness. The SDAccel compiler enables using any combination of OpenCL, C, and C++ to develop the hardware kernel. Moreover, SDAccel automatically manages the FPGA runtime. In this way, applications can have multiple kernels swapped in and out of the FPGA during runtime (via partial reconfiguration) without disrupting the interface between the server CPU and the FPGA, i.e., the SDAccel shell. Finally, SDAccel wraps Vivado HLS and Vivado Design Suite toolchains, abstracting and automating all the steps towards the bitstream generation.

SDSoC: The SDSoC development environment [184] is a tool by Xilinx and has similarities with SDAccel functionalities, even though they target different scenarios. Indeed, SDSoC provides a framework for developing hardware-accelerated applications for embedded systems using C/C++/OpenCL languages. Specifically, it targets Zynq SoC and MPSoC platforms, i.e., systems comprising both a hard ARM processor and an FPGA along with units for real-time and multimedia processing (on Zynq MPSoCs) on the same chip. The compiler analyzes the input program to determine the dataflow between software and hardware functions. On the one hand, it automatically implements the hardware functions on FPGA, and, on the other hand, it generates the host code for bare metal, Linux, or FreeRTOS. It also allows design exploration and system-level profiling without requiring prior knowledge of FPGAs. Indeed, SDSoC provides an automated flow for software acceleration on FPGA and system connectivity generation, management of data transfer, synchronization of hardware accelerators, and automatic hardware-software partitioning.

TAPAS: Built on top of a parallel IR called Tapir [149], TAPAS is an open-source toolchain from Simon Fraser University, released in 2018 [103]. TAPAS aims to provide HLS support for dynamic task parallelism and parallel programming in general. Among the supported patterns, TAPAS handles arbitrarily nested parallelism, irregular task parallelism, and dynamic scheduling. The tool is language-agnostic, but it heavily depends on Tapir's supported languages/frameworks, i.e., Cilk, Cilk-P, and OpenMP [103]. TAPAS operates in three stages. The first one analyzes the IR and extracts task dependencies and the top-level Chisel (Section 2.1) module that instantiates the DRAM interface, a shared L1 cache coherent with the L3 cache of the core processor, and the task units. Then, stage 2 analyzes the program graph for each task and generates each unit's RTL dataflow. Finally, stage 3 configures a set of parametric components and generates the bitstream.

Vitis Unified Software: Vitis [186] is a unified software released by Xilinx in 2019, which incorporates the functionalities of previous tools like SDAccel and SDSoC, adding new ones. In particular, Vitis supports the development of accelerated applications and embedded software, targeting both high-end and embedded FPGA-based platforms, as well as Xilinx's ACAP platform [50]. Also, Vitis relies on Vitis HLS [187] to synthesize C/C++ and OpenCL code into RTL.

One of the key components available within Vitis is Vitis AI. This development environment permits users to accelerate the inference of Artificial Intelligence models on FPGA. Starting from a high-level description in TensorFlow, PyTorch, or Caffe, Vitis AI optimizes and compiles the model into a binary for Xilinx's Deep Learning Processing Units (DPUs). Another relevant feature of Vitis is a better integration with the Xilinx Runtime library (XRT) [189]. This library aims at easing the communication between the host and the accelerators, independently from their physical connection or target platform. Finally, Vitis offers a set of open-source out-of-the-box accelerated libraries [188] covering various fields ranging from computer vision to linear algebra.

Intel oneAPI Toolkits: Intel oneAPI Toolkits [65] is a set of toolkits by Intel to deploy accelerated applications on systems comprising CPUs, GPUs, and FPGAs with a unified programming model. It provides a common developer experience across different accelerator platforms to boost performance and productivity. The core of oneAPI specification is *Data Parallel C++* (DPC++), a cross-architecture programming language based on the SYCL standard. DPC++ supplies explicit parallel constructs and offload interfaces to target a broad range of heterogeneous computing environments, including CPUs and FPGAs. Moreover, Intel oneAPI accepts partially other C-based languages. Indeed, it can automatically migrate 80-90% of C/C++ and Compute Unified Device Architecture (CUDA) code to DPC++, while the designer has to manually migrate the remaining part and integrate it within the oneAPI/DPC++ flow. After implementing a kernel, the developer can perform a software emulation on the CPU to verify functional correctness. Likewise, oneAPI can generate various reports to identify the design bottleneck, view the schedule, and provide area and timing estimates. In this way, the designer can fine-tune the kernel via directives. Once satisfied with the performance design, oneAPI takes care of the FPGA bitstream compilation.

3.3 Summary

Table 2 reports the key characteristics of the analyzed HLS/ACS tools. Both companies and academia contributed to the growth and success of the third generation of HLS. In particular, a significant initial effort came from academia (Figure 5), whose research products, like LegUp, AutoPilot (now Vivado HLS), and ROCCC, turned into commercial tools. Another pivotal factor that made this generation successful is the adoption of C-based languages. Indeed, most tools rely on C-based languages with specific restrictions, e.g., recursion, dynamic memory allocation. Only HDL Coder and Hastlayer do not follow such a trend. Among the supported languages, OpenCL is getting more attention, particularly for ACS, providing a unique language to target multiple devices.

Moving to the output of the HLS process, multiple tools generate RTL code suitable for more than one FPGA vendor, as well as ASICs. This feature makes such tools particularly flexible, especially when FPGAs are just an intermediate prototyping step towards ASIC deployment. On the other hand, most ACS tools target just one vendor, which is also their developer. Indeed, since ACS tools perform both HLS and system-level integration steps, it is easier for an FPGA vendor to integrate different products within a unified environment and provide designers with a complete toolchain. In this way, thanks to the predefined flow, the vendor can introduce additional optimizations to improve the hardware design for the target FPGA.

Considering other relevant aspects, we believe that functional verification of a hardware design is one of the essential features HLS tools should implement since it enables checking the correctness of the produced RTL code. However, various tools, especially "pure" HLS ones, offer partial

Table 2. Comparison table of the presented HLS/ACS tools.

	Tool	License	Owner	Input Language	RTL Language	Target Vendor	TestBench Generation	Simulation	Domain	Precision FP	Precision FixP	Other Relevant Features
HLS	HDL Coder★	Commercial	Mathworks	MATLAB	VHDL, Verilog	Intel, Xilinx	Yes	SW, HW	DSP, Image Proc.	No	Yes	ASIC support, graphical design
HLS	Catapult-HLS★	Commercial	Mentor Graphics (now Siemens)	C++, SystemC	VHDL, Verilog	Intel, Xilinx	Yes	SW, HW	All	Yes	Yes	ASIC support
HLS	GAUT†	Academic	U. Bretagne Sud	C, C++	VHDL, SystemC	Intel, Xilinx	Yes	HW	DSP	No	Yes	ASIC support
HLS	ROCCC†	Commercial	Jacquard Computing	C subset	VHDL	Xilinx	Yes	HW	Streaming	Yes	No	Smart buffers
HLS	LegUp★	Commercial	LegUp Computing (now Microchip)	C, C++	Verilog	Microchip	Yes	SW, HW	All	Yes	Yes	Pthread and OpenMP support
HLS	CyberWorkBench	Commercial	NEC	C, C++, SystemC	VHDL, Verilog	Intel, Xilinx	Yes	SW, HW	All	Yes	Yes	Multi-clock design, clock-gating, ASIC support
HLS	Bambu†‡★	Academic	PoliMI	C	VHDL, Verilog	Intel, Xilinx, Lattice	Yes	SW, HW	All	Yes	No	Modular and extensible, ASIC support
HLS	Vivado/Vitis HLS★	Commercial	Xilinx	C, C++, OpenCL SystemC	VHDL, Verilog, SystemC	Xilinx	Yes	SW, HW	All	Yes	Yes	-
HLS	Stratus HLS★	Commercial	Cadence	C, C++, SystemC	VHDL, Verilog, SystemC	Agnostic	Yes	SW, HW	All	Yes	Yes	ASIC and SoC support
HLS	Hastlayer†‡★	Commercial	Lombiq Technologies	.NET	VHDL	Intel, Xilinx	No	SW	All	No	Yes	Support for .NET framework languages
HLS	Intel HLS Compiler★	Commercial	Intel	C, C++	Verilog	Intel	Yes	SW, HW	All	Yes	Yes	-
HLS	Dynamatic†‡★	Academic	EPFL	C/C++	VHDL	Xilinx	Yes	HW	All	Yes	N/A	Dynamically scheduled circuits
HLS	XLS†‡★	Commercial	Google	DSLX, C++	SystemVerilog	Agnostic	No	SW, HW	All	Yes	Yes	ASIC support, fuzzer, HW-oriented IR
ACS	Intel FPGA SDK★ for OpenCL	Commercial	Intel	OpenCL	VHDL, (System)Verilog	Intel	Yes	SW, HW	All	Yes	Yes	System Integration Bitstream Generation
ACS	SDAccel	Commercial	Xilinx	C, C++, OpenCL	VHDL, (System)Verilog	Xilinx	Yes	SW, HW	All	Yes	Yes	System Integration Bitstream Generation
ACS	SDSoC	Commercial	Xilinx	C, C++, OpenCL	VHDL, Verilog	Xilinx	Yes	SW, HW	All	Yes	Yes	System Integration Bitstream Generation
ACS	TAPAS†	Academic	Simon Fraser	Cilk(-P), OpenMP	Chisel	Intel SoC	No	HW	All	Yes	No	System Integration Bitstream Generation
ACS	Vitis Unified Software★	Commercial	Xilinx	C, C++, OpenCL	VHDL, Verilog	Xilinx	Yes	SW, HW	All	Yes	Yes	System Integration Bitstream Generation
ACS	Intel oneAPI Toolkits★	Commercial	Intel	C, C++, CUDA, OpenCL	VHDL, Verilog	Intel	No	SW, HW	All	Yes	Yes	System Integration Bitstream Generation

† Open-source ‡ Last update in the last years (2020-2021) ★ Maintained, to the best of authors' knowledge

functionalities for design verification, as they just supply one type of simulation (SW or HW). In contrast, most ACS tools provide a more comprehensive experience, implementing SW and HW simulation and even performance analysis via profiling tools. Similarly, modern ACS/HLS tools are now general and mature enough to implement features like floating and fixed-point data types and enable the hardware design of applications belonging to different domains. This last aspect shows a firm evolution compared to the first HLS tools of the third generation, which focused on a specific computational field (e.g., HDL Coder and GAUT). Nonetheless, domain specialization has its advantages, and Section 4 analyzes research efforts in that direction.

In the future, we foresee that vendors will keep furthering their toolchains, especially ACS ones, to make FPGAs more appealing to software developers. Indeed, these tools, which we consider the fourth generation of HLS [104], resemble the software development experience more than the hardware one, hiding and automating low-level technicalities like system-level integration. Besides, they represent the ideal entry point to both accelerating and deploying compute-intensive workloads vendors' accelerator cards. However, if, on the one hand, this approach helps the designer as it offers automated system-level integration and runtime APIs to manage the interaction with the FPGA, on the other, it constraints the development to a specific flow. For this reason, we believe that "pure" HLS tools, i.e., the third generation, will continue to exist as they enable fast software-like development of IPs for custom systems out of the scope of ACS tools.

4 DOMAIN-SPECIFIC LANGUAGES FOR FPGA DESIGN

So far, we have described the most relevant HDLs and HLS tools in the literature. Our analysis highlights how such approaches provide a higher abstraction level than (System)Verilog and VHDL when designing FPGA-based systems. In this context, it is worth noting that, as the abstraction level increases, the required hardware expertise decreases, thus reducing the steepness of the FPGA-design learning curve. For instance, ACS tools hide the system-level design phase, dramatically

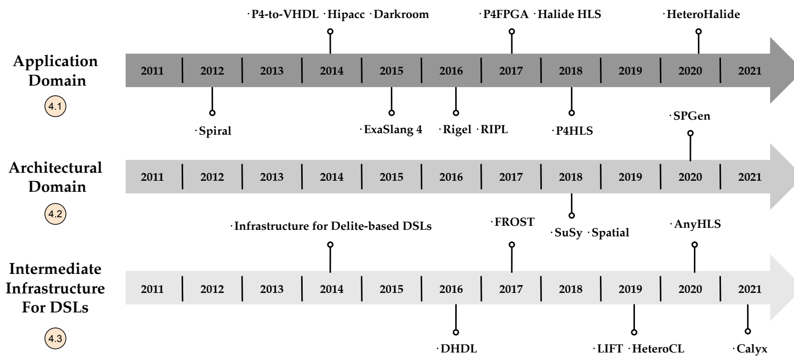

Fig. 6. Timeline of the reported DSLs and Intermediate Infrastructures from when FPGA support started.

simplifying the designer's work. Nonetheless, some hardware knowledge is still necessary to develop effective designs. For instance, given a target scenario, a designer may choose among various architectural solutions (e.g., streaming, Systolic Array) and consider different optimizations to improve the final circuit quality. To this end, automating the optimization of a hardware design would further raise the abstraction level; however, it is no trivial task due to the vast design space to explore, especially for tools covering different domains. While the literature contains several approaches to optimize FPGA designs (semi)automatically under certain constraints [22, 137, 157, 172, 175, 194], this Section focuses on a precise solution, namely Domain-Specific Languages (DSLs).

