IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received August 18, 2020, accepted August 31, 2020, date of publication September 14, 2020, date of current version October 5, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3024098

Towards Automatic High-Level Code Deployment on Reconfigurable Platforms: A Survey of High-Level Synthesis Tools and Toolchains

MOSTAFA W. NUMAN[1], BRADEN J. PHILLIPS[2], (Member, IEEE), GAVIN S. PUDDY[1], (Associate Member, IEEE), AND KATRINA FALKNER[1]

[1]School of Computer Science, The University of Adelaide, Adelaide, SA 5005, Australia
[2]School of Electrical and Electronic Engineering, The University of Adelaide, Adelaide, SA 5005, Australia

Corresponding author: Mostafa W. Numan (mostafa.numan@adelaide.edu.au)

This work was supported by the Maritime Division of the Defence Science and Technology Group, Australia.

ABSTRACT Heterogeneous computing systems with tightly coupled processors and reconfigurable logic blocks provide great scope to improve software performance by executing each section of code on the processor or custom hardware accelerator that best matches its requirements and the system optimisation goals. This article is motivated by the idea of a software tool that can automatically accomplish the task of deploying code, originally written for a conventional computer, to the processors and reconfigurable logic blocks in a heterogeneous system. We undertake an extensive survey of high-level synthesis tools to determine how close we are to this vision, and to identify any capability gaps. The survey is structured according to a new framework that clearly expresses the relationships between the many tools surveyed. We find that none of the existing tools can deploy general high-level code without manual intervention. Logic synthesis from arbitrary high-level code remains an open problem with dynamic data structures, function pointers and recursion all presenting challenges. Other challenges include automating the tasks of code partitioning, optimisation and design space exploration.

INDEX TERMS Automatic code deployment, field programmable gate arrays, high level synthesis, heterogeneous platforms.

I. INTRODUCTION

For the last four decades, Moore's law and Dennard scaling have relentlessly delivered improvements in computing performance [1]. Since the early 2000s their impact has begun to wane and alternative ways to improve performance have begun to emerge. Heterogeneous computing is a promising approach in which a group of processing nodes execute a workload in parallel. Given different kinds of nodes including multi-core CPUs, real-time processors, DSPs, GPUs, and accelerators on FPGAs or ASICs, the computing workload can be partitioned such that each part is executed on a processor that is well-matched to its requirements and the performance optimisation goals.

This article is concerned with the engineering task of writing software for a heterogeneous system and considers how close existing tools and technologies are to a fully automatic system in which high-level source code is partitioned and deployed to heterogeneous nodes with a minimum of human intervention. This is an ambitious scope so we constrain ourselves, in this article, to the task of deploying source code blocks onto custom FPGA logic.

It is possible, of course, to write software specifically for a particular heterogeneous system by manually partitioning tasks among the processors, and using the most appropriate programming language for each of the different processors. For example a Hardware Description Language (HDL) such as Verilog could be used for tasks executing on an FPGA, and CUDA for those on a GPU. An alternative, which has seen a great deal of research activity in recent years, is to use *High-Level Synthesis* (HLS) for generating hardware modules from code written in a High-Level Language (HLL) (such as C, C++ or Python).

There are benefits of using HLS instead of HDL so that the entire application is in a high-level language: simulation speed is generally faster; debugging is less difficult; it is

The associate editor coordinating the review of this manuscript and approving it for publication was Christian Pilato.

easier to explore and evaluate design alternatives; and the high-level language may include features that cannot be easily expressed in a HDL [2].

Although current HLS tools do not always produce performance-optimised implementations, applications without stringent performance requirements can be more quickly and easily developed using HLS. HLS software developers do not necessarily need to be FPGA or HDL experts, and optimisation opportunities can be exposed to the designer that cannot be easily explored via HDL approaches. In some cases, a project that would not have been practical in HDL, given its complexity, limited time frame and small development team, can be feasible in HLS at a low performance cost compared to an HDL-based approach [3]–[5].

An HLS-based design for a heterogeneous system could be started from scratch, or use pre-existing code originally written for a conventional CPU. Either way, to effectively use current HLS technology, system developers require considerable knowledge and experience in the application domain, computer programming, and HLS design flow.

Deploying pre-existing code written for a conventional CPU onto a heterogeneous system with the aim of improving performance or efficiency is even more difficult. The code needs to be substantially restructured to be synthesisable, and to produce optimised hardware. This needs to be done for all the code: not just the application source code but also any library functions it uses. To date *Automatic Code Deployment* (ACD) tools capable of performing this challenging task without human intervention have been the subject of limited research (e.g. [6], [7]) but a considerable amount of work has been done on automating some of the more challenging, tedious and time-consuming steps in this process (e.g. [8]–[11]).

This article surveys recent toolchains and workflows for high-level synthesis to FPGA with a focus on technologies that might eventually be used for automatic code deployment. Section I introduces HLS and the motivation for heterogeneous computing based on HLS. Section II categorises different contemporary approaches to deploy compute-intensive code segments to FPGA hardware accelerators. The categories introduced in Section II are used to organise a thorough survey, in Sections III and IV, of approaches that take a candidate function expressed in a HLL and produce low-level HDL suitable for FPGA deployment. The arguments in these sections focus on contemporary HLS tools currently used in academia or industry; legacy tools are included in summary tables for completeness. Specification of a hypothetical tool for ACD to FPGA, as well as a brief summary of progress reported in the literature towards making HLS-based FPGA code deployment less dependent on human judgement and proficiency, are provided in Section V.

II. HIGH-LEVEL CODE DEPLOYMENT APPROACHES

To build an FPGA-based heterogeneous system, a common practice is to begin with an application running on a CPU, which is profiled to identify functions (or code blocks) to be offloaded to the FPGAs. Careful choice of the offloaded code blocks can improve latency, throughput or energy efficiency of the application as a whole. One part of the application, which controls and synchronises the execution, usually remains on a *host* CPU; and compute-intensive parts are synthesised as *accelerators* on the FPGAs. In the past, hardware developers had to manually rewrite the compute-intensive HLL functions as custom accelerators at the Register Transfer Level of abstraction (RTL), using a Hardware Description Language (HDL) (e.g. VHDL or Verilog). Hardware synthesis for FPGAs from a manually written HDL description is a well-understood process in the literature, and is beyond the scope of this article. However, we have illustrated this method with the label ① in FIGURE 1, to indicate that it is still one possible approach.

The manual transformation of a HLL function to RTL has some fundamental limitations:

- Understanding the HLL code and then manually writing the HDL code for the accelerator, requires repeated design effort that reduces productivity and increases time-to-market of the end product.
- Notions of timing at RTL require the developer to have a good understanding of the hardware platform. It is hard to find experts with the necessary blend of software and hardware expertise. For a hardware developer, understanding the HLL code can be as problematic as understanding hardware optimisation for a software engineer. This gap often results in an inefficient hardware implementation.
- High-level constructs that lead to an efficient implementation on a CPU can be problematic in an FPGA. For example, floating point operations, dynamic data structures, functions called by reference, and recursion can all be difficult to map to efficient hardware.
- Even small changes to the software application to fix bugs or add new features can require significant modification of the HDL code and extensive restructuring of the hardware.
- The task becomes even more challenging if there is a legacy software code base, such as in runtime libraries, that needs to be deployed on the FPGA.
- Parallelism is usually key to improved performance on FPGA but it is difficult to extract parallelism from code that was originally optimised for sequential CPUs.

Researchers have been working towards defining hardware accelerators at higher levels of abstraction using HLLs, and building tools to automatically transform HLL code to an RTL description. Common practice at present is to write the compute-intensive parts of the code as functions that use only a subset of the HLL such that that they are *synthesisable* for the FPGA. C and C-based HLLs are most common because of their wide-acceptance among both software and hardware engineers.

This design flow is shown as ② in FIGURE 1, and will be discussed in Section III. It can be divided into subcategories:

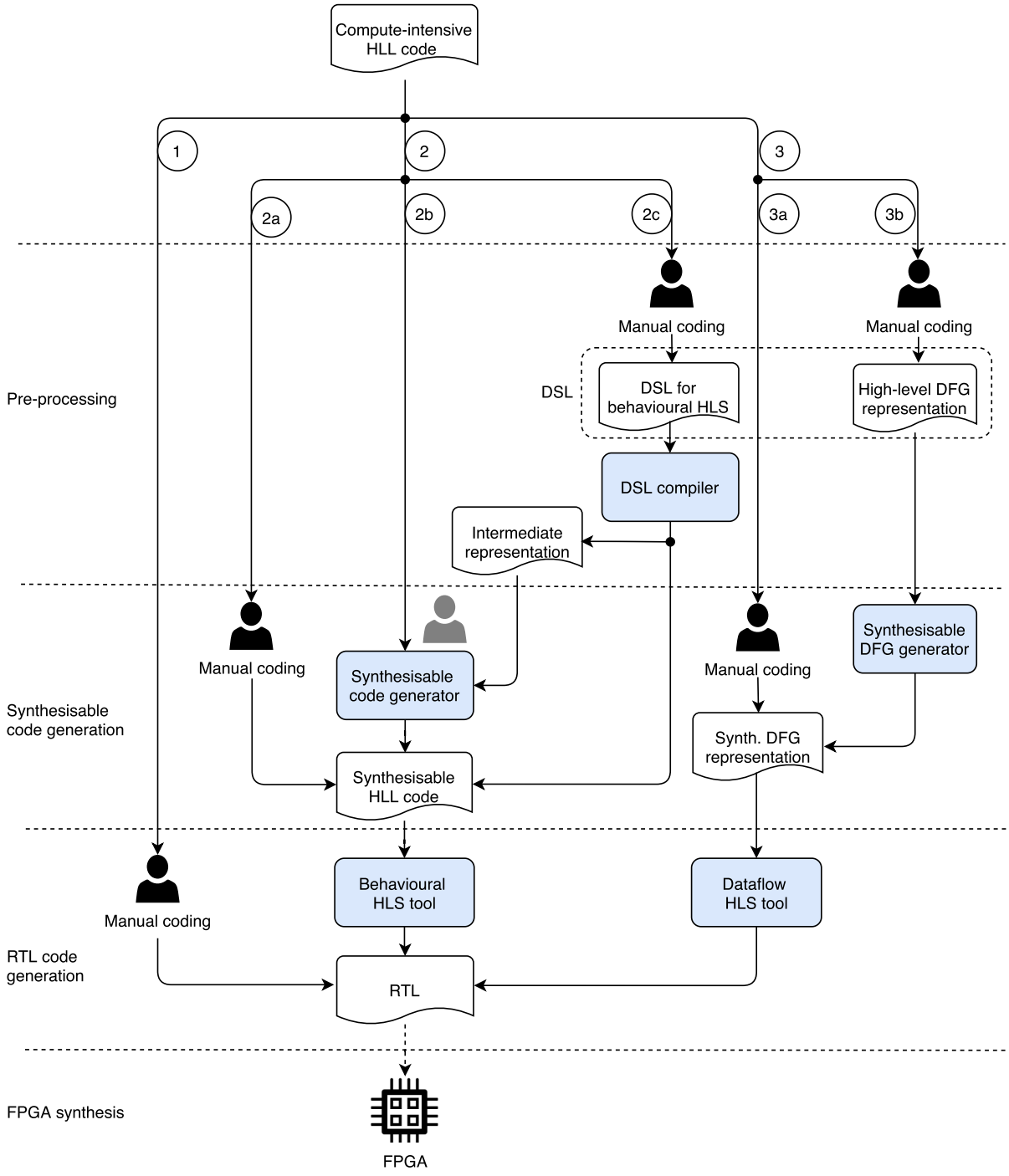


FIGURE 1. Design flows for high-level code deployment.