Although they have been around for several decades [47], DSLs, and domain specialization in general, gained a lot of popularity in recent years [58, 71, 97, 120, 139, 171] for many reasons. First of all, modern DSLs enable developers to quickly and easily develop portable code for multiple architectures, especially CPUs and GPUs, increasing productivity. Then, the domain restriction allows DSL compilers to explore the design space quickly, identify the typical computational patterns of the target domain, and produce highly optimized implementations. Similarly, the language restrictions permit a better static code analysis, removing unnecessary constructs which may impact its effectiveness, like pointer management [84]. Consequently, DSL applications may reach remarkable performance with a relatively minor design effort than other more general languages and frequently outtake hand-tuned libraries [7, 138].

Similar to CPUs and GPUs, DSLs are particularly convenient also for FPGAs. Thanks to domain specialization, the compiler can quickly explore the design space and leverage the FPGA features. In particular, such an exploration involves both evaluating many and different optimizations and selecting the most proper base architecture. In this way, the compiler relieves the burden of manually exploring various solutions, permitting designers to just focus on the functional description of the target algorithm and further reducing the learning curve steepness. This aspect is highly beneficial in the FPGA scenario, where the development of new solutions is highly time-consuming.

In this Section, we describe the most relevant DSLs targeting FPGAs available in the literature. In particular, our analysis only examines languages of limited expressiveness developed for a specific domain or intermediate frameworks upon which such languages can build to target FPGAs. Therefore, we exclude other domain-specific tools that do not directly involve the usage of a DSL. For instance, although they perform a similar task, we exclude the various machine learning frameworks available in literature as they require as input a high-level description in a given format (e.g., JSON, Protocol Buffers) that does not fall under the DSL definition [47]. However, an interested reader may look at other specific literature surveys for more details about such tools [175].

We group the DSLs in three main clusters according to a taxonomy based on the DSL features and purpose. The first cluster considers DSLs that focus on a particular application domain (Section 4.1), while the second contains the ones tailored to a specific architectural model (Section 4.2). Finally, the third cluster comprises intermediate languages and infrastructures for DSLs (Section 4.3).

4.1 Application Domain

This Section examines the group of DSLs that concentrate on a given application domain. Such languages implement specific constructs and abstractions that ease the development of efficient code for image processing, packet processing, numerical solvers, and so on. From the code, the DSL compiler generates the FPGA design enforcing effective optimizations for the target domain.

Spiral: Based on the Signal Processing Language (SPL) [190], Spiral is a hardware generation framework that employs a high-level mathematical formalism to generate custom hardware implementations of linear signal transforms [114], e.g., discrete Fourier transform (DFT) and fast Fourier transforms (FFTs). Spiral developers extend SPL-based formulae to introduce on top of SPL new datapath concepts for the sequential reuse of hardware structures called streaming and iterative reuse. This extension of SPL is called hardware SPL (HSPL), being more friendly to hardware generation. The overall compilation framework starts by inputting a signal transform and performing algorithmic manipulation to produce an SPL-friendly formula. A step called *formula rewriting* transforms the SPL-based formulae into HSPL-based ones, making explicit the sequential reuse. Finally, the framework translates HSPL formulae into synthesizable Verilog code that can target both FPGAs and ASICs. In 2018, the developers extended Spiral with a new formalism called Operator Language (OL) and extended the support to both CPUs and GPUs [48].

P4 Frameworks: P4 [15] is a high-level language for programming packet processors born in 2014 to answer the increasing demand for adaptable switches. In particular, P4 is a protocol-independent, platform-agnostic, and field reconfigurable language. Indeed, a developer can compile P4 code for both ASICs or software switches and change the packet processing after deployment. Thanks to these features and the wide employment of FPGAs in the networking field, many P4-to-FPGA frameworks were born after its release. Here we review the most relevant ones.

P4-to-VHDL is an experimental framework developed in 2016 that, starting from a P4 program, produces a VHDL-based architecture for packet parser at 100Gbps [11]. The general architecture P4-To-VHDL employs for the final VHDL translation is based on the so-called HFE M2 architecture [11], which presents a structure based on two components: a protocol analyzer and a pipeline of processing modules targeting Xilinx Virtex 7 FPGA. Based on this work, Cabal et al. propose a new version of the target packet parser architecture that achieved even better throughput [18].

P4FPGA is a framework that provides a P4-to-BSV (Section 2.3) translation [178]. P4FPGA starts by taking standard P4 IR [15], performs an IR-to-IR transformation, then composes the basic blocks of the programmable pipeline, and finally emits the pipeline as BSV code for standard FPGA flow. Besides, P4FPGA produces a runtime system that provides hardware-independent abstractions for functionalities such as transceiver management, and host/control plane communication.

P4HLS is an open-source framework for the generation of programmable packet parsers [146, 147]. This framework outputs highly templated C++ classes that can then be synthesized through traditional HLS toolchains. P4HLS obtains better trade-offs on latency and resource usage thanks to its architectural improvements and its graph reduction algorithm for pipeline simplification.

Hipacc: Born in 2012 for GPUs mainly [109], Hipacc is an open-source framework [111] from a joint effort of Nürnberg and Saarland Universities that enables the design of image processing kernels in a domain-specific language [110]. The developers extend their framework to support the generation of C- and OpenCL-based kernels for HLS toolchains of Xilinx [143] and Intel [142],

respectively. The Hipacc DSL is embedded in C++, and the framework exploits LLVM infrastructure [85] for a source-to-source compilation. Moreover, Hipacc generates optimized code for a target architecture leveraging vendor-specific optimizations. For instance, the first FPGA-capable version [143] employs a float-to-integer conversion of convolution coefficient to exploit the FPGA resources better. Finally, Hipacc produces a testbench for the verification step on HLS toolchains.

Darkroom: Darkroom [56] is an image processing DSL and a compiler embedded in the Terra language [45]. Designers can exploit Darkroom to realize image processing pipelines for FPGAs, ASICs, and CPUs. Darkroom expresses image processing algorithms as direct acyclic graphs of image operations, restricting them to fixed-size stencils. In particular, Darkroom implements such pipelines based on the *line-buffering* architectural pattern, which consists of storing intermediate data between the pipeline stages. This pattern permits minimizing the memory bandwidth and improving performance and power efficiency. Given an input pipeline, Darkroom finds the minimal buffer size via an integer linear programming formulation and automatically schedules the computation.

ExaSlang 4: ExaStencils language (ExaSlang) [150] is a multi-layer DSL designed to accelerate multigrid-based numerical solvers on FPGA. ExaSlang consists of four abstraction layers tailored to different classes of users. ExaSlang 4 represents the most concrete layer where designers can develop procedural programs exploiting domain-specific elements (e.g., stencils and fields) and communication statements to manage the layer parallelism. A peculiar feature of ExaSlang 4 is the Level Specification, which enables designers to define objects, like functions and stencils, depending on the multigrid level, thus overriding default ones. Finally, ExaSlang 4 implements the multigrid algorithm as a sequence of kernels connected by FIFOs and translates it into code for Vivado HLS.

Rigel: Based on the Darkroom framework, Hegarty et al. presented a new DSL framework for image processing called Rigel [57], whose paradigm focuses on productivity rather than supplying automatic scheduling. In contrast with Darkroom, Rigel can also support pyramids image processing and sparse computations thanks to a new multi-rate architecture. The Rigel DSL is embedded in Lua and enables two different flows. The first flow translates Rigel into Terra modules to allow a fast cycle-accurate simulation environment. The second flow compiles Rigel into an intermediate representation, called Systolic, which is translated into Verilog modules.

RIPL: Rathlin Image Processing Language (RIPL) [160, 161] is an open-source declarative DSL for memory-efficient FPGA-based image processing pipelines. RIPL provides the designers with stream combinator primitives, or *algorithmic skeletons*, to specify and compose image processing kernels. These skeletons capture common data access patterns (e.g., filters, convolutions, and map operations). Their composition defines a dataflow graph that the RIPL compiler can analyze to extract and minimize the on-chip memory requirements. From this analysis, RIPL produces a sequence of small computational modules connected by FIFOs. Finally, RIPL leverages Xronos [12], an open-source HLS tool for the RVC-CAL language, to generate RTL code for Xilinx FPGAs.

Halide-HLS: Halide-HLS extends the Halide language and compilation framework [138] to accelerate image processing pipelines on Xilinx Zynq SoCs [133]. Halide is an open-source image processing DSL that decouples the computation from the scheduling. Since Halide was developed for CPUs and GPUs, Halide-HLS provides Halide with additional scheduling commands to control some crucial aspects of the resulting FPGA designs, like the depth of the FIFOs between kernels. Moreover, the overall framework eases the user in the HW/SW partitioning task and generates an integration infrastructure of kernel drivers and APIs that enable an easy-to-use accelerator. Finally, this infrastructure increases the cooperation within a heterogeneous device where CPU and FPGA can work concurrently on different computation steps [133].

HeteroHalide: HeteroHalide [94] is an end-to-end solution that compiles Halide programs to FPGA designs simplifying the overall process. The compilation flow takes an algorithm defined in Halide and its scheduling commands and translates them into code for HeteroCL [82] (described in

Section 4.3). In particular, HeteroHalide exploits HeteroCL as an Intermediate Representation (IR) and its multiple backends to generate FPGA accelerators. The authors also extended the Halide scheduling commands with lazy transformations. In this way, Halide does not implement the command directly into its IR but explicitly lowers it to HeteroCL to produce efficient code.

4.2 Architectural Domain

The second cluster of DSLs shifts the focus from the application level to the architectural one. Indeed, they implement a particular architectural model/template and offer features and constructs to support classes of algorithms that benefit from it. Eventually, the compiler takes the input code and customizes the underlying architecture to better fit it.

SuSy: Born from a joint effort from Cornell, Intel, and UCLA, SuSy is a framework that comprises a DSL and a compilation infrastructure for productively building Systolic Arrays [76] on FPGAs [83]. Since Systolic Arrays are a killer application for spatial architectures, this framework extends the Halide OpenCL generation framework to tailor those specific classes of computations. The extended programming framework exploits the inherited decoupling of computation and scheduling from Halide [138] and proposes a DSL based on Uniform Recurrence Equations (URE) [72] for space-time transformations [77, 86]. Besides, SuSy provides a set of spatial optimization primitives that do not require any change in the algorithm structure but offer several spatial mappings. Finally, the compilation framework maps the application on a Systolic Array architecture based on a shift-register processing element designed in OpenCL for Intel FPGAs.

Spatial: Spatial [73, 169] is an open-source DSL and compiler for the design of spatial accelerators targeting FPGAs, CGRAs, and ASICs. Spatial provides various target-agnostic abstractions able to boost both productivity and design performance, while the compiler manages most of the optimizations. Spatial compiles the code to C++ (host) and Chisel (accelerator). According to the target device, Spatial relies on either Xilinx and Intel's toolchains, Plasticine CGRA [132], or Synopsys tools. Spatial builds upon four criteria that the developers considered necessary to offer a good balance between productivity and performance, namely *control*, *memory hierarchy*, *host interfaces*, and *DSE*. First, Spatial provides multiple control structures to enable the designers to describe the accelerator architecture briefly. Such constructs range from finite-state machines and loops to streaming and parallel ones. Then, the memory hierarchy supplies various memory templates to abstract and yet control data allocation on both on-chip and off-chip memories. Within Spatial code, the designer can implement both the accelerator and the host, leveraging constructs that abstract the underlying communication interfaces between host and target device. Finally, even though the Spatial compiler automatically optimizes the control and memory constructs according to statically inferable information, Spatial also provides a DSE engine based on the HyperMapper [13] machine learning framework.

SPGen: Streaming Processing Generator (SPGen) is a DSL framework for designing streaming processing FPGA accelerators based on OpenACC in the HPC context [145, 180]. Given an OpenACC code, the framework analyzes it and extracts both an OpenCL host and kernel code. Next, the framework implements the kernel converting body loops into SPGen code, which is then translated into an HDL module. Given the lack of support for memory access of SPGen, the framework also instantiates an OpenCL kernel wrapper to enable memory access. The final result is an OpenCL code with an HDL-based IP that leverages Intel FPGA SDK for OpenCL to produce the bitstream.

4.3 Intermediate Infrastructure for DSLs

The third and last cluster covers those solutions that propose an intermediate layer lying between the DSL and the RTL/HLS code. Developers can rely on such a layer to design new DSLs or extend existing ones to support FPGAs as target devices. In this way, the intermediate infrastructure

further decouples the code development from the hardware-related translation and optimization, increasing the compilation flow modularity.

Infrastructure for Delite-based DSLs: Based on an extension of the Delite compilation framework [89, 162], George et al. [51] proposed a compiler infrastructure generating a complete FPGA-based system from an application expressed in Delite-based DSLs (e.g., OptiML [163]). This infrastructure leverages Delite to automatically extract a set of computational patterns (that they also call kernels) and a dependency graph. Then, it generates the hardware based on an architectural template, which accommodates both softcore and custom kernels, along with interconnection and interface components. The compiler maps serial kernels on the softcore. In contrast, it creates optimized HLS-based implementations for the parallel ones. The compiler produces several implementations, called variants, to provide several area-performance trade-offs. Nonetheless, it picks the top-performing one if it fits the resources budget. An exciting feature of this infrastructure is the ability to manage dynamic memory allocation within a maximum size. The final step, also called system synthesis, selects the kernel variant, interconnects every component within the architectural template, and creates the control circuitry for kernel scheduling and dynamic memory allocation management. In particular, if that memory overflows, the execution terminates immediately. After this step, the compiler follows the standard flow for Xilinx FPGAs to generate the final bitstream.