- *Behavioural High-Level Synthesis (HLS)* (2a): In this approach, the hardware developer selects compute-intensive functions and rewrites them to synthesisable HLL. The behavioural HLS tool then converts the functions into RTL descriptions for FPGA implementation. Tools to support this flow are surveyed in Section III-A.
- *Synthesisable behavioural code generation* (2b): There are some tools to generate synthesisable code (sometimes just called 'HLS code') from HLL code. They are usually 'semi-automatic' tools that require some manual code refactoring and annotation. The generated synthesisable code can be passed to HLS tools to generate RTL descriptions. This approach is discussed in Section III-B.
- *Domain-Specific Language (DSL) for HLS* (2c): In this approach, compute-intensive HLL code is manually re-written using a DSL, and a DSL compiler translates it to synthesisable code directly, or by using a

synthesisable code generator. The generated synthesisable HLL code is then used by a HLS tool to generate an RTL description. DSL tools for HLS are discussed in Section III-C.

Dataflow descriptions are another way to express compute-intensive HLL functions for FPGA implementation. This is illustrated as design flow ③ in FIGURE 1. A dataflow description represents a system as a directed graph, known as dataflow graph (DFG), with nodes as computational units and the edges as data flow between them. The asynchronous and concurrent behaviour of the computation units provide a good match with likely implementations on FPGAs. Dataflow descriptions can be defined at different levels of abstraction. We will discuss this design flow in Section IV under two subcategories:

- *Dataflow HLS* ③a: In this approach, a synthesisable DFG is manually described using a meta-language (e.g. MaxJ [12], CAL [13]), and then transformed into an RTL description using an appropriate HLS tool. Section IV-A surveys efforts using this approach.
- *High-level DFG representation* ③b: Some tools allow developers to define DFGs at higher level of abstraction, and transform them to synthesisable representations. Then, using dataflow HLS they are translated to RTL descriptions. These tools usually work for particular application domains. The effort to define DFGs at high levels of abstractions are discussed in Section IV-B.

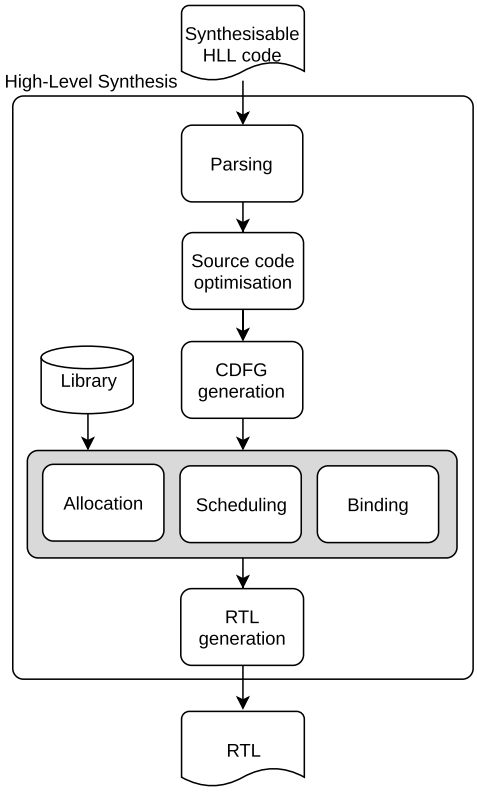


FIGURE 2. A generic HLS design flow.

III. BEHAVIOURAL APPROACH FOR FPGA SYNTHESIS

A common and straightforward approach to deploy a compute-intensive function on an FPGA is to write a behavioural description of that function in a HLL as a sequence of statements with control structures such as loops and conditions. Existing tools support a variety of HLLs including including C, C++, Java, Python, C# and Matlab and are able to transform the behavioural description into an RTL description in HDL (typically VHDL, Verilog or SystemVerilog). This design flow is labelled as ② in FIGURE 1. Variations of this design flow are discussed in the following subsections.

A. BEHAVIOURAL HIGH-LEVEL SYNTHESIS (HLS) ②A

High-Level Synthesis (HLS) allows developers to define hardware using a HLL (e.g. C, C++ or SystemC). A HLS tool automatically transforms the untimed HLL code to a fully timed circuit specification in HDL to perform the same function on an FPGA. The introduction of HLS in the design flow reduces the transition from the HLL code to the start of the automatic process, as can be seen from the design flow in FIGURE 1.

FIGURE 2 shows a generic HLS design flow to synthesise an accelerator on an FPGA as in [14], [15]. The flow starts with a behavioural description, which is written in a synthesisable way using a HLL. Not all HLL constructs are synthesisable by all HLS tools. For example, many HLS tools do not support pointers, recursion or dynamic memory allocation. After *parsing* the source code during compilation, the HLS tool generates an intermediate representation (IR) and performs *source code optimisations* such as dead-code elimination and redundant expression elimination. Most HLS tools are based on the LLVM compiler infrastructure [16], which generates a *Control and Data Flow Graph (CDFG)* to capture all dependencies of the IR. The HLS tool determines the right functional units, memory elements and connectivity components to use from a design library during *resource allocation*. In the *scheduling* step, each operation from the source code is assigned to a control step, which will correspond to system clock cycles in the hardware. Next, in *resource binding*, the HLS tool maps operations to specific functional units, variables to memory elements, and data transfers to interconnection components such that data can be correctly computed and passed, according to the scheduling. Finally, an RTL model of the synthesised design is produced in the *RTL generation* step. Depending upon the scheduling and the binding information, interconnections between the circuit modules of the datapath are created, and a finite state machine (FSM) is generated to control the data flow in the datapath.

HLS has been the subject of continuous research since the 1970s, but these efforts were not notably successful until the early 2000s. The historical evolution of HLS tools and the reasons behind the success and failure of different generations of HLS development are discussed in [17], [18].

TABLE 1. Currently available commercial HLS tools.

Tool	Provider	Intro.	Update	Pedigree	Platform	Input	Output	Design tool	Domain	Supported features								
										PR	P2P	P2F	FxP	FP	AP	RF	DM	AN
Vivado HLS [23]	Xilinx	2008	2019	xPilot (UCLA) → AutoPilot (AutoESL) → Vivado HLS (Xilinx)	Xilinx FPGAs	C, C++, System C, OpenCL API	Verilog, VHDL	Included in Vivado Design Suite	All	●	◐	○	●	●	●	○	○	●
Intel FPGA SDK for OpenCL [24]	Intel (Altera)	2013	2019	Intel acquired Altera in 2015	Heterogeneous platforms with Intel CPUs and FPGAs	OpenCL kernels	Verilog	Stand-alone	All	●	○	○	●	●	●	○	○	●
Intel HLS Compiler [35]	Intel (Altera)	2017	2019	Intel acquired Altera in 2015	Intel FPGAs	C++	Verilog	Included in Intel Quartus Prime Design	All	●	○	○	●	●	●	◐	○	●
Catapult HLS [36]	Mentor Graphics	2004	NI	Catapult-C (Mentor Graphics) → Catapult-C (Calypto, 2011) → Catapult HLS (Mentor Graphics, 2015)	All FPGAs and ASICs	C++	Verilog, VHDL	NI	All	◐	○	○	●	●	●	○	○	●
Stratus HLS [32]	Cadence	2015	NI	Cynthesizer (Forte) + C-to-Silicon Compiler (Cadence) → Stratus HLS (Cadence)	All FPGAs and ASICs	C++, SystemC	Verilog	NI	All	●	○	○	●	●	●	○	○	●
CyberWorkBench [43]	NEC	2011	NI	Based on Cyber [44]	Xilinx and Altera FPGAs	BDL	Verilog, VHDL	Stand-alone	All	●	○	○	●	●	●	○	○	●
C-to-Hardware [45]	Altium	2008	2019		Xilinx, Intel and Microsemi FPGAs	C	Verilog, VHDL	Included in Altium Designer	All	●	○	○	●	●	○	○	○	●
Synphony HLS [46]	Synopsys	2003	2017	Based on PICO [47]	All FPGAs and ASICs	C, C++, Simulink models	Verilog, VHDL	Used as a blockset in Simulink	DSP	○	○	○	●	○	○	○	○	○
HDL Coder and Verifier [48]	Mathworks	2012	2019		Xilinx, Intel and Microsemi FPGAs	Matlab functions and Simulink models	Verilog, VHDL	Used as an add-on in MATLAB and Simulink	All	○	○	○	●	○	○	○	○	○

NI = No Information found
PR = Pointer, *P2P* = Pointer-to-pointer, *P2F* = Pointer-to-function, *FP* = Floating point, *FxP* = Fixed-point, *AP* = Arbitrary bit precision
RF = Recursive function, *DM* = Dynamic memory allocation, *AN* = Annotations required
● = Supported, ○ = Not supported, ◐ = Supported with restrictions

The authors of [18] describe a state-of-the-art HLS tool of the time, AutoPilot [19], which was later acquired by Xilinx, and renamed Vivado HLS. The major research efforts in compiling HLLs for reconfigurable computing were surveyed in [14], [20]. The paper [21] presented an evaluation of more recent HLS tools in terms of capabilities, usability and quality of results. A survey of HLS tools and compilers targeting heterogeneous high performance computing was presented in [15]. The authors of [22] provided studies to compare the performance of three commercially available HLS tools: Vivado HLS [23], Intel FPGA SDK for OpenCL [24], and MaxCompiler [12]. A comprehensive survey of commonly used open-source HLS tools appears in [25].

A recent survey on HLS tools along with a comparative study of three academic HLS tools – Bambu [26], DWARV [27] and LegUp [28] – was published in [29]. This article surveyed an exhaustive list of HLS tool including abandoned ones. Since it was published, some of the tools have merged (e.g. Forte Cynthesizer [30] and Cadence C-to-Silicon Compiler [31] are combined in Stratus HLS [32]), some have lost their importance (e.g. eXCite [33], NAPA-C [34]), and some interesting new tools have appeared in the industry and academia (e.g. Intel FPGA SDK for OpenCL [24]). We provide updated lists of the currently available commercial and academic HLS tools in TABLE 1 and TABLE 2. Here, we consider only those tools that are in

TABLE 2. Currently available academic HLS tools.

Tool	Provider	Intro.	Update	Pedigree	Platform	Input	Output	Design tool	Domain	Supported features								
										PR	P2P	P2F	FxP	FP	AP	RF	DM	AN
LegUp [28], [49]	University of Toronto and LegUp Computing	2011	2020		Intel FPGAs and limited support to Xilinx FPGAs	C	Verilog	Stand-alone	All	●	○	○	○	●	○	○	○	○
Bambu [26]	Politecnico di Milano	2012	2017		All FPGAs	C	Verilog	Compatible with Xilinx Vivado Design Suite, Altera Quartus and Lattice Diamond	All	●	○	○	○	●	○	○	○	○
ROCCC [37], [38]	UC Riverside and Jacquard Computing	2002	2013	Inspired by SA-C compiler	FPGAs	A sub-set of C	VHDL	Stand-alone	Streaming applica-tions	○	○	○	○	●	○	○	○	○
GAUT [39], [40]	Université de Bretagne-Sud	2008	2017		Xilinx and Intel FPGAs	C, C++	VHDL	Compatible with Vivado Design Suite and Synopsys Design Compiler	DSP	○	○	○	●	○	●	○	○	◐
Kiwi Compiler [41], [42]	University of Cambridge	2008	2017		Xilinx and Altera FPGAs	.NET byte-code	Verilog	Stand-alone	All	●	●	○	○	●	○	◐	●	●
Handel-C Compiler [50], [51]	Oxford University Computing Laboratory	1996	NI		Xilinx and Altera FPGAs	Handel-C	Verilog, VHDL	Used in DK Design Suite [52] from Mentor Graphics	All	○	○	○	●	○	○	○	○	○
DWARV [27]	Delft University of Technology	2007	2014		FPGAs	C	VHDL	Included in Delft Work-bench [53]	All	●	○	○	●	●	○	○	○	●

NI = No Information found
PR = Pointer, *P2P* = Pointer-to-pointer, *P2F* = Pointer-to-function, *FP* = Floating point, *FxP* = Fixed-point, *AP* = Arbitrary bit precision
RF = Recursive function, *DM* = Dynamic memory allocation, *AN* = Annotations required
● = Supported, ○ = Not supported, ◐ = Supported with restrictions

use, under active research and popular in industry. We also provide the pedigrees of HLS tools, if available.

The choice of a HLS tool is often dominated by available applications, price, community support and its support for FPGAs from a particular vendor, such that no specific HLS tool dominates the industry. However, Xilinx Vivado HLS [23], Intel HLS Compiler [35], Intel FPGA SDK for OpenCL [24], Mentor Catapult [36], and Cadence Stratus [32] are some popular commercial HLS tools. LegUp [28], Bambu [26], ROCCC [37], [38], GAUT [39], [40] and Kiwi Compiler (KiwiC) [41], [42] are some notable HLS tools produced as research outputs. We will discuss these commercial and academic HLS tools in this section. Other available HLS tools are mentioned in TABLE 1 and TABLE 2. It should be noted that it is generally harder to find the design details (e.g. compiler architecture) of commercial tools than those of research products. Commercial tools built upon university research outputs (e.g. Vivado HLS [23]) have more information in the literature than those built entirely in industry (e.g. Intel HLS Compiler [35]).

1) VIVADO HLS

Vivado HLS [23] is a commercial HLS tool provided by Xilinx for their own FPGAs. It is based on AutoPilot [19], which was actually a commercialisation of the xPilot system [54] developed in the University of California, Los Angeles. Vivado HLS provides a design environment to generate RTL descriptions in VHDL and Verilog from synthesisable C, C++, SystemC or OpenCL HLL code. Being a commercial tool, the internal design flow of Vivado HLS is not available but it seems reasonable to assume it adopts the same design flow as AutoPilot, which is shown in FIGURE 3. Vivado HLS is based on LLVM compiler infrastructure, which compiles the HLL code to a LLVM-IR. The LLVM-IR then passes through a series of standard compilation tasks including dead-code elimination, constant propagation and loop unrolling, and hardware-specific optimisations such as bit-width optimisation for reducing code complexity and redundancy, maximising data locality and exposing parallelism. Vivado HLS uses the modified IR to perform synthesis and interconnect-centric optimisations during operation scheduling and resource binding phases taking user-specified

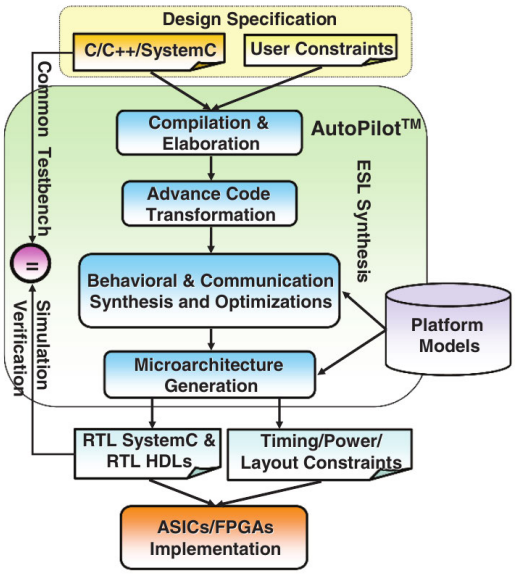


FIGURE 3. AutoPilot design flow [19].

constraints into account. Vivado HLS has a rich set of annotations to direct the compilation and synthesis processes to optimise performance, resource utilisation, data communication between CPU and custom hardware. It also contains many hardware-optimised libraries and APIs to support hardware developers. Finally, the IR is synthesised into RTL implementations for Xilinx FPGAs. The generated RTL can be saved in a IP library for later use. Vivado HLS also provides features to verify the functionality of the RTL against HLL description using a testbench. This HLS tool is included in Vivado Design Suite [55], and can be used with heterogeneous system development tools SDSoC [56] and SDAccel [57]. In Xilinx's recently released Vitis Unified Software Platform [58], it is named as Vitis HLS with some additional functionalities.

2) INTEL FPGA SDK FOR OpenCL

Open Computing Language (OpenCL) is a framework for programming parallel applications for heterogeneous platforms including processors, GPUs and FPGAs. It is an open standard maintained by the Khronos Group [59]. Altera introduced an OpenCL SDK for multi-core programming in 2013. Intel acquired Altera in 2015 and renamed the product as the Intel FPGA SDK for OpenCL [24]. Instead of executing parallel threads of expensive functions on multiple cores, this compiler generates deeply pipelined hardware circuits that can be implemented on Intel FPGAs. The tool transforms OpenCL to Verilog RTL and runtime libraries for the part of the system running on the processor. It uses LLVM-Clang [60] to parse OpenCL constructs and produces a IR. The IR passes through a series of optimisations including branch elimination, loop fusion and auto vectorisation. Users guide optimisation by inserting annotations for loop unrolling, pipelining and streaming data. The compiler automatically applies these optimisations and translates the optimised IR to Verilog RTL.

3) INTEL HLS COMPILER

The Intel HLS Compiler [35] transforms a high-level hardware description written in C++ to an RTL description for Intel FPGAs. In addition to standard compiler optimisations, the compiler performs optimisation tasks such as loop unrolling, data dependency and pipelining based on user-provided annotations. It allows the user to explore hardware architectures including interfaces, parallelism, memories, datapaths and loops using specific attributes and annotations. Like Vivado HLS, Intel HLS Compiler also facilitates generation of reusable IPs for system integration, and reduces FPGA development time. The tool supports software testbench verification against the generated RTL. It is included in the Intel Quartus Prime Design Software Suite [61].

4) CATAPULT HLS PLATFORM

The Catapult HLS Platform [36] was initially developed by Mentor Graphics as Catapult-C [62], and was later owned by Calypto Design Systems [63]. It was reacquired by Mentor Graphics in 2015. Using Catapult HLS, developers can define hardware using a subset of C++ and SystemC and generate optimised Verilog and VHDL description for FPGAs and ASICs. Catapult supports most C++ constructs including classes, structs, arrays and pointers to statically allocated objects. However, it does not support dynamic memory allocation. It allows developers to include integers of arbitrary length, fixed-point, floating-point and complex data types using Mentor Graphics's Algorithmic-C data types [64]. Hardware developers can specify parallelism, throughput and memory configuration at high levels of abstraction using attributes and annotation; however an in-depth understanding of the underlying hardware is required to achieve a good result. During hardware synthesis, Catapult performs a number of optimisations including loop unrolling, loop merging and pipelining. Catapult can automatically generate simulation infrastructure for verifying HLS-generated RTL against the original HLL source code. Catapult HLS also allows incorporating specification changes and porting to a different technology by the separation of the design functionality and the implementation details. The RTL can simply be regenerated based on the modified HLS model and new constraints.

5) STRATUS HLS

Stratus HLS [32] is a popular HLS tool provided by Cadence. Cadence acquired Forte Design Systems in 2014 and tied Forte's Cynthesizer [30] with its own C-to-Silicon Compiler [31] under the name Stratus HLS. The HLS tool accepts hardware specification in SystemC and C/C++ and generates a Verilog RTL utilising high-level implementation constraints. By separating these constraints from hardware functionality, Stratus HLS allows the verified HLL hardware specification to be reused as behavioural IPs without

modification for different platforms and clock speeds. The HLS tool allows hardware developers to design hardware hierarchically to manage design complexity. By supporting hierarchical decomposition, it allows design and verification of multiple functions and their interfaces operating concurrently.

6) CyberWorkBench

CyberWorkBench [43] is a HLS tool developed over 25 years of research in NEC. It allows developers to write SoC applications in C for the software part and in Behavioural Description Language (BDL) [65] for hardware accelerators. The BDL is an extended C language that excludes non-synthesisable constructs (such as dynamic allocation with pointers and recursions) but includes a number of hardware-specific constructs (such as user-defined variable bitwidth and explicit clock boundary specification). CyberWorkBench uses the Cyber behavioural synthesiser [44] to transform the BDL description into RTL descriptions in Verilog or VHDL. The tool provides automatic pipelining and parallelism extraction, which are guided by attributes in the source code and global synthesis options. CyberWorkBench is built based on the idea "all-in-C", as described in [66], meaning all modules including control-intensive and data-dominant circuits are described in BDL. The synthesis and verifications are also done at the higher level of abstraction. The tool has its own verifier to check functionality of the generated RTL against the BDL description.

7) LegUp

LegUp [28], [49] is an open-source HLS compiler developed by researchers at the University of Toronto, and now supported by LegUp Computing Inc. It accepts ANSI C code as input and generates Verilog code that can be synthesised onto FPGAs. LegUp supports FPGAs provided by Intel, Xilinx, Lattice, Microsemi and Achronix. It works in two modes: hardware-only and software-hardware modes. In *hardware-only mode*, the entire input C code is synthesised to RTL. In *software-hardware mode*, the high-level C code is profiled using a built-in profiler [67] to identify compute-intensive functions for hardware offloading. After manual selection of the functions for hardware acceleration, LegUp synthesises a heterogeneous system comprising a processor and accelerators built on an FPGA. The tool supports most C features for FPGA synthesis but not recursion or dynamic memory allocation. LegUp is built using the LLVM framework with Clang as its frontend. It allows hardware optimisations to perform function inlining, loop unrolling and source code modifications to parallelise sequential executions using Pthreads or OpenMP annotations. However, the insertion of these annotations is a manual process.

8) BAMBU

Bambu [26] is an open-source HLS framework built in Politecnico di Milano under the PandA project [68]. It takes a behavioural description in C and an XML file specifying the configurations corresponding to different stages of the design flow. It uses the *gcc* compiler to perform target and language-independent optimisations and produces IRs in the form of a call-graph and CDFG. Bambu generates hardware modules for the functions separately, reflecting the structure of the call-graph. The modules include datapaths, FSM controllers and memory interfaces. Finally, the HLS tool combines these modules to generate an HDL description and produces logic synthesis scripts for the desired toolchains, as specified in the XML file. Currently Bambu is compatible with Xilinx Vivado Design Suite [55], Intel Quartus [61] and Lattice Diamond [69]. Bambu also produces testbenches and scripts for RTL simulation that can be run in Xilinx simulators, Mentor Modelsim [70], Verilator [71] and Verilog Icarus [72].

9) ROCCC

Riverside Optimizing Compiler for Configurable Circuits (ROCCC) [37], [38] is an open-source HLS compiler initially developed at the University of California Riverside. The latest versions of this tool have been developed by Jacquard Computing, who are currently maintaining it. The developers of this tool consider that it is a *code accelerator* instead of a HLS tool because "typically, accelerators have their semantic root in a loop nest while a general HLS tool can target any arbitrary C code. Hence, the focus of the ROCCC compiler transformations has been on loop nests" [37]. FIGURE 4 illustrates a high-level design flow of the ROCCC compiler. It does not compile entire applications to hardware, rather it allows *modules* and *system code* to be transformed to hardware. Modules refer to compute-intensive functions that can be synthesised to hardware modules. They can be reused as hardware building blocks in other modules. System code blocks perform computations on large streams of data using repeated loop iterations. The may or may not be synthesised to hardware modules but they usually result in hardware for memory interfaces. The modules and system code can be written using a subset of C constructs. Common C features including generic pointers, dynamic memory allocation, recursion, including C-library calls, and while loops are not supported by the ROCCC compiler. The compiler performs high-level transformations, such as loop and array transformations, using the SUIF compiler infrastructure [73] and generates an IR. The IR passes through the LLVM framework for low-level optimisations, such as pipelining, before VHDL

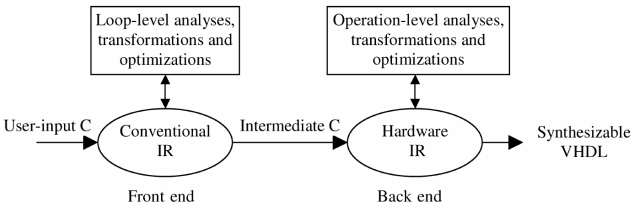


FIGURE 4. ROCCC design flow [74].

code is generated. It does not require annotations in the HLL source code to generate efficient pipelined hardware.