DHDL: Delite Hardware Definition Language (DHDL) [74, 131] is an intermediate language devised to describe hardware datapaths. More technically, DHDL is a DSL embedded in Scala, and designers can either write code directly in DHDL or target it from another DSL or high-level language. DHDL aims to generate efficient hardware implementations for FPGA from parallel patterns, like map, reduce, groupBy, etc. To this end, DHDL provides a set of parametrized architectural templates that capture parallelism, locality, and off- and on-chip access pattern details. DHDL represents the given program as a dataflow graph where the nodes are the selected templates, and the edges are the data dependencies. The analysis of the dataflow graph provides estimates of both cycle count and area usage. Moreover, DHDL offers a DSE tool that relies on such estimates to explore the design space. From the final design, DHDL emits MaxJ code for Maxeler's DFEs.

FROST: FROST [42, 43] is a common backend for DSLs that enables them to generate FPGA designs even though they do not support FPGAs natively. In particular, FROST targets data-parallel algorithms operating on dense arrays and tensors and provides an Intermediate Representation (IR) that DSLs can target. Alternatively, a designer can use one of the already supported DSLs, i.e., Halide [138] and Tiramisu [7]. FROST also exposes a scheduling language like Halide that provides various commands the designer can apply to optimize the code. Such scheduling commands affect the final design at multiple levels, from the computation (e.g., pipelining) and local memory usage (e.g., partitioning) to the overall architecture (e.g., dataflow). Given the input code and the scheduling commands, FROST optimizes the IR to produce C/C++ suitable for the SDAccel toolchain by Xilinx.

HeteroCL: HeteroCL [82] is a multi-paradigm programming infrastructure that decouples the algorithm specification from hardware customization. In particular, HeteroCL is a Python-based DSL extended from TVM [24], a tensor-oriented declarative DSL. In addition to TVM features, HeteroCL enables designers to exploit both hardware optimization techniques and an imperative paradigm. On the one hand, after specifying the algorithm, the designers can customize and optimize three hardware components: compute (e.g., loop transformation and parallelization), data type (e.g., bit-accurate types and quantization schemes), and memory (e.g., partitioning and reuse buffers). In this way, the designers can easily explore the performance/area and accuracy trade-offs. On the other hand, the support for the imperative paradigm permits the designers to implement algorithms not suitable for a declarative one (e.g., sorting algorithms). Finally, HeteroCL features three different backends: a general backend that compiles to code for HLS tools, a stencil backend based on the SODA framework [25], and a Systolic Array backend based on the PolySA framework [33].

LIFT: Aiming at performance portability across heterogeneous architectures [159], LIFT [75] is a code generation framework building upon a high-level, data-parallel intermediate language. A designer can either target LIFT parallel primitives from DSLs and software libraries or directly write code in LIFT to generate code for FPGAs, CPUs, GPUs. FPGA backend takes the input computations expressed as lambda functions and translates them into VHDL code. At the same time, the remaining part becomes the host code on a Xilinx Zynq system. LIFT also provides an automatic design space exploration based on a rewriting rule methodology that applies optimization to the code. The proposed mapping optimizations are tiling, vectorization, and what the authors call coarse-grained parallelism. For instance, considering a dot-product computation, tiling divides the data processed in a loop improved locality, vectorization operates on multiple data simultaneously, and coarse-grained parallelism replicates the same dataflow numerous times, i.e., multi-row processing.

AnyHLS: AnyHLS [123] is an open-source [122] framework for the design of high-level and modular domain-specific libraries targeting HLS. It builds on top of AnyDSL [92], a compiler framework for the functional language Impala that leverages partial evaluation [49] and shallow embedding [91]. The former enables the optimization of algorithm variants at compile-time, the latter supports the addition of domain-specific structures without affecting the compiler. AnyHLS inherits such features from AnyDSL and exploits them to build abstractions for FPGA design. Specifically, AnyHLS offers design utilities about loop transformations, reductions, finite state machines, and memory types and abstractions. Given a functional description of the algorithm in Impala, AnyHLS combines the previously mentioned utilities and then synthesizes optimized HLS code for either Vivado HLS or Intel SDK for OpenCL.

Calyx: Aiming at combining high-level abstraction and control flow details of HLS imperative languages and HDL structural details and high performance, Calyx is an intermediate language that provides a shared compilation infrastructure to quickly design and deploy computational accelerators in Verilog [118]. Calyx provides a higher level of abstraction than IR for RTL languages [66] and grants precise control over scheduling logic generation, borrowing the decoupling of algorithm and scheduling from Halide [138] while explicitly representing low-level resources. Calyx is not tied to any specific hardware design methodology, and it provides a general infrastructure to let new DSL-to-RTL be fastly prototyped. Indeed, the developers present two frontends based on a Systolic Array generator and Dahlia, an annotation language for predictable HLS accelerators designs. However, Calyx does not provide any target-specific optimizations, e.g., mux cost in ASIC or FPGA designs [118], and does not guarantee any feasibility on the design implementation.

4.4 Summary

Table 3 summarizes the main features of the tools we described in this Section. As reported by the second column, most owners are universities, which usually developed such DSLs for internal research projects or collaborations with other institutes/companies and then released them as open source. However, some of these languages are currently not maintained anymore, according to the last updates on their repositories. In terms of the input language, while some developers designed a new language from scratch (e.g., Rigel, RIPL), the majority leverage existing languages and adapt their structure to target FPGAs, especially when considering the first two clusters. On the other hand, developers can exploit the intermediate infrastructure of the third cluster to either build a new language that directly supports FPGAs or add FPGA support to multiple existing languages. Moving to the domain, the first cluster of DSLs mainly focuses on image and packet processing domains, in which FPGAs are particularly effective. The same holds for the second cluster, shifting the specialization from the application level to the architectural one. Finally, the third cluster broads the supported domains thanks to their agnostic approach.

Table 3. Comparison table of the presented DSLs for FPGA design.

	Tool	Owner	Input Language	Domain	Output	Target Vendor	Other Relevant Features
Application Domain	Spiral	CMU	OL	Signal Processing	Verilog	Xilinx	ASIC, CPU, GPU
	P4-to-VHDL	CESNET a.l.e. Netcope Tech.	P4	Packet Processing	VHDL	Xilinx	-
	Hipacc†	Nürnberg Saarland	C++ DSL	Image Processing	OpenCL/C++	Intel, Xilinx	CPU, GPU
	Darkroom†	Stanford	Terra	Image Processing	Verilog	Xilinx	CPU and ASIC Support, Automatic Scheduling
	ExaSlang 4	ExaStencils Consortium	ExaSlang 4	Numerical Solvers	C/C++	Xilinx	Abstraction Layers, Level Specification
	Rigel†‡★	Stanford	Rigel	Image Processing	Verilog	Xilinx	Simulation by Terra
	RIPL†‡	Heriot-Watt University	RIPL	Image Processing	Verilog	Xilinx	Algorithmic Skeletons
	P4FPGA†	P4FPGA Project	P4	Packet Processing	BSV	Xilinx	-
	Halide-HLS†	Stanford, Berkeley	Halide	Image Processing	C/C++	Xilinx	Automatic Integration (APIs, Drivers)
	P4HLS†	Montréal	P4	Packet Processing	C++	Xilinx	-
	HeteroHalide†‡	UCLA	Halide	Image Processing	HeteroCL	Intel, Xilinx	Lazy Transformations
Architectural Domain	SuSy	Cornell, Intel, UCLA	URE	Systolic Arrays	OpenCL	Intel	Extends the Halide OpenCL Generation Framework
	Spatial†‡★	Stanford	Spatial	Application Accelerators	Chisel	Intel, Xilinx	DSE, CGRA and ASIC Support
	SPGen	Riken Center, University Tsukuba	OpenACC	Streaming Processing	OpenCL, SPGen	Intel	-
Intermediate Infrastructure for DSLs	Infrastructure for Delite-based DSLs	EPFL, Stanford	OptiML	Delite Languages	Bitstream	Xilinx	Dynamic Memory Allocation
	DHDL	Stanford	DHDL, other DSLs	Parallel Patterns	MaxJ	Intel, Maxeler's DFE	Cycle Count and Area Estimates, DSE
	FROST	PoliMI, MIT	Halide, Tiramisu	Data-Parallel	C/C++	Xilinx	Scheduling Commands
	HeteroCL†‡★	UCLA, Cornell University	HeteroCL	Any	C/C++, SODA, PolySA	Intel, Xilinx	Decoupling of Algorithm and HW Customization
	LIFT†‡	Edingburgh Glasgow	LIFT	Functional Patterns	VHDL	Xilinx	Host Generation, Automatic DSE
	AnyHLS†‡	University Erlangen-Nürnberg	Impala	Any	C/C++/OpenCL	Intel, Xilinx	Partial Evaluation, Shallow Embedding
	Calyx†‡★	Cornell University	Systolic Arrays, Dahila, TVM	Any	Verilog	Xilinx	Support for Any DSL

† Open-source ‡ Last update in the last years (2020-2021) ★ Maintained, to the best of authors' knowledge

After processing the input code, most DSLs generate either RTL or code for HLS/ACS tools, whereas just one DSL offers a full flow that produces the bitstream as output. Therefore, they all need to interact with commercial tools to complete the design flow. According to the point where this interaction starts, this dependency may become more and more binding. Consequently, the DSL infrastructure may require continuous updates to keep pace with FPGA tool changes. In this scenario, a modular approach similar to the third DSL cluster eases maintaining the language.

Summing up, the domain specialization introduces an additional abstraction layer that impacts the design for FPGAs at multiple levels. On the one hand, it reduces the steepness of the FPGA learning curve, mainly requiring domain knowledge to write the algorithm in a given language. On the other hand, it automates various analyses and optimizations that, otherwise, would take too much time in a multi-domain scenario, relieving the designer from this burden. Nonetheless, making a DSL successful is no easy task as, at first glance, many DSLs look similar, especially when considering the same domain. We believe that maintaining a language and building a community around it, especially if open source, is paramount for its success. Likewise, being a vendor-agnostic DSL and supporting heterogeneous architectures help cover a broader range of designers. Even though this aspect potentially implies a higher complexity, a modular structure may ease addressing it thanks to the decoupling of the internal components. Finally, despite the automatic optimizations, we believe that a DSL would be more comprehensive if it exposed a manual optimization flow (e.g., a scheduling language like in Halide) that expert designers may exploit to hand-tune their designs.

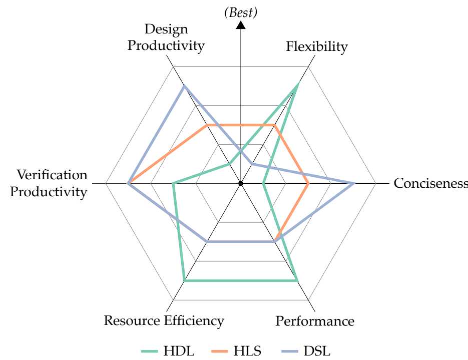

Fig. 7. A qualitative comparison of HDLs, HLS tools, and DSLs over three levels (the arrangement does not say anything about a quantitative comparison when two or more solutions belong to the same level).

5 CONCLUSIONS AND FINAL REMARKS

FPGAs are becoming increasingly pervasive in the computing landscape, from small low-power embedded systems to large-scale datacenters [17, 52, 168]. Similar to what happened with programming languages and frameworks for general-purpose processors [102], tools for FPGA hardware design changed and evolved according to the developer's needs and target contexts. Indeed, modern HDLs and HLS tools offer a new level of abstraction and productivity than (System)Verilog and VHDL. On the one hand, hoisting of abstraction level allows to reduce the design time, and facilitate IP reuse, customization, and verification. On the other hand, it makes the FPGA learning curve smoother for non-hardware designers. Examples are ACS tools, which offer an efficient HLS environment and completely abstract and automatize the hardware design flow, hiding the complexity of the system-level design. Despite the high level of abstraction provided, both HDLs and HLS tools require knowledge and familiarity with hardware design. The possibility to use high-level languages like C, C++, and OpenCL is for sure advantageous over HDLs; however, such languages are not natively designed to describe hardware. While HLS compilers do a great job in translating and scheduling high-level languages into hardware RTL, the designer must clearly know the desired hardware architecture and implement it consequently to ease the HLS process. Moreover, manual optimization through options and directives is often required to guide this process further, increasing the productivity level than HDLs at the cost of a lower design control and a time-consuming exploration [81, 116]. Therefore, DSLs emerge to overcome these drawbacks and exploit the narrowing of the domain specialization. In this way, hardware developers benefit from specialized toolchains and infrastructures, optimized architectural templates, and expressing computations more simply and intuitively. Figure 7 shows a qualitative evaluation of HDLs, HLS tools, and DSLs according to six metrics[5] that combine our analysis and literature comparisons on single case studies [51, 128, 143]. The chart highlights how HDLs offer the best solution in terms of flexibility, performance, and resource efficiency at the cost of design and verification time, and conciseness. Conversely, DSLs boost productivity and conciseness and deliver good design quality. Finally, HLS represents an adequate trade-off between HDLs and DSLs.

Overall, the community efforts push towards a constantly increasing abstraction of FPGAs' programmability to ease their usage and open to a broader public. Indeed, many of the toolchains we described in this manuscript and other domain-specific toolchains that do not leverage a DSL as

[5] Appendix C and Table 5 define the six qualitative metrics.

intended in this manuscript (such as machine learning ones [39, 172, 175] or general IR [154]) push the FPGA democratization. Although these toolchains cover several application fields — acceleration mainly —, we believe a low-level component (HDL) is still necessary to devise a complete design experience. For instance, designing processors [1, 2, 28, 31, 127] or components at the analog/digital boundaries is not straightforward through HLS or DSLs.

We identify six big takeaways from the body of knowledge presented in this survey.