10) GAUT

GAUT [39], [40] is an open-source HLS tool developed at Université de Bretagne-Sud, France for DSP applications. It generates a RTL description in VHDL from bit-accurate high-level C/C++ code. The compiler allows developers to use the Algorithmic-C library [64] from Mentor Graphics for signed and unsigned bit-accurate integer and fixed-point variables. The design flow of GAUT is depicted in FIGURE 5. This *gcc*-based HLS tool generates a bit-width information annotated dataflow graph (DFG) of the application, which is then converted to GIMPLE-IR. The IR undergoes performance optimisation including dead-code elimination, redundancy eliminations and loop optimisations, and instruction-level parallelism is extracted from it. GAUT performs resource allocation, operation scheduling and binding, and storage optimisation on the IR, and finally generates a VHDL and/or SystemC description for a particular FPGA platform. During this process, GAUT uses a library of time-characterised operators that were generated using DFG and logic synthesis tools from Xilinx and Intel. The generated RTL contains a controller FSM, a datapath, and memory and communication interfaces. The tool also generates necessary scripts and data for testbenches for simulations using the Modelsim simulator. The generated RTL can be synthesised using Xilinx Vivado Design Suite [55], Intel Quartus [61] and Synopsys Design Compiler [75].

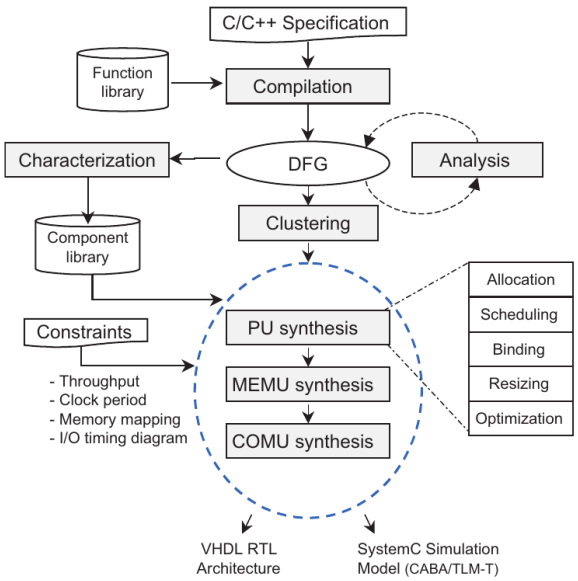


FIGURE 5. GAUT design flow [39].

11) KIWI COMPILER

Kiwi Compiler (KiwiC) [41], [42] was developed as an open-source tool at the University of Cambridge and Microsoft Research Limited, and is now maintained by the university. Most HLS tools accept C-like sequential programs and synthesise them for FPGAs. This HLS compiler and its associated library allow programmers to model parallel computations using a subset of the C# language and transforms them into circuits for realisation on Xilinx and Intel FPGAs. The compiler converts.NET bytecode generated as an IR from Microsoft.NET or Mono C# compilers to Verilog RTL. It maps system-level concurrency such as events, monitors and thread onto appropriate hardware implementations. The Kiwi Compiler is based on the Value State Flow Graph (VSFG) compilation technique [76]. The compilation technique exploits dynamic execution scheduling, aggressive branch prediction and loop unrolling to achieve significant performance improvements. KiwiC supports a wider subset of HLL features than many other HLS tools, including multi-dimensional arrays, threading, file-server I/O, object management and limited recursion. The tool also provides a *performance predictor* to allow users to explore the expected speed-up without fully synthesising the hardware.

B. SYNTHESISABLE HLL CODE GENERATION (2b)

Although HLS tools allow design space exploration more quickly and easily by avoiding the restrictions of HDLs, developers still need to be aware of hardware implementation constraints while writing HLS code. Naive HLS code written without an understanding of the resultant hardware structure often yields results worse than simply running the code on a processor. Moreover, HLS tools often use C-based HLLs as inputs, and the generic C/C++ programming model does not take advantage of the concurrent nature of an FPGA. Hence, developers have to manually do complex code restructuring and insert special directives to generate synthesisable code and efficient hardware. Authors of [77] provide a guideline for transforming high-level C++ to efficient synthesisable code. But this manual process significantly impacts the productivity of system development and this situation becomes more acute for complex applications with a large base of existing code written for conventional computers.

Investigations are underway in academia and industry for generating efficient synthesisable code (i.e. HLS code) from existing HLL. After identifying compute-intensive functions, these tools analyse the source code to extract parallelism before generating synthesisable code using their own source-to-source compilers. The generated code is then fed into existing HLS tools to generate accelerator code in RTL for deployment on FPGAs. This approach is shown in FIGURE 1 on Page 174694 as design flow (2b). Because this process still involves human interventions in many cases, we consider it to be "semi-automatic". The major synthesisable code generation tools are listed in TABLE 3 and will be discussed in the following subsections.

1) SLX-FPGA

Silexica developed the SLX-FPGA Tool Suite [78] to help convert non-synthesisable C/C++ code to synthesisable HLS C code for Xilinx's Zynq SoC and MPSoC devices. It uses

TABLE 3. HLS code generation tools.

Tool	Provider	Open source	Intro.	Update	Pedigree	Platform	Input	Output	Anno-tations	Target HLS	Domain
SLX-FPGA [78]	Silexica	No	2017	2019	Based on MAPS project [79]	Xilinx Zynq SoC/ MPSoCs	C, C++	HLS C	Yes	Vivado HLS	All
Merlin Compiler [80]	Falcon Computing Solutions	No	2016	2018		Xilinx and Intel FPGAs	C, C++	HLS C/ OpenCL	Yes	Vivado HLS, Intel OpenCL Compiler	All
Hot & Spicy [81]	University of Southern California	Yes	2018	2018		Xilinx FPGAs	Python	HLS C	Yes	Vivado HLS	All
Delft Workbench [53]	Delft University of Technology	Yes	2014	NI		Hetetogeneous platforms with CPUs and FPGAs	C	VHDL	Yes	DWARV [27]	All
FROST [82]	Massachusetts Institute of Technology	No	2017	NI		Xilinx FPGAs	FROST IR	HLS C/C++	No	Vivado HLS	All
CAOS [83]	Politecnico di Milano	Yes	2017	2019		Xilinx SoC/MPSoC	C, C++, OpenCL	HLS C/C++	NI	Vivado HLS	All

NI = No Information found

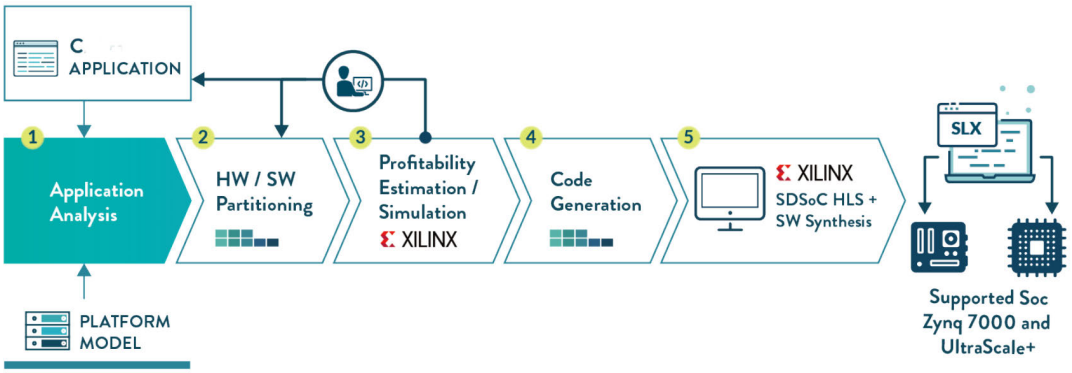


FIGURE 6. SLX-FPGA tool flow [78].

Vivado HLS [23] and SDSoC [56] under the hood to implement the HLS code onto the Zynq devices. SLX-FPGA also works with Xilinx's recently released Vitis Unified Software Platform [58]. This tool suite is based on an academic project called MPSoC Application Programming Studio (MAPS) [79], which started in 2007 at RWTH Aachen University.

FIGURE 6 illustrates the tool flow of SLX-FPGA. It first analyses an existing conventional C/C++ application and identifies parallelisation opportunities in the form of task, data and pipeline level parallelism. It performs static analysis of the source code and dynamic analysis of the executable code, and converts the source code to a language called *C for Process Networks (CPN)*. This language is a C extension that models concurrent processes and applications. Next, it suggests a partitioning of the application between the embedded host processor and FPGA regions of the Zynq device by analysing the computation and communication behaviour of the CPN specification. The expected benefits of this partitioning are then evaluated using Xilinx's performance estimation engine. Finally, the tool suite performs source-to-source translation to emit HLS C code ("synthesisable behavioural code" in path ②b) considering both the CPN and the mapping configuration generated by the SLX Mapper. Vendor-specific annotations for loop unrolling and pipelining are automatically inserted into the HLS code to extract parallelism and define interfaces based on optimisation decisions taken by the programmer. SLX-FPGA also provides hints for manual code restructuring to make any designated non-synthesisable functions synthesisable. The generated HLS C code then passes through the Xilinx toolchain including Vivado HSL (the "behavioural HLS tool" in path ②b) for implementation onto the Zynq devices.

SLX-FPGA provides some useful features that move the state of the art closer to the ACD, however it is still not fully automatic. It does not perform automatic code refactoring to make a non-synthesisable function synthesisable, or to optimise the performance of the generated hardware accelerator. Although the latest release of SLX-FPGA supports some Vivado HLS annotations for loop unrolling, pipelining, array partitioning and function inlining, it still does not support all of the available annotations to extract best possible performance in the generated hardware.

2) MERLIN COMPILER

Merlin Compiler [80] is a synthesisable code generation tool introduced by Falcon Computing Solutions that automates the code rewrite effort for FPGA programming. It accepts C/C++ HLL code as input and transforms it to optimised OpenCL code that can be synthesised on Xilinx and Intel FPGAs using the vendor-specific toolchains. FIGURE 7 illustrates the design flow of Merlin Compiler.

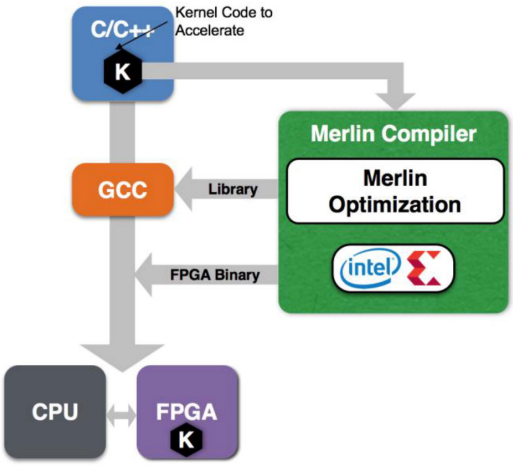


FIGURE 7. Merlin Compiler design flow [80].

Merlin Compiler can work in two modes. In *manual optimisation* mode, developers manually annotate the source code with a small set of OpenMP-like annotations for pipelining and parallelism. Developers who are not familiar with FPGA constructs can use *automatic optimisation* mode, in which they can annotate functions for acceleration, and the compiler will use a machine learning based *Deep Space Exploration (DSE)* to perform micro-architecture optimisations, such as global memory bursting and coalescing, memory partitioning, data reuse and data flow streaming. Merlin Compiler produces synthesisable OpenCL code (the "synthesisable behavioural code" in design flow (2b)) that is transformed to binary files for FPGA configuration using Xilinx SDAccel [57] or Intel OpenCL SDK [24] (the "behavioural HLS tools" in path (2b)) in the background. Although, the automatic mode can accelerate the application, there is more scope for improving performance by manually annotating the code. Moreover, in either mode the developer must still modify the code with annotations, and to include header files and initialise the Merlin runtime libraries.

Merlin compiler provides various verification strategies at different stages of the process. It generates an optimised C program before OpenCL generation, which can be compared with the input C program in CPU execution to confirm functional correctness of the accelerator. The generated OpenCL is also verified by a CPU emulation, and RTL generated by HLS is verified by co-simulation.