Takeaway 1: The field of digital design abstraction for FPGAs is cyclical. From a very specific language for hardware description (i.e., VHDL and Verilog), we move to HLS with the employment of generic C language (after two failure tools generations [104]), and we finally find DSLs with their extreme specificity for given computations.

Takeaway 2: HDLs will be a constant standard, and researchers will foster further improvements, given their fundamental role in hardware design and verification. They still represent the most efficient hardware design methodology, though highly time-consuming. (System)Verilog and VHDL will not be the unique way for hardware description [99], given the wide variety of high-level HDLs (as discussed in Section 2). However, a designer will think in low-level hardware components or at the RTL level with wires and registers. In the future, we envision a further bloom of high-level generators for architectures [8, 193] or more complex systems [1, 80].

Takeaway 3: The introduction of efficient HLS tools broaden the userbase of FPGA systems, enabling non-expert hardware designers to this technology. However, these tools move the complexity from the IP design towards the system-level design. Therefore, ACS tools come into play to automate this flow, particularly useful for accelerators design. Nevertheless, custom designs beyond accelerators exclude ACS and require low-level hardware design knowledge. We envision that both HLS and ACS tools will remain essential for enabling a faster hardware development cycle than HDLs, especially for accelerators or application domains continuously evolving [88].

Takeaway 4: DSLs fall at the extreme of specialization, where the designer needs algorithm or applicative domain knowledge, and the compiler takes the burden of efficiently implement the hardware. Although they are highly beneficial in targeting specific domains (both applicative and architectural), at the current status DSLs heavily rely on the toolchains on top of which they build, creating a tight bind with DSL and tool.

Takeaway 5: Among DSL clusters, the intermediate infrastructure one represents an exciting trend for hardware design. Indeed, this approach embodies a trade-off between generality and specificity. Besides, its modularity enables researchers to build general-purpose languages that run on CPUs and FPGAs [154]. Undoubtedly, this trend will require a specialized compilation and toolchain flow for a hardware design that can be vendor-agnostic and open, similar to LLVM [85].

Takeaway 6: What comes naturally after the fifth takeway is our belief in a future open FPGA ecosystem. Similarly to the RISC-V revolution [3], we envision a open ecosystem for FPGAs[6] from the architecture itself [93, 167] to the design toolchains [115, 153]. Indeed, open-source FPGA-based projects are more and more, and even some vendors started to open their toolchains [87, 136, 181].

In conclusion all these three abstraction efforts are necessary as they cover different aspects of digital design; thus, we believe the community will continuously push the research in the FPGA programmability field, from low-level RTL design to domain-specific abstractions.

ACKNOWLEDGEMENTS

The authors are grateful for feedbacks from Reviewers and NECSTLab members, with a particular mention to A. Damiani, A. Parravicini, E. D'Arnese, F. Carloni, F. Peverelli, and R. Brondolin.

[6]https://osfpga.org/

REFERENCES

[1] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* 40, 4 (2020), 10–21. https://doi.org/10.1109/MM.2020.2996616

[2] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. 2016. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* (2016).

[3] Krste Asanović and David A Patterson. 2014. Instruction sets should be free: The case for risc-v. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146* (2014).

[4] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. 2010. Cλash: Structural descriptions of synchronous hardware using haskell. In *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*. IEEE, 714–721.

[5] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. 2012. Chisel: Constructing hardware in a Scala embedded language. In *DAC Design Automation Conference 2012*. 1212–1221.

[6] John Backus. 1978. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM* 21, 8 (1978), 613–641.

[7] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE.

[8] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, et al. 2016. OpenPiton: An open source manycore research framework. *ACM SIGPLAN Notices* 51, 4 (2016), 217–232.

[9] Shunning Jiang Christopher Torng Christopher Batten. 2018. An Open-Source Python-Based Hardware Generation, Simulation, and Verification Framework. In *Workshop on Open-Source EDA Technology (WOSET'18)*. 1–5.

[10] Peter Bellows and Brad Hutchings. 1998. JHDL - An HDL for Reconfigurable Systems. In *IEEE Symposium on FPGAs for Custom Computing Machines*. IEEE.

[11] Pavel Benácek, Viktor Pu, and Hana Kubátová. 2016. P4-to-VHDL: Automatic generation of 100 gbps packet parsers. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE.

[12] Endri Bezati, Marco Mattavelli, and Jorn W Janneck. 2013. High-level synthesis of dataflow programs for signal processing systems. In *2013 8th International Symposium on Image and Signal Processing and Analysis (ISPA)*. IEEE.

[13] Bruno Bodin, Luigi Nardi, M. Zeeshan Zia, Harry Wagstaff, Govind Sreekar Shenoy, Murali Emani, John Mawer, Christos Kotselidis, Andy Nisbet, Mikel Lujan, Björn Franke, Paul H.J. Kelly, and Michael O'Boyle. 2016. Integrating algorithmic parameters into benchmarking and design space exploration in 3D scene understanding. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation* (Haifa, Israel) *(PACT '16)*. Association for Computing Machinery, New York, NY, USA, 57–69. https://doi.org/10.1145/2967938.2967963

[14] Thomas Bollaert. 2008. Catapult synthesis: a practical introduction to interactive C synthesis. In *High-Level Synthesis*. Springer, 29–52.

[15] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[16] Thomas Bourgeat, Clément Pit-Claudel, and Adam Chlipala. 2020. The essence of Bluespec: a core language for rule-based hardware design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 243–257.

[17] Andrew Boutros and Vaughn Betz. 2021. FPGA Architecture: Principles and Progression. *IEEE Circuits and Systems Magazine* 21, 2 (2021), 4–29.

[18] Jakub Cabal, Pavel Benáček, Lukáš Kekely, Michal Kekely, Viktor Puš, and Jan Kořenek. 2018. Configurable FPGA packet parser for terabit networks with guaranteed wire-speed throughput. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 249–258.

[19] Cadence. 2021. Stratus High-Level Synthesis. https://www.cadence.com/ko_KR/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html.

[20] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, Association for Computing Machinery, New York, NY, USA, 33–36.

[21] Joao MP Cardoso, Pedro C Diniz, and Markus Weinhardt. 2010. Compiling for reconfigurable computing: A survey. *ACM Computing Surveys (CSUR)* 42, 4 (2010), 1–65.

[22] Riccardo Cattaneo, Giuseppe Natale, Carlo Sicignano, Donatella Sciuto, and Marco Domenico Santambrogio. 2015. On how to accelerate iterative stencil loops: a scalable streaming-based approach. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 4 (2015), 1–26.

[23] Raghunandan Chaware, Kumar Nagarajan, and Suresh Ramalingam. 2012. Assembly and reliability challenges in 3D integration of 28nm FPGA die on a large high density 65nm passive interposer. In *2012 IEEE 62nd Electronic Components and Technology Conference*. IEEE, 279–283.

[24] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 578–594.

[25] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. 2018. SODA: stencil with optimized dataflow architecture. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[26] Michael D Ciletti. 2003. *Advanced digital design with the Verilog HDL*. Vol. 1. Prentice Hall Upper Saddle River.

[27] John Clow, Georgios Tzimpragos, Deeksha Dangwal, Sammy Guo, Joseph McMahan, and Timothy Sherwood. 2017. A pythonic approach for rapid hardware prototyping and instrumentation. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–7.

[28] Alessandro Comodi, Davide Conficconi, Alberto Scolari, and Marco D Santambrogio. 2018. TiReX: Tiled regular expression matching architecture. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 131–137.

[29] Katherine Compton and Scott Hauck. 2002. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys (csuR)* 34, 2 (2002), 171–210.

[30] Davide Conficconi, Eleonora D'Arnese, Emanuele Del Sozzo, Donatella Sciuto, and Marco D Santambrogio. 2021. A Framework for Customizable FPGA-based Image Registration Accelerators. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Association for Computing Machinery, New York, NY, USA, 251–261.

[31] Davide Conficconi, Emanuele Del Sozzo, Filippo Carloni, Alessandro Comodi, Alberto Scolari, and Marco Domenico Santambrogio. 2022. An Energy-Efficient Domain-Specific Architecture for Regular Expressions. *IEEE Transactions on Emerging Topics in Computing* (2022). https://doi.org/10.1109/TETC.2022.3157948

[32] Jason Cong, Vivek Sarkar, Glenn Reinman, and Alex Bui. 2010. Customizable domain-specific computing. *IEEE Design & Test of Computers* 28, 2 (2010), 6–15.

[33] Jason Cong and Jie Wang. 2018. PolySA: Polyhedral-based systolic array auto-compilation. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[34] Achronix Semiconductor Corporation. 2020. Speedster7t Network on Chip User Guide (UG089) . https://tinyurl.com/achronixnoc. Last accessed: June 15th 2021.

[35] Philippe Coussy, Cyrille Chavet, Pierre Bomel, Dominique Heller, Eric Senn, and Eric Martin. 2008. GAUT: A high-level synthesis tool for DSP applications. In *High-Level Synthesis*. Springer, 147–169.

[36] Philippe Coussy, Daniel D Gajski, Michael Meredith, and Andres Takach. 2009. An introduction to high-level synthesis. *IEEE Design & Test of Computers* 26, 4 (2009), 8–17.

[37] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 95–105.

[38] Tomasz S Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P Singh. 2012. From OpenCL to high-performance hardware on FPGAs. In *22nd international conference on field programmable logic and applications (FPL)*. IEEE, 531–534.

[39] Andrea Damiani, Emanuele Del Sozzo, and Marco D Santambrogio. 2022. Large Forests and Where to "Partially" Fit Them. In *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 550–555.

[40] Jan Decaluwe. 2004. MyHDL: a Python-Based Hardware Description Language. *Linux journal* 127 (2004), 84–87.

[41] André DeHon. 2000. The density advantage of configurable computing. *Computer* 33, 4 (2000), 41–49.

[42] Emanuele Del Sozzo, Riyadh Baghdadi, Saman Amarasinghe, and Marco D Santambrogio. 2017. A Common Backend for Hardware Acceleration on FPGA. In *Computer Design (ICCD), 2017 IEEE International Conference on*. IEEE, 427–430.

[43] Emanuele Del Sozzo, Riyadh Baghdadi, Saman Amarasinghe, and Marco D Santambrogio. 2018. A unified backend for targeting fpgas from dsls. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 1–8.

[44] Emanuele Del Sozzo, Marco Rabozzi, Lorenzo Di Tucci, Donatella Sciuto, and Marco D Santambrogio. 2018. A scalable FPGA design for cloud n-body simulation. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 1–8.

[45] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. 2013. Terra: A Multi-Stage Language for High-Performance Computing. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 105–116.

[46] ECE Department, University of Toronto. 2021. LegUp High-Level Synthesis. https://github.com/wincle626/HLS_Legup. Last accessed: March 16th 2021.
[47] Martin Fowler. 2010. *Domain-specific languages.* Pearson Education.
[48] Franz Franchetti, Tze Meng Low, Doru Thom Popovici, Richard M Veras, Daniele G Spampinato, Jeremy R Johnson, Markus Püschel, James C Hoe, and José MF Moura. 2018. SPIRAL: Extreme performance portability. *Proc. IEEE* 106, 11 (2018), 1935–1968.
[49] Yoshihiko Futamura. 1983. Partial computation of programs. In *RIMS Symposia on Software Science and Engineering.* Springer, 1–35.
[50] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. 2019. Xilinx adaptive compute acceleration platform: VersalTM architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* 84–93.
[51] Nithin George, HyoukJoong Lee, David Novo, Tiark Rompf, Kevin J Brown, Arvind K Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. 2014. Hardware system synthesis from domain-specific languages. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL).* IEEE, 1–8.
[52] Maya Gokhale and Lesley Shannon. 2021. FPGA Computing. *IEEE Micro* 41, 4 (2021), 6–7.
[53] Google. 2021. XLS: Accelerated HW Synthesis. https://google.github.io/xls/.
[54] Zhi Guo, Walid Najjar, Frank Vahid, and Kees Vissers. 2004. A quantitative analysis of the speedup factors of FPGAs over processors. In *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays.* 162–170.
[55] Robert Harper, David MacQueen, and Robin Milner. 1986. *Standard ml.* Department of Computer Science, University of Edinburgh.
[56] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. 2014. Darkroom: compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph.* 33, 4 (2014), 144–1.
[57] James Hegarty, Ross Daly, Zachary DeVito, Jonathan Ragan-Kelley, Mark Horowitz, and Pat Hanrahan. 2016. Rigel: Flexible Multi-Rate Image Processing Hardware. *ACM Trans. Graph.* 35, 4, Article 85 (July 2016), 11 pages.
[58] John L Hennessy and David A Patterson. 2019. A new golden age for computer architecture. *Commun. ACM* 62, 2 (2019), 48–60.
[59] Steven F Hoover. 2017. Timing-abstract circuit design in transaction-level Verilog. In *2017 IEEE International Conference on Computer Design (ICCD).* IEEE, 525–532.
[60] Lan Huang, Da-Lin Li, Kang-Ping Wang, Teng Gao, and Adriano Tavares. 2020. A survey on performance optimization of high-level synthesis tools. *Journal Of Computer Science and Technology* 35 (2020), 697–720.
[61] Maxeler Inc. 2021. Multiscale Dataflow Programming. https://www.maxeler.com/products/software/maxcompiler/.
[62] Intel. 2020. Intel Quartus Documentation. https://www.intel.com/content/www/us/en/programmable /products/design-software/fpga-design/quartus-prime/user-guides.html.
[63] Intel. 2021. Intel FPGA SDK for OpenCL. https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html.
[64] Intel. 2021. Intel HLS Compiler. https://www.intel.it/content/www/it/it/software/programmable/quartus-prime/hls-compiler.html.
[65] Intel Inc. 2021. Intel oneAPI. https://software.intel.com/content/www/us/en/develop/tools/oneapi.html.
[66] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. 2017. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD).* 209–216. https://doi.org/10.1109/ICCAD.2017.8203780
[67] Ricardo Jasinski. 2016. *Effective coding with VHDL: principles and best practice.* MIT Press.
[68] Shunning Jiang, Berkin Ilbeyi, and Christopher Batten. 2018. Mamba: closing the performance gap in productive hardware development frameworks. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC).* IEEE, 1–6.
[69] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. 2019. DYNAMATIC - Dynamically Scheduled High-Level Synthesis. https://github.com/lana555/dynamatic
[70] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. 2020. Invited Tutorial: Dynamatic: From C/C++ to Dynamically Scheduled Circuits. In *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* 1–10.
[71] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture.* 1–12.
[72] Richard M Karp, Raymond E Miller, and Shmuel Winograd. 1967. The organization of computations for uniform recurrence equations. *Journal of the ACM (JACM)* 14, 3 (1967), 563–590.

[73] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. 2018. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 296–311.

[74] D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun. 2016. Automatic Generation of Efficient Accelerators for Reconfigurable Hardware. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 115–127. https://doi.org/10.1109/ISCA.2016.20

[75] Martin Kristien, Bruno Bodin, Michel Steuwer, and Christophe Dubach. 2019. High-level synthesis of functional patterns with Lift. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*. 35–45.

[76] Hsiang Tsung Kung and Charles E Leiserson. 1978. *Systolic Arrays for (VLSI)*. Technical Report. Carnegie-Mellon University, Pittsburgh, PA, Department of Computer Science.

[77] S Kung. 1985. VLSI array processors. *IEEE ASSP Magazine* 2, 3 (1985), 4–22.

[78] Ian Kuon and Jonathan Rose. 2007. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on computer-aided design of integrated circuits and systems* 26, 2 (2007), 203–215.

[79] Ian Kuon, Russell Tessier, and Jonathan Rose. 2008. *FPGA architecture: Survey and challenges*. Now Publishers Inc.

[80] Andreas Kurth, Pirmin Vogel, Alessandro Capotondi, Andrea Marongiu, and Luca Benini. 2017. HERO: Heterogeneous embedded research platform for exploring RISC-V manycore accelerators on FPGA. *arXiv:1712.06497* (2017).

[81] Sakari Lahti, Panu Sjövall, Jarno Vanne, and Timo D Hämäläinen. 2018. Are we there yet? A study on the state of high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 5 (2018).

[82] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. 2019. HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 242–251.

[83] Yi-Hsiang Lai, Hongbo Rong, Size Zheng, Weihao Zhang, Xiuping Cui, Yunshan Jia, Jie Wang, Brendan Sullivan, Zhiru Zhang, Yun Liang, et al. 2020. SuSy: a programming model for productive construction of high-performance systolic arrays on FPGAs. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.

[84] William Landi. 1992. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)* 1, 4 (1992), 323–337.

[85] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE, 75–86.

[86] Dominique Lavenier, Patrice Quinton, and Sanjay Rajopadhye. 1999. Advanced systolic design. *Digital Signal Processing for Multimedia Systems* (1999), 657–692.

[87] C. Lavin and A. Kaviani. 2018. RapidWright: Enabling Custom Crafted Implementations for FPGAs. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 133–140.

[88] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (28 05 2015), 436–444.

[89] HyoukJoong Lee, Kevin Brown, Arvind Sujeeth, Hassan Chafi, Tiark Rompf, Martin Odersky, and Kunle Olukotun. 2011. Implementing domain-specific languages for heterogeneous parallel computing. *Ieee Micro* 31, 5 (2011), 42–53.

[90] LegUp Computing. 2019. High-Level Synthesis For Any FPGA. https://www.legupcomputing.com/.

[91] Roland Leißa, Klaas Boesche, Sebastian Hack, Richard Membarth, and Philipp Slusallek. 2015. Shallow embedding of DSLs via online partial evaluation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. Association for Computing Machinery, New York, NY, USA, 11–20.

[92] Roland Leißa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. 2018. AnyDSL: A partial evaluation framework for programming high-performance libraries. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.

[93] Ang Li and David Wentzlaff. 2019. PRGA: An open-source framework for building and using custom FPGAs. In *The First Workshop on Open-Source Design Automation; Florence, Italy*. 1–6.

[94] Jiajie Li, Yuze Chi, and Jason Cong. 2020. HeteroHalide: From image processing DSL to efficient FPGA acceleration. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 51–57.

[95] Yanbing Li and Miriam Leeser. 2000. HML, a Novel Hardware Description Language and Its Translation to VHDL. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 8, 1 (2000), 1–8.

[96] Olav Lindtjorn, Robert G Clapp, Oliver Pell, Oskar Mencer, and Michael J Flynn. 2010. Surviving the end of scaling of traditional microprocessors in HPC. *IEEE Hot Chips* 22 (2010), 22–24.

[97] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. 2019. A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications. *ACM Computing Surveys (CSUR)* 52, 6 (2019), 1–39.

[98] Yanqiang Liu, Yao Li, Zhengwei Qi, and Haibing Guan. 2019. A scala based framework for developing acceleration systems with FPGAs. *Journal of Systems Architecture* 98 (2019), 231–242.

[99] Derek Lockhart, Stephen Twigg, Ravi Narayanaswami, Jeremy Coriell, Uday Dasari, Richard Ho, Doug Hogberg, George Huang, Anand Kane, Chintan Kaur, Tao Liu, Adriana Maggiore, Kevin Townsend, and Emre Tuncer. 2018. Experiences Building Edge TPU with Chisel. https://www.youtube.com/watch?v=x85342Cny8c.

[100] Derek Lockhart, Gary Zibrat, and Christopher Batten. 2014. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. In *47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE.

[101] Lombiq Technologies. 2019. Hastlayer SDK - GitHub. https://github.com/Lombiq/Hastlayer-SDK.

[102] Kenneth C Louden and Kenneth A Lambert. 2011. *Programming languages: principles and practices*. Cengage Learning.

[103] Steven Margerm, Amirali Sharifian, Apala Guha, Arrvindh Shriraman, and Gilles Pokam. 2018. TAPAS: Generating parallel accelerators from parallel programs. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 245–257.

[104] Grant Martin and Gary Smith. 2009. High-level synthesis: Past, present, and future. *IEEE Design & Test of Computers* 26, 4 (2009), 18–25.

[105] Ali Mashtizadeh. 2007. PHDL: A Python Hardware Design Framework. https://dspace.mit.edu/handle/1721.1/41543.

[106] MathWorks Inc. 2021. HDL Coder. https://www.mathworks.com/help/pdf_doc/hdlcoder/hdlcoder_ug.pdf.

[107] Clive Maxfield. 2004. *The design warrior's guide to FPGAs: devices, tools and flows*. Elsevier.

[108] Wim Meeus, Kristof Van Beeck, Toon Goedemé, Jan Meel, and Dirk Stroobandt. 2012. An overview of today's high-level synthesis tools. *Design Automation for Embedded Systems* 16, 3 (2012), 31–51.

[109] Richard Membarth, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. 2012. Generating device-specific GPU code for local operators in medical imaging. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 569–581.

[110] Richard Membarth, Oliver Reiche, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. 2015. Hipa^{cc}: A domain-specific language and compiler for image processing. *IEEE Transactions on Parallel and Distributed Systems* 27, 1 (2015), 210–224.

[111] Richard Membarth, Oliver Reiche, Özkan Mehmet Akif, and Bo Qiao. 2013. Repo HIP^{acc}. https://github.com/hipacc/hipacc. Last accessed: March 31st 2021.

[112] Mentor Graphics. 2021. Catapult HLS. https://www.mentor.com/hls-lp/catapult-high-level-synthesis/.

[113] Mentor Graphics. 2021. Design Creation. https://www.mentor.com/products/fpga/hdl_design/.

[114] Peter Milder, Franz Franchetti, James C Hoe, and Markus Püschel. 2012. Computer generation of hardware for linear digital signal processing transforms. *ACM Transactions on Design Automation of Electronic Systems* 17, 2 (2012), 1–33.

[115] Kevin E Murray, Mohamed A Elgammal, Vaughn Betz, Tim Ansell, Keith Rothman, and Alessandro Comodi. 2020. SymbiFlow and VPR: An Open-Source Design Flow for Commercial and Novel FPGAs. *IEEE Micro* 40, 4 (2020), 49–57.

[116] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, et al. 2016. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 10 (2016), 1591–1604.

[117] NEC. 2011. CyberWorkBench: High Level Synthesis from C/C++/SystemC to ASIC/FPGA. https://www.nec.com/en/global/prod/cwb/index.html.

[118] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. 2021. A compiler infrastructure for accelerator generators. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 804–817.

[119] Rishiyur Nikhil. [n.d.]. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04.*

[120] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-dataflow acceleration. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 416–429.

[121] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. 2004. The Scala language specification.

[122] M Akif Özkan, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, Roland Leißa, Sebastian Hack, Jürgen Teich, and Frank Hannig. 2020. AnyHLS. https://github.com/AnyDSL/anyhls.

[123] M Akif Özkan, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, Roland Leißa, Sebastian Hack, Jürgen Teich, and Frank Hannig. 2020. AnyHLS: High-Level Synthesis With Partial Evaluation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3202–3214.

[124] Samir Palnitkar. 2003. *Verilog HDL: a guide to digital design and synthesis*. Vol. 1. Prentice Hall Professional.

[125] Panda Team. 2021. Bambu: A Free Framework for the High-Level Synthesis of Complex Applications. https://panda.dei.polimi.it/?page_id=31.

[126] C Papon. 2017. SpinalHDL: An alternative hardware description language. *FOSDEM* (2017).

[127] Daniele Parravicini, Davide Conficconi, Emanuele Del Sozzo, Christian Pilato, and Marco D Santambrogio. 2021. CICERO: A Domain-Specific Architecture for Efficient Regular Expression Matching. *ACM Transactions on Embedded Computing Systems (TECS)* 20, 5s (2021), 1–24.

[128] Maxime Pelcat, Cédric Bourrasset, Luca Maggiani, and François Berry. 2016. Design productivity of a high level synthesis compiler versus HDL. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. IEEE, 140–147.

[129] Christian Pilato and Fabrizio Ferrandi. 2013. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *Field Programmable Logic and Applications (FPL), 23rd International Conference on*. IEEE.

[130] Oron Port and Yoav Etsion. 2017. DFiant: A dataflow hardware description language. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–4.

[131] Raghu Prabhakar, David Koeplinger, Kevin J Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. 2016. Generating configurable hardware from parallel patterns. *Acm Sigplan Notices* 51, 4 (2016).

[132] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A reconfigurable architecture for parallel patterns. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 389–402.

[133] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. 2017. Programming Heterogeneous Systems from an Image Processing DSL. *ACM Trans. Archit. Code Optim.* 14, 3, Article 26 (Aug. 2017), 25 pages. https://doi.org/10.1145/3107953

[134] David Pursley and Tung-Hua Yeh. 2017. High-level low-power system design optimization. In *VLSI Design, Automation and Test (VLSI-DAT), 2017 International Symposium on*. IEEE, 1–4.

[135] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 13–24.

[136] QuickLogic. 2021. QuickLogic Open Reconfigurable Computing (QORC) MCU + eFPGA SoC Open Source Software Tools. https://www.quicklogic.com/software/qorc-mcu-efpga-fpga-open-source-tools/. Last accessed: July 2nd 2021.

[137] M. Rabozzi, G. Natale, E. Del Sozzo, A. Scolari, L. Stornaiuolo, and M. D. Santambrogio. 2017. Heterogeneous exascale supercomputing: The role of CAD in the exaFPGA project. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. 410–415. https://doi.org/10.23919/DATE.2017.7927025

[138] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices* 48, 6 (2013), 519–530.

[139] Parthasarathy Ranganathan, Daniel Stodolsky, Jeff Calow, Jeremy Dorfman, Marisabel Guevara, Clinton Wills Smullen IV, Aki Kuusela, Raghu Balasubramanian, Sandeep Bhatia, Prakash Chauhan, et al. 2021. Warehouse-scale video acceleration: co-design and deployment in the wild. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 600–615.

[140] Enrico Reggiani, Emanuele Del Sozzo, Davide Conficconi, Giuseppe Natale, Carlo Moroni, and Marco D Santambrogio. 2021. Enhancing the scalability of multi-fpga stencil computations via highly optimized hdl components. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 14, 3 (2021), 1–33.

[141] Enrico Reggiani, Eleonora D'Arnese, Andrea Purgato, and Marco D Santambrogio. 2017. Pearson Correlation Coefficient acceleration for modeling and mapping of neural interconnections. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 223–228.

[142] Oliver Reiche, M Akif Özkan, Richard Membarth, Jürgen Teich, and Frank Hannig. 2017. Generating FPGA-based image processing accelerators with Hipacc. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1026–1033.

[143] Oliver Reiche, Moritz Schmid, Frank Hannig, Richard Membarth, and Jürgen Teich. 2014. Code generation from a domain-specific language for C-based HLS of hardware accelerators. In *2014 international conference on hardware/software codesign and system synthesis (CODES+ ISSS)*. IEEE, 1–10.

[144] David I Rich. 2003. The evolution of SystemVerilog. *IEEE Annals of the History of Computing* 20, 04 (2003), 82–84.