Merlin Compiler is available on the AWS marketplace so that FPGA-accelerated applications can be developed using

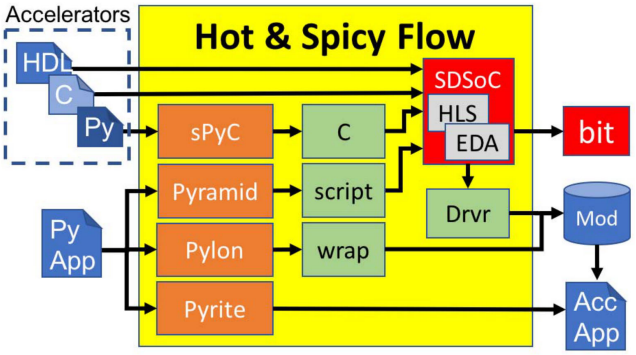


FIGURE 8. Implementation flows using the Hot & Spicy tools [81].

Amazon AWS F1 instances that contain Xilinx FPGA accelerator cards. In addition to cloud-based data centres, it also supports on-premise data centres that include Intel and Xilinx FPGA acceleration cards for data processing.

3) HOT & SPICY

Researchers at the Information Sciences Institute, University of Southern California, have developed Hot & Spicy [81], an open-source framework and toolchain for exploiting FPGA accelerators in applications developed completely in Python. The Hot & Spicy workflow consists of: the Python-to-C or *sPyC* tool, which is a source-to-source translation tool for Python syntax (at the function level) to HLS C/C++ code for the Xilinx Vivado HLS tool; the Python linker, *Pylon*, which generates C API wrappers/bindings for linking the Python application to C/C++ call accelerators; the Python rewriter, *Pyrite*, which refactors the original Python source code to make importing the accelerators possible via the wrappers generated by Pylon (instead of the Python functions, the C functions are called); and *Pyramid*, which generates scripts to drive the EDA implementation flow (i.e. by employing the Xilinx Vivado HLS through invoking the Xilinx SDSoC tool to generate a complete system design).

4) DELFT WORKBENCH

Delft Workbench (DWB) [53] is a toolchain to develop heterogeneous systems applications. As shown in FIGURE 9, a high-level application is first profiled and characterised using Quipu [84] and Quad [85] to identify candidate hardware functions. Quipu predicts the hardware resources needed to implement different functions on a reconfigurable hardware platform. Quad provides an overview of the memory access behaviour of the application. Based on the profiling information, the application is manually annotated, and partitioned and mapped to the processor and FPGA. The compute-intensive functions are mapped to the FPGA, and the reminaing functions are executed on the processor. It uses the DWARV (Delft Workbench Automated Reconfigurable VHDL) HLS compiler [27] (the "behavioural HLS tool" in path (2b)) to translate the selected functions to VHDL RTL code for synthesis for the FPGA. The HLS compiler

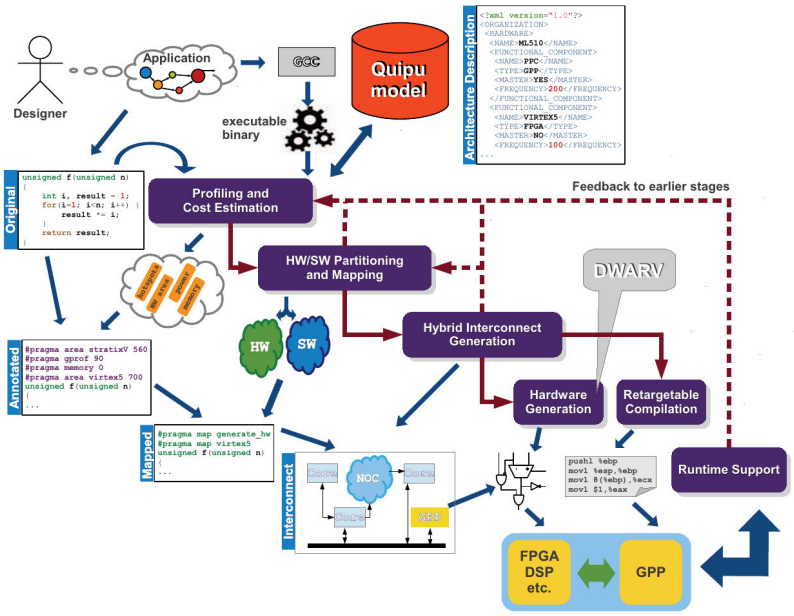


FIGURE 9. Delft Workbench toolchain [53].

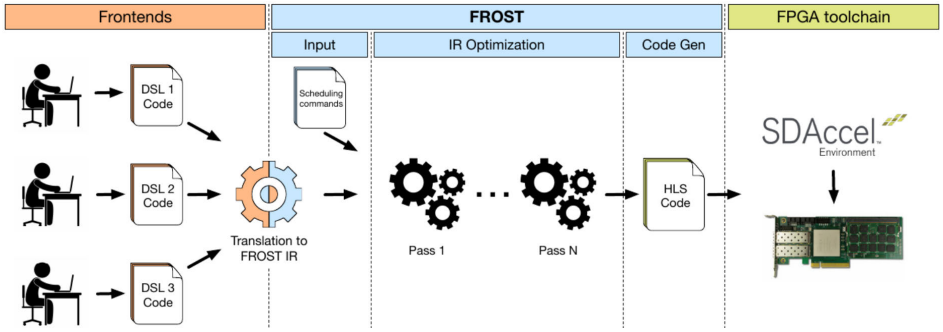


FIGURE 10. FROST design flow [86].

computes common expressions and ensures data locality by putting the results in registers for subsequent use, performs register allocation based on operation chaining, and allocates register and memory interfaces for parallel memory accesses. The toolchain implements a network-on-chip (NoC) based custom interconnect for communication among the accelerators. It automatically invokes the architecture-specific synthesis tools to generate bitstreams from the generated VHDL.

5) FROST

Research publications [82], [86] introduce FROST as a common backend for Domain-Specific Language (DSL) compilers (to be discussed in Section III-C) such as Halide [87] and Tiramisu [88], targeting FPGAs. As presented in FIGURE 10, DSL compilers generate IRs (the "intermediate representation" in path (2b)), and use FROST to generate efficient HLS code for Xilinx FPGAs. In these IRs, functions are represented as abstract syntax trees (ASTs). FROST takes an IR and a list of scheduling commands to produce an efficient hardware implementation. It provides a way for the hardware developer to apply guided optimisations at different design levels as well as to select the best architecture for realising the function on the target FPGA. Using the scheduling commands, a developer can specify FPGA-specific optimisations such as loop pipelining, unrolling and vectorisation, as well as the type of communication with the off-chip memory. FROST analyses the IR and manipulates function ASTs to apply FPGA-oriented transformations and optimisations. It then analyses the updated ASTs to identify libraries and variable data types for HLS code generation. FROST generates HLS-friendly C/C++ code (the "synthesisable behavioural code" in path (2b)), and applies the remaining scheduling commands as HLS annotations. The generated HLS code can be synthesised and implemented on Xilinx FPGAs using the SDAccel toolchain, which involves Vivado HLS (the "behavioural HLS tool" in path (2b)) in it.

6) CAOS

CAD as an Adaptive Open-platform Service (CAOS) [83] is a design platform developed to help application designers identify and optimise kernel functions for hardware acceleration. It also helps produce efficient, fine-tuned, accelerators for

the FPGA, by generating the runtime management and the configuration files needed.

The tool takes as inputs the application code written in C/C++, datasets for code profiling, and details of the target FPGA. Users are encouraged to make use of pre-defined architectural templates, which describe the computational model of the hardware accelerator and how it communicates with the off-chip memory. This not will only improves the performance of the hardware implementation but also facilitates and improves the analyses and optimisations that might need to be carried out on the corresponding algorithms later in the CAOS workflow.

The CAOS interactive design flow consists of three phases: *frontend*, *function optimisation*, and *backend*. In the frontend phase, CAOS generates an IR. Using the IR and through application profiling, it then pinpoints the bottleneck functions (core/kernel functions) that may benefit from being offloaded onto the FPGA as accelerators. It also gives some hints to the designer on how the application can be partitioned into FPGA- and CPU-friendly functions. To do that, the CAOS frontend performs an applicability check which is meant to find suitable architectural templates for the application code [89].

In the functional optimisation phase, based on the architectural template suggested in the frontend phase, and using a series of static analyses and hardware resource estimations on the candidate functions for FPGA acceleration, CAOS provides performance and resource estimates for the functions offloaded to the FPGA, and suggests a range of optimisation strategies including, but not limited to, loop pipelining, loop tiling and loop unrolling. This design loop may iterate until the system developer is satisfied with the application performance and the design meets the provided constraints.

In the backend phase, CAOS produces both the runtime for executing the CPU functions and the final bitstreams needed to configure the FPGAs within the system, by leveraging the specific FPGA vendor tools (high-level synthesis and hardware synthesis tools) including Xilinx's SDAccel and Vivado HLS as well as MaxCompiler from Maxeler Technologies, based on the selected architectural template. Wherever possible, the backend can also support runtime reconfiguration of the FPGA devices through partial dynamic reconfiguration [90].

C. DOMAIN-SPECIFIC LANGUAGE (DSL) FOR HLS (2c)

Although HLS tools are improving, the benefit delivered by an FPGA hardware accelerator still depends strongly on the system developers' hardware expertise.

Domain-Specific Languages (DSLs) are another promising step towards design automation [91]. A DSL can be used to describe a domain-specific computing system at a higher level of abstraction than an HDL. This frees the system developers from hardware-influenced programming details so they can focus on describing the domain-specific design. At the same time the DSL can improve the efficiency of the FPGA accelerator by matching computing patterns typical of the domain with well optimised hardware. TABLE 4 shows current DSL tools for HLS.

1) OptiML DSL

An automated design framework for realising FPGA accelerators from high-level programs written in OptiML is presented in [92], [102]. OptiML is a Scala-embedded machine learning DSL implemented using the Delite compiler framework [103], which provides a programming environment similar to MATLAB that supports machine learning code structures.

Using OptiML the parallelism of the application can be efficiently identified, possible optimisation opportunities can be located, domain-specific operations can be mapped into the corresponding structured computation patterns and, as a result, the best architecture template (which describes how different hardware modules in the final realisation need to be connected) for FPGA implementation can be suggested.

As shown in FIGURE 11 the automated OptiML methodology first takes an application program written in OptiML. Then the Delite compiler (the "DSL compiler" in path (2c) in FIGURE 1) applies general optimising transformations, as well as domain-specific algorithms (based on the DSL domain). The Delite compiler also extracts DSL operations from the input and maps them onto a collection of serial/parallel computational patterns or kernels (the "intermediate representation" in path (2c) in FIGURE 1). In addition, the compiler constructs the dependency graph expressing the order those kernels must be executed in the final FPGA implementation. The Delite compiler also selects the best system-architecture template that expresses the way hardware kernels are needed to interconnect on the FPGA.

Then for every kernel received, the kernel synthesis stage (the "synthesisable code generator" in FIGURE 1) synthesises multiple hardware realisations (or variants) with different area/performance trade-offs but with the same computational functionality as the corresponding kernel, using the configuration information about the target FPGA as well as the nominated system-architecture template.

The next step in the flow, system synthesis takes the many possible hardware realisations and all the information extracted during compilation, and finds the combination of hardware variants that provides the best performance within the design constraints and the FPGA resource budget. Using the system-architecture template, the selected variants are connected together to form the final design's data path. This stage also extracts the final hardware system's control unit from the application's dependency graph generated by the Delite compiler. In the final step of the workflow, the Xilinx Vivado HLS (the "behavioural HLS tool" in FIGURE 1) is used to generate the RTL representation, produce the bitstream and program the target FPGA.

Similar to OptiML, there are OptiQL, OptiMesh and OptiGraph DSLs for data querying and transformation, mesh computation and graph analysis [104] applications

TABLE 4. Current DSL tools for HLS.