[145] Kentaro Sano, Hayato Suzuki, Ryo Ito, Tomohiro Ueno, and Satoru Yamamoto. 2014. Stream processor generator for HPC to embedded applications on FPGA-based system platform. *arXiv preprint arXiv:1408.5386* (2014).

[146] Jeferson Santiago da Silva, François-Raymond Boyer, and JM Pierre Langlois. 2018. P4-compatible high-level synthesis of low latency 100 Gb/s streaming packet parsers in FPGAs. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 147–152.

[147] Jeferson Santiago da Silva, François-Raymond Boyer, and JM Pierre Langlois. 2018. Repo P4HLS. https://github.com/engjefersonsantiago/P4HLS. Last accessed: March 31st 2021.

[148] Simpei Sato and Kenji Kise. 2013. ArchHDL: a new hardware description language for high-speed architectural evaluation. In *2013 IEEE 7th International Symposium on Embedded Multicore Socs*. IEEE, 107–112.

[149] Tao B Schardl, William S Moses, and Charles E Leiserson. 2017. Tapir: Embedding fork-join parallelism into LLVM's intermediate representation. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel*

Programming. 249–265.

[150] Christian Schmitt, Moritz Schmid, Frank Hannig, Jürgen Teich, Sebastian Kuckuk, and Harald Köstler. 2015. Generation of multigrid-based numerical solvers for FPGA accelerators. In *Proceedings of the 2nd International Workshop on High-Performance Stencil Computations (HiStencils)*. 9–15.

[151] Sanjit Seshia, Albert Magyar, David Biancolin, John Koenig, Jonathan Bachrach, and Krste Asanovic. 2021. Golden Gate: Bridging The Resource-Efficiency Gap Between ASICs and FPGA Prototypes. (2021).

[152] Ofer Shacham, Sameh Galal, Sabarish Sankaranarayanan, Megan Wachs, John Brunhaver, Artem Vassiliev, Mark Horowitz, Andrew Danowitz, Wajahat Qadeer, and Stephen Richardson. 2012. Avoiding Game Over: Bringing Design to the Next Level. In *DAC Design Automation Conference*. IEEE, 623–629.

[153] David Shah, Eddie Hung, Clifford Wolf, Serge Bazanski, Dan Gisselquist, and Miodrag Milanovic. 2019. Yosys+ nextpnr: an open source framework from verilog to bitstream for commercial fpgas. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 1–4.

[154] Amirali Sharifian, Reza Hojabr, Navid Rahimi, Sihao Liu, Apala Guha, Tony Nowatzki, and Arrvindh Shriraman. 2019. μir-an intermediate representation for transforming and optimizing the microarchitecture of application accelerators. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 940–953.

[155] Sergey Shumarayev. 2017. Intel's 14nm Heterogeneous FPGA System-in-Package Platform. In *Hot Chips 29 Symp.*

[156] Silexica. 2021. SLX FPGA. https://www.silexica.com/products/slx-fpga/.

[157] Marco Siracusa, Marco Rabozzi, Emanuele Del Sozzo, Lorenzo Di Tucci, Samuel Williams, and Marco D. Santambrogio. 2020. A CAD-based methodology to optimize HLS code via the Roofline model. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–9.

[158] Wilson Snyder. 2004. Verilator and systemperl. In *North American SystemC Users' Group, Design Automation Conference*.

[159] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code. *ACM SIGPLAN Notices* 50, 9 (2015), 205–217.

[160] Robert Stewart, Kirsty Duncan, Greg Michaelson, Paulo Garcia, Deepayan Bhowmik, and Andrew M. Wallace. 2018. RIPL: A Parallel Image Processing Language for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems* 11, 1 (2018), 7:1–7:24.

[161] Robert Stewart, Greg Michaelson, Deepayan Bhowmik, Paulo Garcia, and Andy Wallace. 2016. RIPL. https://github.com/robstewart57/ripl.

[162] Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions on Embedded Computing Systems (TECS)* 13, 4s (2014), 1–25.

[163] Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand R. Atreya, Martin Odersky, and Kunle Olukotun. 2011. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In *ICML*. 609–616.

[164] Ian Swarbrick, Dinesh Gaitonde, Sagheer Ahmad, Brian Gaide, and Ygal Arbel. 2019. Network-on-chip programmable platform in versal acap architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 212–221.

[165] Synopsys. 2021. RTL Synthesis. https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test.html.

[166] Shinya Takamaeda-Yamazaki. 2015. PyVerilog: A Python-Based hardware design processing toolkit for Verilog HDL. In *Applied Reconfigurable Computing*. Springer, 451–460.

[167] Xifan Tang, Edouard Giacomin, Baudouin Chauviere, Aurelien Alacchi, and Pierre-Emmanuel Gaillardon. 2020. OpenFPGA: An open-source framework for agile prototyping customizable FPGAs. *IEEE Micro* 40, 4 (2020), 41–48.

[168] Russell Tessier, Kenneth Pocek, and Andre DeHon. 2015. Reconfigurable computing architectures. *Proc. IEEE* 103, 3 (2015), 332–354.

[169] The Stanford Pervasive Parallelism Lab. 2017. Spatial: Specify Parameterized Accelerators Through Inordinately Abstract Language. https://github.com/stanford-ppl/spatial.

[170] Lenny Truong and Pat Hanrahan. 2019. A golden age of hardware description languages: Applying programming language techniques to improve design productivity. In *3rd Summit on Advances in Programming Languages (SNAPL 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[171] Yaman Umuroglu, Davide Conficconi, Lahiru Rasnayake, Thomas B Preusser, and Magnus Själander. 2019. Optimizing bit-serial matrix multiplication for reconfigurable computing. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 12, 3 (2019), 1–24.

[172] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. ACM, 65–74.

[173] Universitè Bretagne Sud. 2020. GAUT – High-Level Synthesis Tool from C to RTL. http://hls-labsticc.univ-ubs.fr/.

[174] University of California, Riverside. 2010. ROCCC 2.0. http://roccc.cs.ucr.edu/.
[175] Stylianos I Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. 2018. Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.
[176] Jason Villarreal, Adrian Park, Walid Najjar, and Robert Halstead. 2010. Designing modular hardware accelerators in C with ROCCC 2.0. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*. IEEE, 127–134.
[177] Kizheppatt Vipin and Suhaib A Fahmy. 2018. FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–39.
[178] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. 2017. P4fpga: A rapid prototyping framework for p4. In *Proceedings of the Symposium on SDN Research*. 122–135.
[179] Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA. In *Proceedings of the 2021 ACM/SIGDA international symposium on Field-programmable gate arrays*.
[180] Yutaka Watanabe, Jinpil Lee, Kentaro Sano, Taisuke Boku, and Mitsuhisa Sato. 2020. Design and preliminary evaluation of openacc compiler for fpga with opencl and stream processing dsl. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region Workshops*. 10–16.
[181] Xilinx. 2021. Vitis HLS Front-end is Now Open Source. https://github.com/Xilinx/HLS. Last accessed: July 2nd 2021.
[182] Xilinx Inc. 2013. Vivado Design Suite. http://www.xilinx.com/products/design-tools/vivado.html.
[183] Xilinx Inc. 2013. Vivado HLS. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.
[184] Xilinx Inc. 2018. SDSoC. https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html.
[185] Xilinx Inc. 2019. SDAccel. https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.
[186] Xilinx Inc. 2019. Vitis Unified Software Platform. https://www.xilinx.com/products/design-tools/vitis.html.
[187] Xilinx Inc. 2020. Vitis HLS. https://www.xilinx.com/html_docs/xilinx2020_1/vitis_doc/introductionvitishls.html.
[188] Xilinx Inc. 2021. Vitis Accelerated Libraries. https://github.com/Xilinx/Vitis_Libraries.
[189] Xilinx Inc. 2021. Xilinx Runtime Library. https://github.com/Xilinx/XRT.
[190] Jianxin Xiong, Jeremy Johnson, Robert Johnson, and David Padua. 2001. SPL: A language and compiler for DSP algorithms. *ACM SIGPLAN Notices* 36, 5 (2001), 298–308.
[191] Alberto Zeni, Guido Walter Di Donato, Lorenzo Di Tucci, Marco Rabozzi, and Marco D Santambrogio. 2021. The Importance of Being X-Drop: High Performance Genome Alignment on Reconfigurable Hardware. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 133–141.
[192] Alberto Zeni, Kenneth O'Brien, Michaela Blott, and Marco D Santambrogio. 2021. Optimized implementation of the hpcg benchmark on reconfigurable hardware. In *European Conference on Parallel Processing*. Springer, 616–630.
[193] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. (May 2020).
[194] Wei Zuo, Peng Li, Deming Chen, Louis-Noël Pouchet, Shunan Zhong, and Jason Cong. 2013. Improving polyhedral code generation for high-level synthesis. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE, 1–10.

A CODE EXAMPLES

This Appendix shows three implementations of a *Blur Filter*. The purpose is to highlight how the implementation of the very same computation changes according to the employed language or tool. To this end, we report one solution per design abstraction. In particular, we used SystemVerilog, as it is the foundation of different Hardware Description Languages (HDLs), C/C++ for a Vitis HLS design, and HeteroHalide, as Halide is the entry point of multiple image processing Domain-Specific Languages (DSLs) for Field Programmable Gate Array (FPGA). Please note that this Appendix aims to overview the implementation differences between these abstraction solutions. Thus, the reader shall not consider these designs as fully optimized.

A.1 Blur Filter Background

A Blur Filter [36] is a widely used effect in image processing whose purpose is to reduce image noise and details. This filter smoothens an input image through a $N \times N$ kernel. The following formula defines a Blur kernel:

$$K = \frac{1}{N \cdot N} \cdot \begin{bmatrix} 1 & 1 & ... & 1 \\ 1 & 1 & ... & 1 \\ ... & & & \\ 1 & 1 & ... & 1 \end{bmatrix} \tag{1}$$

For our hardware designs, we chose $N = 3$.

The implementation of this filter can exploit its separable property. Indeed, it is possible to divide this process into two passes: the first one applies a one-dimensional horizontal or vertical kernel on the input image. Then, the second applies the same one-dimension kernel in the remaining dimension. The outcome is equivalent to the usage of a two-dimensional kernel. In the following examples, we implemented a two-dimensional kernel.

Another aspect to consider when applying filters like the Blur one is the image borders. Indeed, we cannot directly use the filter on the image edges as there are not enough pixels to convolve. Consequently, the output image would be smaller than the input one, as shown by the following formulae:

$$H_o = H_i - N + 1 \tag{2}$$

$$W_o = W_i - N + 1 \tag{3}$$

where H_o, W_o, H_i, and W_i are the height and width of output and input images, respectively. There are various ways to handle the image borders in order to produce an output that maintains the same size as the input. For instance, one approach consists of extending the input image boundaries and filling them with values. The padding values may be the mirror of the borders, an extension of the last pixel, or a fixed color. Our implementations of the Blur Filter assume that the host or a previous hardware module has already padded the input image.

A.2 Blur Filter in SystemVerilog

Listing 1 reports a SystemVerilog implementation of the Blur Filter. The core of this design is the `blur_filter` module, which reads and processes one `pixel_t` input pixel per clock cycle. The data type `pixel_t` may refer to a single channel pixel or a three-channel one (e.g., RGB). This module relies on a line buffer `lb` (lines 22 to 30) to store the portion of the input required to apply the filter. On every clock cycle, `lb` extracts specific pixels from its internal buffer and fills the window of pixels `pixel_window` (line 19). We omitted the internal implementation of the line buffer because our focus is on the `blur_filter` module. Nonetheless, a possible design for this component could use the on-chip memory to store the pixels. Please note that, in such a case, the designer has

to either instantiate the memory module explicitly or let the synthesizer infer it from the code. Similarly, we could design `pixel_window` as a set of shift registers. We also omitted the definition of the `init` (line 41) and `blur_kernel` (line 61) functions for the sake of conciseness. The former function initializes the coefficients of the kernel, while the latter applies it.