Tool	Provider	Intro	Domain	Platform	Input	Output	Target HLS	Related tool
OptiML [92]	Ecole Polytechnique Fédérale de Lausanne (EPFL) And Stanford University	2014	Machine learning	Xilinx and Intel FPGAs	DSL embedded in Scala	Retargetable (HLS C/C++, CUDA, OpenCL)	Vivado HLS, Intel OpenCL Compiler	Delite compiler
PolyMage [93]	Indian Institute of Science	2015	Image processing	Intel Xeon E5-2680	DSL embedded in Python	OpenMP annotated C++ code		Intel C++ Compiler is used to decide which loops to vectorise and what way
Halide [94]	MIT	2013	Image processing	Xilinx FPGAs	C++-embedded DSL	HLS C++	Vivado HLS	FROST [82] to generate HLS code
Darkroom [95]	Stanford University	2014	Image processing	ASIC, Xilinx FPGA	Terra-embedded [96] DSL	Genesis2 HLS	Synopsys Synplify G-2012.09-SP1 and Xilinx Vivado 2013.3	Terra compiler [96]
HIPA^{cc} [97]	Friedrich-Alexander-Universität Erlangen-Nürnberg	2015	Image processing	Xilinx and Intel FPGAs	DSL embedded in C++	HLS C++, OpenCL	Vivado HLS, Intel OpenCL Compiler	Clang/LLVM to convert DSL code to AST (IR)
Tiramisu [88]	MIT	2018	Data parallel applications	Heterogeneous platforms with Xilinx FPGAs	C++	LLVM for multi-processor, CUDA for GPU, HLS C for FPGAs	Vivado HLS	FROST [82] to generate HLS code
S2FA [98]	University of California, LA	2018	Cluster applications	Xilinx FPGAs	Java bytecode generated from Scala source code	HLS C	Vivado HLS	Merlin Compiler [80] for HLS code generation
TVM [99]	University of Washington	2018	Deep learning	CPUs and GPUs	Tensor expression language (DSL for TVM) based on Python	CUDA/OpenCL		Based on Halide [94]
Rigel [100]	Stanford University	2016	Image processing	FPGA	Lua-embedded DSL [101]	Verilog	Xilinx ISE 14.5	Lua compiler [101]
RIPL	Heriot-Watt and Sheffield Hallam Universities	2014	Image processing	FPGAs	RIPL (standalone language - not embedded)	RTL	Vivado HLS	

respectively. These all use the Delite compiler as their back-end to transform input DSL to an IR, explore and perform possible optimisations, and generate HLS-friendly parallel code for hardware synthesis. The Delite compiler, which was originally developed for heterogeneous platforms including CPUs and GPUs, has been modified by various research teams so it can target FPGAs through Xilinx Vivado HLS and also other HLS tools such as the Maxeler MaxCompiler [12] and vMagic [105].

2) FPGA HLS BASED ON THE PolyMage DSL

PolyMage [93] is a complete framework including a new Python embedded DSL and a model-driven compiler (an optimiser and an autotuner) that can implement high-performance image processing pipelines described in the PolyMage DSL onto reconfigurable hardware. While the image processing pipeline must be in PolyMage DSL, the rest of the application software can be expressed in other high-level languages.

The user develops code in PolyMage DSL (the ''manual coding'' in path ②ⓒ in FIGURE 1) that describes pipelines as directed acyclic graphs (DAGs). In these graphs each pipeline stage is mapped to a processing node and the dependencies among the stages are represented by the vertices. The PolyMage compiler (the ''DSL compiler'' in path ②ⓒ in FIGURE 1) takes the pipeline specification from the DSL and automatically extracts the ''pipeline graph'' (the ''intermediate representation'' in path ②ⓒ in FIGURE 1). The compiler also checks static bounds, inconsistencies and invalid accesses, and also inlines nested functions wherever necessary.

The next step carried out by the compiler is running a model-driven heuristic to break the pipeline into a number of smaller groups of stages. The process continues with optimisation via the overlapped tiling technique for a set of heterogeneous functions (pipeline stages) and constructing schedules for overlapped tiles. The PolyMage compiler then iteratively merges groups until no further merging is possible. Generating the HLS code (in C++ augmented with directives/pragmas) and auto-tuning are the final steps (corresponding to the ''synthesisable code generator'' in FIGURE 1).

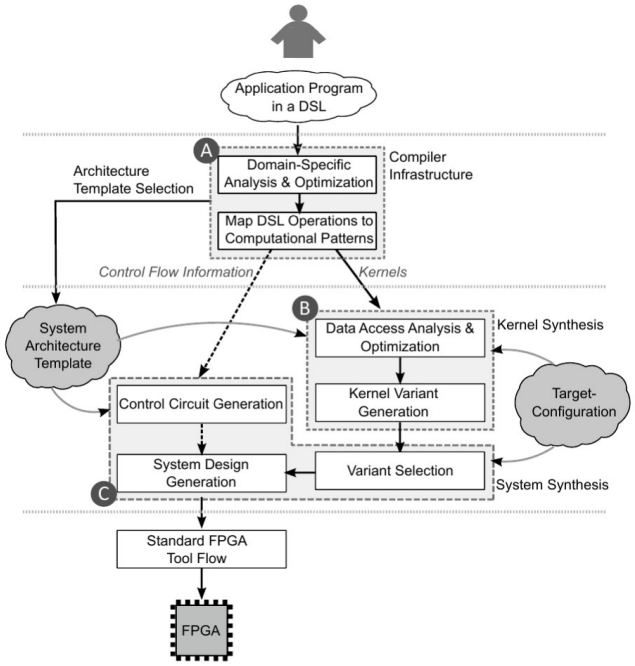


FIGURE 11. Overview of OptiML-based HLS methodology [92].

FIGURE 12. Workflow of PolyMage-HLS for FPGA [107].

Using the PolyMage compiler, researchers in [106], [107], introduce PolyMage-HLS, a technique to efficiently realise image processing pipelines, already described in the PolyMage DSL, onto FPGAs. As shown in FIGURE 12, PolyMage-HLS can be considered as an FPGA backend that translates an input that expresses image processing stages into an equivalent HLS C++ code that can be synthesised on FPGAs using conventional HLS suites (the original PolyMage compiler [93] produces only OpenMP C++ outputs). It should be noted that the code generator, as part of the new PolyMage DSL compiler, automatically annotates (restructures) the output C++ code with appropriate pragmas to make it an optimised HLS code that can generate an efficient HDL.

The HeteroCL programming platform is anther, more recent, Python-embedded DSL that captures the design at a very high-level of abstraction and hence, separates the processing algorithm from hardware-related concerns, including compute/data customisation and memory architectures [108].

3) FPGA HLS BASED ON HALIDE DSL

Halide [94] is a DSL specifically designed for high-performance image and array processing code for a wide variety of CPU families and operating systems, as well as GPUs capable of running APIs in CUDA, OpenCL, OpenGL, OpenGL Compute Shaders, Apple Metal and Microsoft Direct X 12. Like many other DSLs, which are based on conventional software programming languages, Halide is actually an internal DSL in C++ such that the system developer can use Halide's C++ API to express the pipelined structures needed for effective implementation of image-processing algorithms.

The main contribution of the Halide workflow is that the implementation of the algorithm is separated from the algorithm's resource assignment schedule. As a result, any modification in the execution schedule does not necessarily change the computing algorithm. This helps application designers experiment with different scheduling schemes such as loop nesting, parallelisation, loop unrolling and vector instructions to tune the scheduling for maximum performance.

The Halide DSL and its compiler (the "DSL compiler" in path ②ⓒ in FIGURE 1) have recently been upgraded to support HLS code generation for FPGA acceleration [109], [110]. System developers can now capture the hardware functionality at a very high level of abstraction, and can also generate the complete software application including the sequential part of the code that runs on the CPU, and which is responsible for communicating with the functions offloaded onto the FPGA. This workflow is illustrated in FIGURE 13.

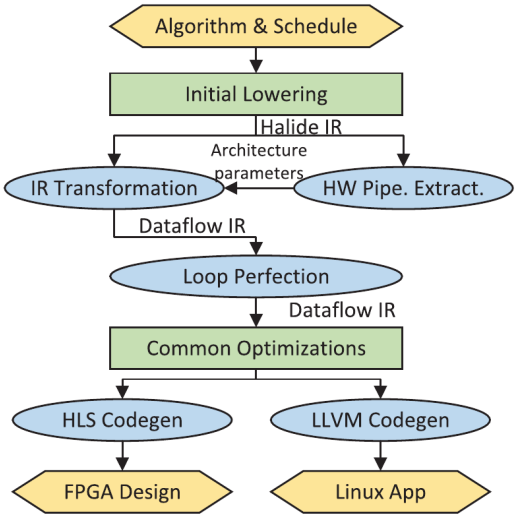


FIGURE 13. Halide DSL compilation flow for FPGA HLS [110]. Blue blocks are new, green blocks are unchanged/existing Halide [94] compilation steps.

In the Halide DSL workflow for FPGA HLS, once a few optimisations are performed, the final IR of the pipeline is simultaneously processed by two code generator tools, HLS and LLVM. While the former (the "synthesisable code generator" in FIGURE 1) translates the FPGA-friendly functions into HLS-C++ code, the latter converts the rest of the IR into a C++ wrapper. Wherever applicable, using a synthesisable C++ template library, the HLS code generator also automatically incorporates HLS pragmas to implement loop pipelining and array partitioning into the final design realised on the FPGA. The final bitstream generation and

FPGA programming are carried out using third-party HLS tools such as Xilinx Vivado HLS (the ''behavioural HLS tool'' in FIGURE 1).

GENESIS [111] is another work that uses Halide with a new DSL source-to-source compiler for FPGA implementation (the ''synthesisable code generator'' in FIGURE 1). By analysing and transforming the input code (Halide's output), this FPGA backend is able to generate highly optimised HLS C++ code that can eventually be realised as an efficient hardware accelerator running a specific image processing algorithm on the target FPGA by using the Xilinx Vivado HLS compiler [112]. The workflow is illustrated in FIGURE 14.

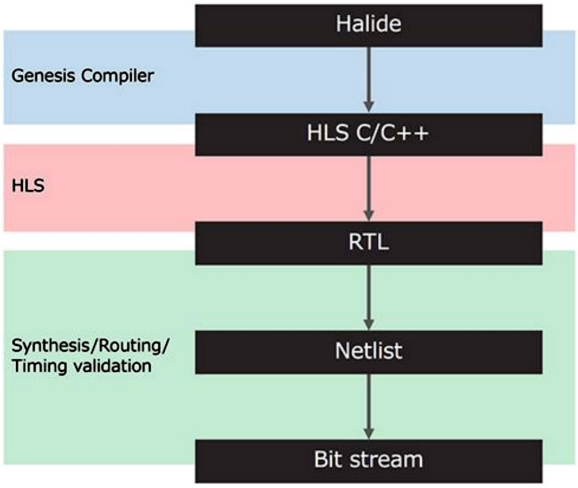


FIGURE 14. FPGA HLS workflow through combination of Halide and GENESIS [112].

In conventional domain-specific hardware design workflows, the optimised implementation can be achieved by heuristic search through the design space but this requires lots of manual coding. It is claimed the GENESIS compiler can reduce this coding effort by controlling different performance-affecting factors just by scheduling the functions in Halide generated outputs.

In addition to major works exploiting the Halide compiler within their toolchain, there are a number of projects such as [113]–[115] that follow the same workflow but with either extended versions of the Halide compiler or Halide-inspired DSL compilers to support their own domain-specific structures.

4) DARKROOM DSL

Darkroom [95] is a Terra-embedded [96] DSL that can capture image processing algorithms as DAGs of basic image processing operations. By restricting image operations to fixed-size windows of pixels, Darkroom can generate very efficient hardware accelerators on the target FPGA that benefit from the line-buffering technique in which intermediate data passing between pipeline stages is saved in on-chip buffers.