It is evident from the code that the designer has to handle signals like reset, clock, and ready manually. Moreover, it is up to the designer to decide whether a part of the module is sequential (`always_ff`, lines 32 to 66) or combinational (`always_comb`, not used in this module). Finally, it is worth noting that this module performs the whole computation in one clock cycle. Thus, this approach may cause timing issues due to the long critical path if the designer wants to target a high clock frequency. In such a case, the designer has to split the critical path and build a pipelined design manually.

```
1  module blur_filter #(
2      parameter WIDTH = INPUT_WIDTH,
3      parameter HEIGHT = INPUT_HEIGHT,
4      parameter PADDING = IMAGE_PADDING,
5      parameter KERNEL_SIZE = 3) (
6      input logic clk,
7      input logic reset,
8      input pixel_t data_in,
9      input logic data_in_ready,
10     output pixel_t data_out,
11     output logic data_out_ready);
12
13     int unsigned row;
14     int unsigned col;
15
16     pixel_t curr_pixel;
17     logic curr_pixel_ready;
18     pixel_t blur_coeffs [KERNEL_SIZE * KERNEL_SIZE];
19     pixel_t pixel_window [KERNEL_SIZE * KERNEL_SIZE];
20     logic pixel_window_ready;
21
22     line_buffer #(.WIDTH(WIDTH), .HEIGHT(HEIGHT), .PADDING(PADDING),
23         .KERNEL_SIZE(KERNEL_SIZE)) lb(
24         .clk(clk),
25         .reset(reset),
26         .data_in(curr_pixel),
27         .data_in_ready(curr_pixel_ready),
28         .data_out(pixel_window),
29         .data_out_ready(pixel_window_ready)
30     );
31
32     always_ff @(posedge clk) begin
33         curr_pixel <= 0;
34         curr_pixel_ready <= 0;
35         data_out <= 0;
36         data_out_ready <= 0;
37         row <= row;
38         col <= col;
39
40         if(reset == 1) begin
```

```
41             init(blur_coeffs);
42             row <= 0;
43             col <= 0;
44         end
45         else begin
46             if (data_in_ready == 1) begin
47                 curr_pixel <= data_in;
48                 curr_pixel_ready <= 1;
49             end
50             if(pixel_window_ready == 1) begin
51                 if(col == (INPUT_WIDTH - 1)) begin
52                     col <= 0;
53                     row <= row + 1;
54                 end
55                 else begin
56                     col <= col + 1;
57                 end
58 
59                 if (row >= (KERNEL_SIZE - 1) && row < INPUT_HEIGHT &&
60                     col < INPUT_WIDTH && col >= (KERNEL_SIZE - 1)) begin
61                     data_out <= blur_kernel(pixel_window, blur_coeffs);
62                     data_out_ready <= 1;
63                 end
64             end
65         end
66     end
67 endmodule
```

Listing 1. SystemVerilog Blur Filter

A.3 Blur Filter in Vitis HLS

Listing 2 contains a C/C++ implementation of the Blur Filter for Vitis HLS. At first glance, the reader can note that this design has some relevant differences compared to the SystemVerilog one. The first one is the absence of signals such as clock and reset. Indeed, this algorithmic/functional specification of the filter is untimed and does not require the management of low-level signals; the tool will automatically take of this aspect during the High-Level Synthesis (HLS) process. Another difference is the usage of directives/pragmas (lines 3, 5, and 12) to guide the optimization process. In particular, for this design, we explicitly report the implementation of the line buffer `lb` to demonstrate how the designer can easily design such a component using pragmas. We modeled `lb` as a bidimensional array partitioned per row. Similarly, we completely partitioned the window of pixels `pixel_window`. The result is that the Vitis HLS allocates the rows of lb on different on-chip memories and the elements of `pixel_window` in registers.

The pragma `PIPELINE` (line 12) enforces the design of a pipeline for the loop `L1` (line 11). Actually, Vitis HLS automatically flattens this loop and the previous one `L0` (lines 10 and 11), resulting in a pipeline covering the whole computation of the `blur_filter` function. Moreover, Vitis HLS chooses the proper pipeline depth, targets an Initiation Interval (II) of 1, if possible, and automatically unrolls the loops inside the pipelined one. Finally, just like in the SystemVerilog design, we do not report the code of the `blur_kernel` function for the sake of conciseness.

As the last aspect, it is worth noting that we used two `pixel_t` pointers for the input and output images, respectively. These pointers may refer to on-chip memory buffers or off-chip ones. In

the former case, Vitis HLS may implement them as First-In First-Outs (FIFOs), especially if the designers modeled the overall design as a dataflow one. Alternatively, the designer can use the Vitis HLS built-in templated class `hls::stream<>` to define these buffers as FIFOs explicitly. In this case, the designer can also select the physical resource for the FIFOs (shift registers or on-chip memories) or let the tool do that.

```
void blur_filter(pixel_t* input_img, pixel_t* output_img){
  pixel_t lb[KERNEL_SIZE - 1][INPUT_WIDTH];
#pragma HLS ARRAY_PARTITION variable=lb dim=1 complete
  pixel_t pixel_window[KERNEL_SIZE][KERNEL_SIZE];
#pragma HLS ARRAY_PARTITION variable=pixel_window dim=0 complete

  unsigned int input_idx = 0;
  unsigned int output_idx = 0;

  L0:for(unsigned int i = 0; i < (INPUT_HEIGHT); i++){
    L1:for(unsigned int j = 0; j < (INPUT_WIDTH); j++){
#pragma HLS PIPELINE
      pixel_t current_pixel = input_img[input_idx++];

      L2:for(unsigned int k0 = 0; k0 < KERNEL_SIZE; k0++){
        L3:for(unsigned int k1 = 0; k1 < KERNEL_SIZE - 1; k1++){
          pixel_window[k0][k1] = pixel_window[k0][k1 + 1];
        }
      }

      L4:for(unsigned int k0 = 0; k0 < KERNEL_SIZE - 1; k0++){
        pixel_window[k0][KERNEL_SIZE - 1] = lb[k0][j];
      }

      L5:for(unsigned int k0 = 0; k0 < KERNEL_SIZE - 2; k0++){
        lb[k0][j] = lb[k0 + 1][j];
      }

      lb[KERNEL_SIZE - 2][j] = current_pixel;
      pixel_window[KERNEL_SIZE - 1][KERNEL_SIZE - 1] = current_pixel;

      if (i >= (KERNEL_SIZE - 1) && j >= (KERNEL_SIZE - 1)){
        output_img[output_idx++] = blur_kernel(pixel_window);
      }
    }
  }
}
```

Listing 2. C/C++ HLS Blur Filter

A.4 Blur Filter in HeteroHalide

Listing 3 shows a HeteroHalide design of the Blur Filter, which we adapted from the HeteroHalide repository [33]. For this code example, we report the allocation of Halide buffers and variables (lines 1 to 3), Blur Filter computation (lines 5 to 13), scheduling commands (lines 15 and 16), and lowering to HeteroCL code (lines 18 to 24). It is interesting to note how the designer can set up the whole hardware design in a few lines of code, unlike SystemVerilog and HLS, which may also require

additional code to manage the communication with the off-chip memory or other modules. Indeed, the Halide paradigm offers a concise definition of the computation and enforcement of optimizations through scheduling commands. In this case, the code contains the `compute_root` command only. According to Halide documentation, this implies that a `Func` (e.g., `blur_filter`) is computed once ahead of time, producing enough results to satisfy all its uses [19]. Nonetheless, please note that the designer may exploit various other scheduling commands to optimize the resultant design further. Finally, the output of this code is a HeteroHalide implementation (`blur_filter.py`) that its infrastructure processes and optimizes to produce a design for the supported backends.

```
Buffer<pixel_t> input_img(INPUT_WIDTH, INPUT_HEIGHT, CHANNELS);
Buffer<pixel_t> output_img(OUTPUT_WIDTH, OUTPUT_HEIGHT, CHANNELS);
Var x("x"), y("y"), c("c");

Func blur_filter("blur_filter");
blur_filter(x, y, c) =
    (input_img(x, y, c) + input_img(x+1, y, c) + input_img(x+2, y, c) +
    (input_img(x, y+1, c) + input_img(x+1, y+1, c) + input_img(x+2, y+1, c) +
    (input_img(x, y+2, c) + input_img(x+1, y+2, c) + input_img(x+2, y+2, c)) /
    (KERNEL_SIZE * KERNEL_SIZE);

Func final("final");
final(x, y, c) = blur_filter(x, y, c);

blur_filter.compute_root();
final.compute_root();

std::vector<int> output_img_shape;
for (int i = 0; i < output_img.dimensions(); i++){
    output_img_shape.push_back(output_img.extent(i));
}

final.compile_to_heterocl("blur_filter.py", {input_img}, output_img_shape,
    "final");
```

Listing 3. HeteroHalide Blur Filter

A.5 Summary

This Appendix gave an overview of the main high-level differences between the design abstractions discussed in this survey. First, HDLs offer constructs to model low-level hardware designs. In this scenario, the designer has complete control over the resulting solution, managing signals (e.g., clock and reset) and implementing combinational and sequential circuits. Increasing the abstraction level, the current generation of HLS tools enables the designer to define a hardware component through the constructs of an untimed algorithmic specification in languages like C/C++, taking care of the production of the corresponding HDL code. Besides, the designer has access to multiple directives/pragmas to handle and optimize aspects of the resulting solution (e.g., off-chip memory interfaces and pipelining). Finally, domain specialization incarnates the highest abstraction level for FPGA development so far. Indeed, DSLs shift most of the design burden from the designer to the compiler, which exploits the domain specialization to enforce multiple transformations and optimizations automatically. Besides, in the case of DSLs like Halide, the designer may also leverage scheduling commands to suggest additional optimizations. Thanks to these features, the designer only concentrates on the algorithmic specification.

B TECHNICAL TERMS DEFINITION

This Appendix defines the technical terms reported throughout the survey. In this way, the reader can easily understand the purpose of unfamiliar features employed by the analyzed tools/languages.

Table 4. Index and description of technical terms employed in the survey.

Technical Term	*Description*	*Reference*
Abstraction layer	An approach to mask the details of an underlying layer, level, or subsystem and decouple the internal aspects to ease interoperability.	Section 2.1, 2.4, 4.1, 4.4, Table 3
Algorithmic skeletons	RIPL design primitives that are combinable in multiple ways to form image processing kernels with common data access patterns.	Section 4.1, Table 3
ASIC support	The design flow is also capable of targeting Application Specific Integrated Circuits (ASICs).	Table 2, 3
Automatic integration	The feature of generating host APIs and runtime for CPU-FPGA communication and management.	Table 3
Automatic scheduling	The feature of scheduling the computations automatically according to their architectural pattern.	Section 4.1, Table 3
Bitstream generation	The feature of a design flow to automatically go from a given input design down to the bitstream.	Section 3, 3.2, Table 2
Bitwidth analysis and optimization	The process of determining the required data container (e.g., 1, 3, 33, or 77 bits) for the target design [4, 10, 14, 15, 28, 31].	Section 3.1
BlackBox IP	The feature of integrating custom Intellectual Propertys (IPs) (mainly VHDL and (System)Verilog) in the target design flow.	Section 2.4, Table 1
Clock-gating	Low-power optimization that removes the clock signal when the circuit is not used to reduce dynamic power dissipation. It saves power by pruning the clock tree at the cost of adding more logic and negligible leak currents.	Section 3.1, Table 2
Clock-less	DFiantHDL feature exposing an RTL experience where the designer does not care about clock considerations, similarly to the untimed description of HLS designs.	Section 2.1, Table 1
Cross domain crossing	In synchronous digital designs, the traversal of a signal from one clock domain region into another.	Section 2.1, 3, 3.1
Cross-probing/cross-linking support	Graphical editors' features of linking/modifying the specific graphical view and the associated source code.	Section 3.1
Dataflow	A computational model describing an architecture where data seamlessly flow through different concurrent modules as soon as the operators are ready. Thus, this model does not require a traditional control path based on a program counter.	Section 2.1, 2.2, 3.1, 3.2, 4.1, 4.3, Table 1, Appendix A.3
DDR design	Usually, a sequential design activates only on the negative or positive edge of the clock, not both. Conversely, a Double Data Rate (DDR) design is active on both the clock edges.	Section 2, 2.3, 3, Table 1
DSE engine/tool	A component that automatically performs the Design Space Exploration (DSE) to support the optimization process.	Section 3.1, 4.2, 4.3, Table 3
Dynamic memory allocation/dynamic execution scheduling	The feature of dynamically scheduling some memory space or functionalities (i.e., at runtime). Usually, HLS tools prevent the usage of such dynamic constructs (e.g., `malloc`).	Section 3.1, 3.2, 3.3, 4.3, Table 3, Appendix D

Technical Term	*Description*	*Reference*
Dynamically scheduled circuits	Almost every HLS tool schedules the operations statically (i.e., at compile time) and, if possible, reserves more resources than necessary at runtime. Conversely, dynamic scheduling means adapting this execution plan dynamically (i.e., at runtime). Dynamatic website has a good example that showcases this behavior, and we point the interested readers to it [17].	Section 3.1, Table 2
External IP integration	The feature of integrating into the design flow external IPs packed in some formats, e.g., XACT.	Section 2.1
Functional verification	The act of verifying that the considered design conforms to specification.	Section 3.1, 3.3, Table 1
Fuzzer	An automated tool to create random input data (e.g., code from a language grammar, in the case of XLS) for verification purposes.	Section 3.1, Table 2
Generators	A kind of generalized hardware design that mixes some meta-programming and parametrization to automatically combine several modules for a more complex system like a System-on-Chip (SoC) [3, 5, 18, 37].	Section 2, 2.1, 2.3
Graphical design	The design flow supports a visual approach other than coding to model the target solution.	Table 2
Hierarchical synthesis	An optimization that generalizes pipelining, allowing various functions to run in a parallel and pipelined manner [7].	Section 3.1
HLS binding	The HLS step of binding specific operations to given functional units [9, 12, 13, 26].	Section 3, 3.1
HLS resource allocation	The HLS step of allocating specific resources for given operations [9, 12, 13, 26].	Section 3, 3.1
Host code design	The design of the code running on the host system in charge of interacting with the FPGA.	Section 3.1, 3.2, 4.2, 4.3, Table 1, Table 3
If-conversion	A widely known software optimization [22] where all the branches are executed in parallel with a simple guard deciding which result should be committed [26].	Appendix D
Initiation interval	The number of clock cycles after which a pipelined functionality/loop can begin a new iteration of the function/loop.	Appendix A.3
Instrumentation	PyRTL feature of modifying the source code directly to measure performance or log significant events.	Section 2.2, Table 1
Intellectual Property (IP)	Reusable design blocks either internally developed or third-party.	Section 1, 2, 2.1, 2.4, 3, 3.1, 3.2, 3.3, 4.2, 5, Table 1
Lazy transformations	HeteroHalide feature of not implementing directly a given transformation in the Halide Intermediate Representation (IR) but explicitly lowering it to HeteroCL.	Section 4.1, Table 3
Level specification	ExaSlang 4 feature of defining objects/functionalities for a given multigrid level that override the default ones [29].	Section 4.1, Table 3
Loop hoisting	Loop hoisting is a traditional compiler-based optimization (also called loop-invariant code motion or scalar promotion). It mainly consists of moving statements or expressions outside the body of a loop without affecting the semantics of the program‡ [2].	Appendix D