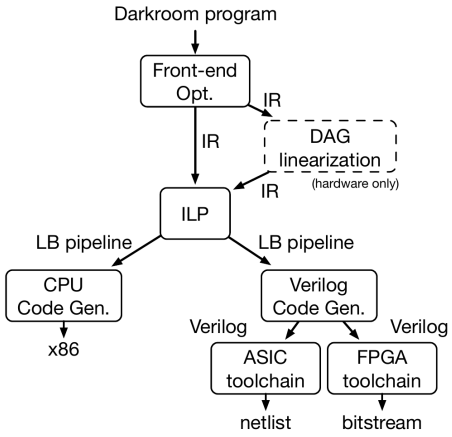


FIGURE 15. Overview of Darkroom DSL workflow [95].

After the system designer describes the application in Darkroom DSL, the high-level code is transformed into a DAG (the ''intermediate representation'' in path ②ⓐ in FIGURE 1) of high-level image processing operations (or the Darkroom IR) by the Darkroom compiler (the ''DSL compiler'' in path ②ⓐ in FIGURE 1). Then an integer linear programming (ILP) solver analyses and optimises the design's IR and converts it into a simple and optimised line-buffered pipeline that processes one input pixel at a time and as a result, generates one pixel of output at a time. In the next step, the ILP module (the ''synthesisable code generator'' in FIGURE 1) represents the candidate pipelines for hardware acceleration in Genesis2 [116], a Verilog metaprogramming language (the ''synthesisable behavioural code'' in FIGURE 1), that can be later translated into RTL Verilog using a code generator (the ''behavioural HLS tool'' in FIGURE 1).

A Halide inspired DSL language, Rigel [100], is another framework for implementing optimised image processing accelerators on FPGAs, and is based on the Darkroom framework. It can be understood as an extension to Halide that supports more advanced kernels as well as static and dynamic scheduling.

5) HIPA^{cc} DSL

Heterogeneous Image Processing Acceleration (HIPA^{cc}) framework [97] is shown in FIGURE 16. It is a DSL and source-to-source compiler that supports C/C++, CUDA, OpenCL, Renderscript, and HLS-friendly C/C++, which is able to produce low-level code for image processing kernels on a wide range of GPUs, CPUs and FPGAs.

Like other DSLs, image processing algorithms are captured in the frontend as DSL code using embedded C++ template classes predefined in HIPA^{cc}. When the code is supplied to the HIPA^{cc} workflow, the image processing structures are detected and translated into an AST using then Clang/LLVM compiler (the ''DSL compiler'' in path ②ⓐ in FIGURE 1). In the next step, HIPA^{cc} parses the AST detect

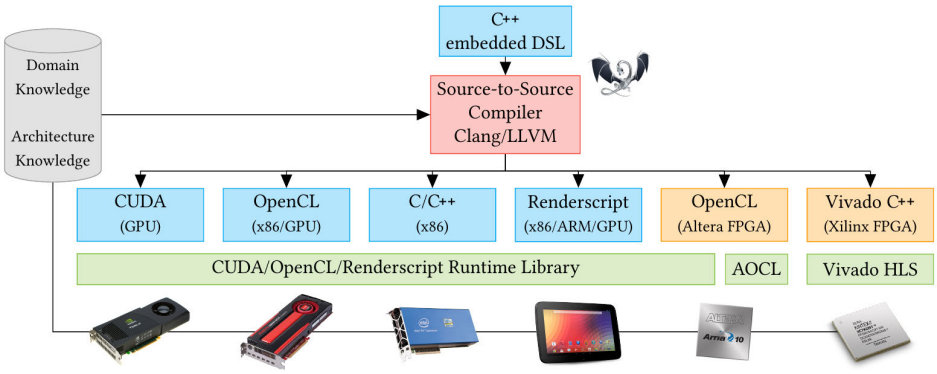


FIGURE 16. HIPA^{cc} framework and its target architectures [97].

data dependencies and hence, to form the internal HIPA^{cc} IR and construct the application's dependency graph (the ''intermediate representation'' in path (2c) in FIGURE 1). The workflow can apply common transformations on the AST followed by vendor-specific AST transformations (Xilinx or Intel) for high performance implementation of the image processing algorithms as FPGA accelerators. The workflow then translates the AST representation into either HLS C++ code (for Xilinx FPGAs) or OpenCL (for Intel FPGAs) using the pretty printer from Clang (the ''synthesisable code generator'' in FIGURE 1) and finishes with generating FPGA synthesisable HDL followed by using the appropriate HLS tool to generate the bitstream and program the FPGA.

IV. DATAFLOW APPROACH FOR FPGA SYNTHESIS

The semantic gap between sequential HLL code and its parallel dataflow representation as FSMs often leads developers to manually optimise hardware accelerators, or to use a restricted subset of HLL constructs. They have to manually insert platform-specific annotations, and use particular libraries to optimise the hardware. These ad-hoc approaches imposed by the target platform lead to a fundamental change in the application design and require developers to have expertise in both hardware and HLL design.

Dataflow-based design, as denoted by label (3) in FIGURE 1 on Page 174694, allows developers to define an application in a different way. Dataflow programming arose from research into parallel computing in the 1970s [117], [118], however it has only emerged as a suitable alternative for configuring FPGAs in the last decade. In this approach, an application is represented as a directed graph, known as a *dataflow graph* (DFG), of computational units with edges defining communication channels between them to transmit atomic pieces of data known as *tokens*. As soon as the inputs of a unit arrive, the unit executes, and its output is forwarded to the next computational units in the flow. Unlike, von-Neumann machines [119], there is no global instruction load or store as each computational unit contains all its relevant instructions. The concurrent execution of computational units is a good match with the parallel execution of FPGAs.

DFG representations for FPGA implementation can be defined in two levels of abstractions. At the lower level they are represented using meta-languages, which can be transformed to RTL using *Dataflow HLS tools*. This flow is denoted as (3a) in FIGURE 1. DFGs can also be defined at a higher level of abstraction using C or C-originated code. They are translated to low-level DFG representations using *synthesisable DFG generators* as in design flow (3b). We will discuss these two approaches in the following sub-sections.

A. DATAFLOW HLS (3A)

In dataflow HLS, the application is expressed using a meta-language such as MaxJ [120] or CAL Actor Language (CAL) [13], as a DFG with computation nodes and communication channels between them. Then, using dataflow HLS tools (e.g. MaxCompiler [12], Exelixi [121]), the hardware specification is transformed to an RTL description. This approach simplifies the translation to actual hardware, as the major task is optimisation. It also makes it easier to estimate resource utilisation and performance directly from the DFG specification. TABLE 5 provides a list of currently available dataflow HLS tools, which will be discussed in the following sections.

1) MaxCompiler

MaxCompiler [12], [128] is a compilation tool from Maxeler Technologies for their proprietary FPGA-based dataflow engines (DFEs) [129]. The DFEs are connected to a processor via PCIe and to other DFEs via high-bandwidth interconnects. The host application is written in C, Python or Fortran to run on the processor and to manage accelerators. The compute-intensive portions of the application are written for DFEs using a Java meta-language named MaxJ (the ''synthesisable DFG representation'' in path (3a) in FIGURE 1). MaxJ has been recently standardised as OpenSPL (Open Spatial Programming Language) by the OpenSPL Consortium [122].

Each DFE configuration contains one or more *computation kernels* (i.e. accelerators) for implementing arithmetic and logical computations and a single *manager configuration* to orchestrate global dataflow between kernels and external I/O.

TABLE 5. Dataflow HLS tools.

Tool	Provider	Open source	Intro	Update	Platform	Input	Output	Domain
MaxCompiler [12]	Maxeler Technologies	No	2010	2018	FPGAs	MaxJ [12], OpenSPL [122]	VHDL	All
CAPH [123], [124]	University of Clermont Auvergne	Yes	2011	2018	FPGAs	CAPH network language	VHDL	Stream processing
OpenDF [125]	Xilinx	Yes	2007	2013	Xilinx FPGAs	CAL [13]	Verilog	Multimedia prcessing
Orcc [126] + Exelixi [121]	Ecole Polytechnique Federale de Lausad	Yes	2016	2018	FPGAs	RVC-CAL [127]	Verilog	Multimedia processing

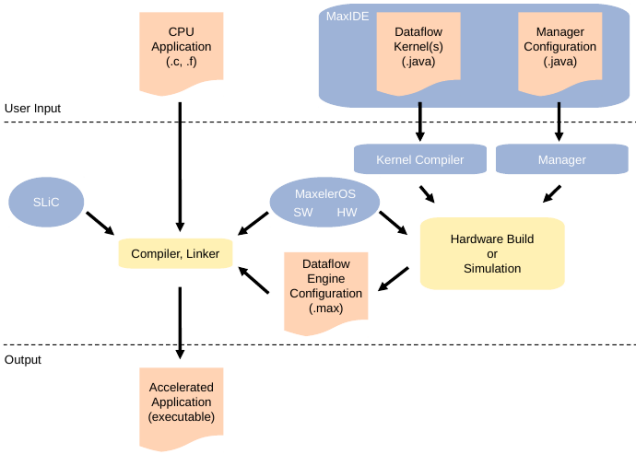


FIGURE 17. MaxCompiler component interactions [120].

Dividing an application into kernels and a manager enables logic to be deeply pipelined without control flow hazards. FIGURE 17 illustrates the flow and components of the MaxCompiler. MaxCompiler transforms the kernel code into an IR, which is manipulated and optimised by simplifying and flattening the code, expanding objects and converting dynamically determined loops into static loops. This intermediate form is FPGA neutral. The compiler then maps the kernel directly to a specific FPGA using the vendor-specific backend and produces an RTL description. A DFE configuration file (`.max` file) is finally generated (not shown in FIGURE 1, as it is a specific step for MaxCompiler), which contains the configuration bitstream generated from the RTL and other data used to access the accelerator at runtime. This configuration file can be accessed by the processor via an automatically generated *SLiC* (Simple Live CPU) interface. A Linux-based runtime called *MaxelerOS* is used to manage the DFEs. The compiler also generates cycle-accurate simulation models of the kernels being compiled.

2) CAPH

CAPH [123], [124] is a domain-specific toolchain for specification and implementation of stream processing applications on FPGAs that follows the "dataflow HLS tool" path (3a) of FIGURE 1. It relies upon the actor/dataflow model of computation. Applications are specified as networks of purely dataflow actors exchanging tokens through

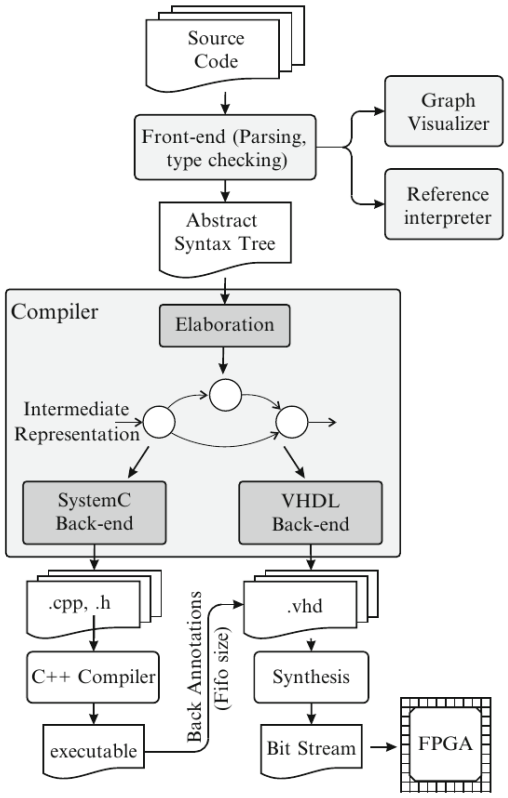


FIGURE 18. CAPH toolchain [123].

unidirectional channels using a high-level, polymorphic functional language. The behavior of each actor is defined as a set of transition rules using pattern matching. The CAPH toolchain, as shown in FIGURE 18, provides a *graph visualiser* to represent the actor network, a reference interpreter and a compiler producing both SystemC and synthesisable VHDL code. The *reference interpreter* provides reference results to check the correctness of the generated SystemC and VHDL code. It is also used to test and debug programs during initial stages of application development. The *compiler* turns the AST into target-independent IR where each process is represented as a FSM and channels as unbounded FIFOs. Two backends produce cycle-accurate SystemC code for simulation and profiling, and VHDL code for FPGA synthesis. SystemC execution provides information to refine the VHDL implementation e.g. to define the sizes of the FIFOs used to implement channels.