‡https://compileroptimizations.com/category/hoisting.html

Technical Term	*Description*	*Reference*
Low-power optimizations	A collection of techniques and methodologies to reduce a digital design's dynamic and static power consumption [27].	Section 3.1
Multiple architectural backends	HeteroCL feature of translating a DSL program into different compatible backends.	Section 4.1, 4.3
Multi-clock design	A design with multiple clock domains, e.g., a design that must preserve a specific frequency for the I/O (such as PCIe) and another for internal processing.	Section 2.1, Table 1, 2
Non-linear pipeline	PolyMage feature of dealing and supporting DAG-based pipelines, where decoupled parallel stages execute without serialization [11].	Appendix D
Non-synthesizable	HDL abstractions that are not automatically convertible from RTL to logic gates through the logic synthesis process, hence, mainly devoted to functional/timing verification/simulation [23].	Section 2
Operation chaining	An optimization for scheduling multiple combinational operators together in a single clock cycle to avoid false paths [26, 32].	Section 3.1, Appendix D
Operation-level parallelism	Hastlayer parallelization of multiple operations (simpler than a single task such as addition, multiplication), or SIMD-like parallelization [21].	Section 3.1
Parallelism levels	PolyMage feature of leveraging different parallelism levels in the image processing domain, e.g., inter-state, channel-level, data-level parallelism levels [11].	Appendix D
Parametrization	The feature of determining some module parameters that define the internal structure. For instance, a Flip-Flop may be parametrized to store different compile-time data widths, enabling the reuse of the same FF module across different scenarios. This approach prevents rewriting the same module for any possible data width.	Section 2, 2.1, 2.2, 2.3, Table 1
Partial evaluation	AnyHLS feature of optimizing algorithm variants at compile-time [16].	Section 4.3, Table 3
Partial reconfiguration	The feature of reconfiguring just a portion of the FPGA, leaving the rest of the current configuration untouched [35].	Section 3.2, Table 1
Polymorphism	The feature of provisioning a single interface to entities of different types [6] or the usage of a symbol to represent multiple different types [8].	Section 1, 2, 2.1, 2.2, 2.3, 2.4, Table 1
Pthread/OpenMP support	The unconventional feature of HDLs and HLS toolchains to manage dynamic thread constructs and translate them into hardware modules.	Section 2.2, 3.1, 3.2, Table 2
Recursion	A standard feature of programming languages usually not supported in HDL/HLS/DSL flows since it involves dynamic memory allocation.	Section 2.1, 2.3, 3.1, 3.3, Table 1, Appendix D
Recursion	The unusual feature of supporting standard recursion by the design abstraction since it involves dynamic allocations.	Section 2, 3.1, 3.3, Appendix D
Reset polarity	The design abstraction feature of activating the reset on a negative or positive edge or being active on logic high or logic low.	Section 2.1, Table 1
Scala-FPGA runtime	The VeriScala software that handles the FPGA-CPU communication.	Table 1

Technical Term	*Description*	*Reference*
Shallow embedding	AnyHLS feature of supporting additional domain-specific structures without affecting the compiler [20].	Section 4.3, Table 3
Smart buffers	On-chip buffers enabling data reuse through loop iterations according to the data access pattern.	Section 3.1, Table 2
System integration	The feature of emulating/building a complete system and not just the IP per se.	Table 2
Task-level parallelism	The parallelism of having multiple tasks executed concurrently among the others.	Section 3.1
Timing verification	Verifying that a given design respects some timing constraints.	Section 2.1, 2.3, Table 1
Tool for quality coverage metrics	Catapult HLS feature of reporting coverage metrics similar to a software one. We report Siemens website details: "Use traditional RTL metrics such as statement, branch, expression, and toggle coverage, combined with functional verification techniques from SystemVerilog to reach high-quality HLS-aware coverage without slow and expensive RTL Simulation."†	Section 3.1

† https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/hls-verification/coverage/

C QUALITATIVE METRIC DEFINITION

This Appendix describes the qualitative metrics we employed in Section 5 to compare HDLs, HLS tools, and DSLs through Figure 7.

Table 5. Qualitative metrics of Figure 7

Metric	*Description*
Design productivity	The degree of features the language/tool offers to build a hardware design.
Flexibility	The capacity of implementing whatever hardware design with that language/tool.
Conciseness	The number of lines of code required to build a hardware design.
Performance	The efficiency of a hardware design in terms of metrics such as latency and throughput.
Resource efficiency	The number of resources required to implement a hardware design given a resource budget.
Verification productivity	The degree of features the language/tool offers to verify a hardware design.

D REMOVED TOOLS AND LANGUAGES

This Appendix reports tools and languages we excluded from the survey due to space limitations.

D.1 High-Level Synthesis Tools

Kiwi: Kiwi [30, 34] is an open-source HLS toolchain developed by the University of Cambridge and Microsoft Research and currently maintained by the former. Kiwi takes C# code, compiles it to .NET bytecode, and then generates Verilog code for both Xilinx and Intel FPGAs. Kiwi supports a broad subset of high-level features. For instance, the designer can leverage C# concurrency constructs (e.g., threads, events, and monitors) to map parallel code to hardware efficiently. Moreover, unlike most HLS tools, Kiwi supports recursion (with some restrictions), dynamic memory allocation, and pointer manipulation. Internally, the Kiwi Compiler (KiwiC) relies on the Value State Flow Graph (VSFG) technique to compile heavy control flow code. Thanks to this technique, KiwiC can apply optimizations like dynamic execution scheduling, speculation, and loop transformations.

DWARV: DWARV [25] builds upon CoSy, an HLS compiler developed by ACE [1]. Therefore, its characteristics are directly related to the features of CoSy: modular and robust backend, easiness of extension with new optimizations. Besides, DWARV provides a flexible way to exploit standard and custom optimizations, such as basic loop optimizations, loop hoisting, scheduling optimizations, if-conversion, operation chaining, bit-width analysis, and more. Another feature of DWARV is a general template that permits the description and integration of IP blocks from external libraries as custom function calls. DWARV accepts general C code and outputs VHDL. Besides, it applies no restrictions on the application domain and can generate hardware for both streaming and control-intensive applications. Over the years, developers extended the subset of supported features, including pointers, memory accesses, and integer and floating-point data types. On the other hand, DWARV does not support global variables, recursion, or the standard C mathematical library.

D.2 Application Domain DSLs

PolyMage: PolyMage [24] is a DSL and compiler implementing optimized image processing pipelines for both CPUs and FPGAs [11]. PolyMage supplies various operators (e.g., point-wise, stencil, and up/downsampling) to specify the filters and supports both linear and non-linear pipelines. The FPGA backend processes the functional input code and derives a directed acyclic graph exposing producer/consumer relationships. The backend connects each producer pipeline stage, which corresponds to a graph node, to the consumers with FIFOs, which correspond to the graph edges. According to the filter performed in the consumer stage, the backend determines the proper FIFO sizes. Furthermore, PolyMage leverages different parallelism forms when implementing the hardware pipelines, from inter-stage to channel-level, maximizing the memory bandwidth usage. Finally, PolyMage compiles the input pipeline into C/C++ code for Vivado HLS.

APPENDIX REFERENCES

[Appendix1] ACE. 2017. CoSy compiler development system. http://www.ace.nl/compiler/cosy.

[Appendix2] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. Compilers, principles, techniques. *Addison wesley* 7, 8 (1986), 9.

[Appendix3] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* 40, 4 (2020), 10–21. https://doi.org/10.1109/MM.2020.2996616

[Appendix4] Jonathan Babb, Martin Rinard, Csaba Andras Moritz, Walter Lee, Matthew Frank, Rajeev Barua, and Saman Amarasinghe. 1999. Parallelizing applications into silicon. In *Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines (Cat. No. PR00375)*. IEEE, 70–80.

[Appendix5] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, et al. 2016. OpenPiton: An open source manycore research framework. *ACM SIGPLAN Notices* 51, 4 (2016), 217–232.

[Appendix6] Bjarne Stroustrup. 2007. Bjarne Stroustrup's C++ Glossary. https://www.stroustrup.com/glossary.html#Gpolymorphism.

[Appendix7] Thomas Bollaert. 2008. Catapult synthesis: a practical introduction to interactive C synthesis. In *High-Level Synthesis*. Springer, 29–52.

[Appendix8] Luca Cardelli and Peter Wegner. 1985. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)* 17, 4 (1985), 471–523.

[Appendix9] Joao MP Cardoso, Pedro C Diniz, and Markus Weinhardt. 2010. Compiling for reconfigurable computing: A survey. *ACM Computing Surveys (CSUR)* 42, 4 (2010), 1–65.

[Appendix10] Stefano Cherubin and Giovanni Agosta. 2020. Tools for reduced precision computation: A survey. *ACM Computing Surveys (CSUR)* 53, 2 (2020), 1–35.

[Appendix11] Nitin Chugh, Vinay Vasista, Suresh Purini, and Uday Bondhugula. 2016. A DSL compiler for accelerating image processing pipelines on FPGAs. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. 327–338.

[Appendix12] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. 2011. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (2011), 473–491.

[Appendix13] Giovanni De Micheli. 1994. *Synthesis and optimization of digital circuits*. Number BOOK. McGraw Hill.

[Appendix14] Yang Ding and Weng Fai Wong. 2005. Bit-width analysis for general applications. (2005).

[Appendix15] Douglas do Couto Teixeira and Fernando Magno Quintao Pereira. 2011. The design and implementation of a non-iterative range analysis algorithm on a production compiler. *SBLP. SBC* (2011), 45–59.

[Appendix16] Yoshihiko Futamura. 1983. Partial computation of programs. In *RIMS Symposia on Software Science and Engineering*. Springer, 1–35.

[Appendix17] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. 2019. DYNAMATIC - From C/C++ to Dynamically-Scheduled Circuits. https://dynamatic.epfl.ch

[Appendix18] Andreas Kurth, Pirmin Vogel, Alessandro Capotondi, Andrea Marongiu, and Luca Benini. 2017. HERO: Heterogeneous embedded research platform for exploring RISC-V manycore accelerators on FPGA. *arXiv preprint arXiv:1712.06497* (2017).

[Appendix19] Halide Language. 2013. Documentation. https://halide-lang.org/docs/class_halide_1_1_func.html.

[Appendix20] Roland Leißa, Klaas Boesche, Sebastian Hack, Richard Membarth, and Philipp Slusallek. 2015. Shallow embedding of DSLs via online partial evaluation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. Association for Computing Machinery, New York, NY, USA, 11–20.

[Appendix21] Lombiq Technologies. 2019. Hastlayer SDK - GitHub. https://github.com/Lombiq/Hastlayer-SDK.

[Appendix22] Scott A Mahlke, Richard E Hank, Roger A Bringmann, John C Gyllenhaal, David M Gallagher, and Wen-mei W Hwu. 1994. Characterizing the impact of predicated execution on branch prediction. In *Proceedings of the 27th annual international symposium on Microarchitecture*. 217–227.

[Appendix23] Clive Maxfield. 2004. *The design warrior's guide to FPGAs: devices, tools and flows*. Elsevier.

[Appendix24] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic optimization for image processing pipelines. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 429–443.

[Appendix25] Razvan Nane, Vlad-Mihai Sima, Bryan Olivier, Roel Meeuws, Yana Yankova, and Koen Bertels. 2012. DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*. IEEE, 619–622.

[Appendix26] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, et al. 2016. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 10 (2016), 1591–1604.

[Appendix27] Preeti Ranjan Panda, BVN Silpa, Aviral Shrivastava, and Krishnaiah Gummidipudi. 2010. *Power-efficient system design.* Springer Science & Business Media.

[Appendix28] Jason RC Patterson. 1995. Accurate static branch prediction by value range propagation. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation.* 67–78.

[Appendix29] Christian Schmitt, Sebastian Kuckuk, Frank Hannig, Harald Köstler, and Jürgen Teich. 2014. ExaSlang: A domain-specific language for highly scalable multigrid solvers. In *2014 Fourth international workshop on domain-specific languages and high-level frameworks for high performance computing*. IEEE, 42–51.

[Appendix30] Satnam Singh and David J Greaves. 2008. Kiwi: Synthesis of FPGA circuits from parallel programs. In *2008 16th International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 3–12.

[Appendix31] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. 2000. Bidwidth analysis with application to silicon compilation. *ACM SIGPLAN Notices* 35, 5 (2000), 108–120.

[Appendix32] Leon Stok. 1994. Data path synthesis. *Integration* 18, 1 (1994), 1–71.

[Appendix33] UCLA VAST Lab. 2020. HeteroHalide: From Image Processing DSL to Efficient FPGA Acceleration. https://github.com/UCLA-VAST/heterohalide.

[Appendix34] University of Cambridge. 2016. Kiwi Scientific Acceleration using FPGA. https://www.cl.cam.ac.uk/~djg11/kiwi/.

[Appendix35] Kizheppatt Vipin and Suhaib A Fahmy. 2018. FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–39.

[Appendix36] OpenCV: Open Source Computer Vision. 2000. Image Filtering. https://docs.opencv.org/3.4/d4/d86/group__imgproc__filter.html. last accessed 6 February 2022.

[Appendix37] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. (May 2020).