3) OpenDF

Open DataFlow (OpenDF) [125], [130] is an open-source initiative of Xilinx. In the OpenDF environment, an algorithm is defined as a DFG with edges representing the data flow through lossless directed FIFO channels. The accelerators are called actors (the ''synthesisable DFG representation'' in path ③ⓐ in FIGURE 1) and are written using CAL Actor Language [13]. Each actor contains one or more actions, each defining a firing rule of the actor. An action accepts tokens at its input ports, modifies the state of the actor, and produces tokens at the actor's output ports. A CAL-to-HDL converter (the ''dataflow HLS tool'' in path ③ⓐ) first transforms the actor to an XML Language Independent Model (XLIM), which is then transformed into a static single assignment (SSA) form to indicate explicit data dependencies between parts of the program. Next, in the synthesis stage, SSA threads are translated into circuits of basic operators (e.g. arithmetic, logic, flow control and memory accesses) with directives to specify loop unrolling and register insertion for improving the maximum clock rate. Finally the CAL-to-HDL converter produces a Verilog RTL to specify the hardware implementation of the actor.

4) ORCC AND EXELIXI

RVC-CAL (RVC for Reconfigurable Video Coding) is a subset of CAL standardised by ISO-MPEG for describing video coding specifications [127]. To support RVC-CAL dataflow applications, a new open-source framework called the Open RVC-CAL Compiler (Orcc) was introduced in [126]. As illustrated in FIGURE 19, frontend of the Orcc toolchain translates RVC-CAL into an AST and then to an IR. The frontend performs semantic validation, type inference and expression evaluation. The Orcc transformation provides a type-accurate simulation of an RVC-CAL program by interpreting its IR. Exelixi [121], which is a successor of Xronos [131], works as a middle and backend of the toolchain. It performs a series of transformations and optimisations on the Orcc IR, and converts it to a HLS-C function HLS stream input and output interfaces. Each RVC-CAL action is turned into a static function, and state variables are also declared as static. Next, a HLS tool consumes this code to generate a synthesisable Verilog representation of the actor. Exelixi also produces C++ code for the processor and necessary configuration for host-accelerator interfaces.

A tool for generating C-based HLS descriptions from a RVC-CAL dataflow description has recently been described in [132]. This work proposes a new interface synthesis approach by using a shared memory that behaves as a circular buffer. Simulation results for one application show this approach increased throughput by 5.2 times and reduced latency by a 3.8 times over a state-of-the-art implementation.

B. HIGH-LEVEL DFG REPRESENTATION ③ⓑ

Some behavioural HLS tools (e.g. Vivado HLS and Intel HLS) provide a mechanism to represent dataflow computing at the task-level using specific directives. This approach achieves some goals of dataflow HLS, without requiring a DSL, but the developers need to have sufficient knowledge to guide the synthesis process through those directives and annotations to achieve an efficient design. On the other hand, although dataflow HLS compilers (e.g. MaxCompiler, Orcc and Exelixi) allow developers to specify DFG applications for FPGAs, they have to use a meta-language such as MaxJ or CAL. The use of these languages is sometimes tedious and error-prone.

To take the advantage of dataflow approach, there is some work underway to write DFG applications at a higher level of abstraction, using C and Scala-based languages. These are mostly DSLs (e.g. Spatial [133]), with some exceptions that are not explicitly described as 'domain-specific' e.g. OXiGen [134]. In either case, they use a HLL to describe dataflow applications, and then use dataflow HLS tools as discussed in the previous sub-section. TABLE 6 gives a list of high-level DFG compilers.

1) SPATIAL

Spatial [133], [140] is a DSL and an open-source compiler developed at Stanford University. It is based on Delite Hardware Definition Language (DHDL) [135]. In contrast to MaxJ [12], which is a lower-level representation to configure Maxeler devices, Spatial is a Scala-based high-level language to define hardware accelerators as a hierarchical DFG (the ''high-level DFG representation'' in path ③ⓑ). Nodes in this DFG represent control structures, data operations and memory allocations, while edges represent data and effect dependencies. The accelerator code is embedded into the host program using specific constructs. The language provides a set of control structures to allow the compiler to identify parallelism opportunities. Users can exploit the memory hierarchy through a library of on-chip and off-chip memory templates. The Spatial compiler (the ''synthesisable DFG generator'' in path ③ⓑ) generates an IR, which goes through a number of passes including controller scheduling, memory analysis, design space exploration, pipelining and unrolling. In the final pass, the code generator works on the modified IR (the ''synthesisable DFG representation'' in FIGURE 1) to instantiate hardware modules from a library of custom and parameterised RTL templates written in Chisel RTL [136]. The generated RTL can be synthesised on a subset of Xilinx and Intel (Altera) FPGAs. The compiler also generates a C++ code for the host CPU to control the FPGA accelerator.

2) OXiGen

OXiGen [134] is a very recent hardware compilation tool that takes a compute-intensive C function and translates it to a dataflow representation for the MaxCompiler (the ''dataflow HLS tool'' in FIGURE 1). The C function is first transformed into a LLVM-IR at the frontend. The IR passes through a series of standard LLVM analysis and vectorisation optimisations. Vectorisation allows the user to specify the size of input and output vectors, and changes the data types of the

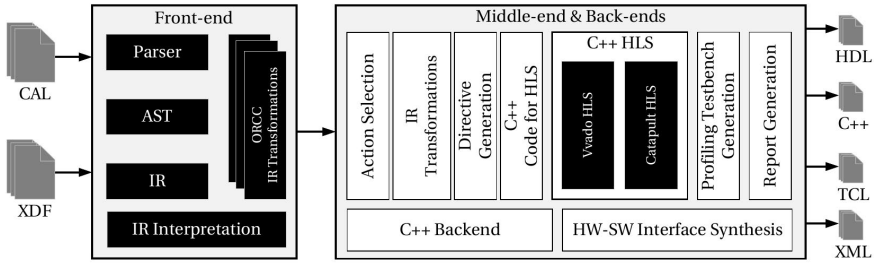


FIGURE 19. Orcc and Exelixi compilation flow [121].

TABLE 6. High-level DFG compilers.

Tool	Provider	Open source	Intro.	Update	Pedigree	Platform	Input	Output	Target HLS	Domain
Spatial [133]	Stanford University	Yes	2018	2019	Successor to Delite [135]	Specific FPGAs and CGRAs	Spatial language	Chisel RTL [136]		All
OXiGen [134]	Politecnico di Milano	No	2018	NI		FPGAs	C	MaxJ	MaxCompiler	All
RIPL [137]	Politecnico di Milano	Yes	2015	2018		FPGAs	RIPL DSL	CAL	Exelixi	Image processing
FAST [138] - LARA [139]	Imperial College London	Yes	2013	NI		FPGAs	C + LARA (aspects)	MaxJ	MaxCompiler	Aspect-oriented programming

NI = No Information found

C function to vector types. It permits the MaxCompiler to improve parallelism and utilise available bandwidth as the hardware resources for each vector element are replicated and they are processed in parallel. The IR is then translated into a custom DFG representation. The tool applies *loop rerolling optimisation* to control the amount of hardware replication for computations within nested loops. The DFG is eventually translated into a MaxJ kernel and its MaxJ manager (the "synthesisable DFG representation" in FIGURE 1). OXiGen still does not support recursive functions and loop-carried dependencies inside the C function.

3) RIPL DSL

The Rathlin Image Processing Language or RIPL is a stand-alone high-level DSL for developing memory-efficient image processing applications [137]. Image processing algorithms developed in RIPL can share data between two image data pipelines. The RIPL DSL is designed based on higher-order algorithmic skeletons [141] (basic sets of computing patterns) in the image processing domain, and it can clearly and unambiguously capture highly regular image processing stencil computations, as well as recursive functions with irregular access patterns to nonlocal memory locations. When processing infinite image streams, the RIPL workflow generates pipelines automatically since parallel FPGA circuits are inherently represented by independent computational blocks of the RIPL dataflow representation.

The RIPL compiler (the "DSL compiler" in path ②c in FIGURE 1) takes the input program in RIPL language and converts it into dataflow process networks (DPNs) [142], an IR for image processing computations. Optimisations are performed on this IR before it is transformed to CAL [13] (the "synthesisable HLL code" in FIGURE 1). In the next step, the CAL representation is translated into synthesisable Verilog using Exelixi [121] (the "behavioural HLS tool" in FIGURE 1). Finally, Xilinx Vivado is used to convert the HDL representation into a bitfile to program the FPGA.

4) FAST-LARA

FAST (Facile Aspect-driven Source Transformation) [138] is a C-based DSL to describe compute-intensive parts of an application as high-level dataflow representations. FAST functions (the "high-level DFG representation" in path ③b) can be embedded within a C application, and are invoked via specific annotations to indicate alternate hardware implementation. The FAST compiler generates interconnected functional kernels that perform computation asynchronously as soon as all the inputs are available. FAST is combined with the LARA aspect-oriented design flow [139], as shown in FIGURE 20. LARA allows non-functional concerns such as optimisation and transformation strategies to be developed and maintained independently from the original application source code. These are described using LARA *apects*, which can developed independently from the application and reused for different applications, thus improving productivity. There are four types of aspects: *system aspects* capture transformations and optimisation strategies that affect the whole application, such as hardware/software partitioning and run-time reconfiguration; *architectural aspects* focus on low-level design optimisations to improve timing, resource usage or exploit specialised architectural features; *exploration aspects* deal with strategies to generate multiple designs to find an optimal implementation based on user-specified constraints; and *development aspects* that relate to concerns that have an impact on the design process, such as debugging, kernel simulation and improving compilation speed. LARA manipulates

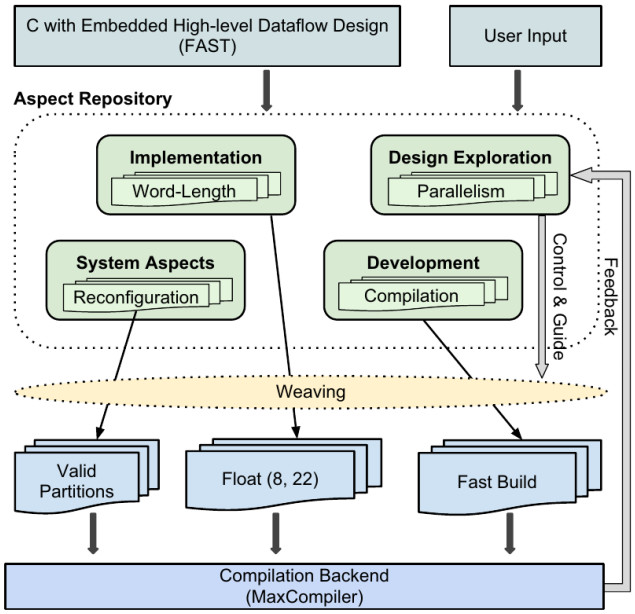


FIGURE 20. FAST-LARA design flow [138].

and transforms the FAST description based on these aspects, and generates configurations (the "synthesisable DFG representation" in FIGURE 1) that can be implemented on FPGAs using the MaxCompiler [12].

V. SPECIFICATION OF AUTOMATIC CODE DEPLOYMENT TOOLS

Manual HLS tools will eventually evolve into toolchains supporting the Automatic Code Deployment (ACD), however, as observed in the previous sections, this goal is still some way off as generation of efficient HLS-compatible code from CPU-friendly code still requires substantial manual intervention. The primary reason for this is the huge gap between the sequential, shared-memory execution paradigm of CPUs and the parallel, distributed-memory paradigm of FPGAs.

In this section, we first elaborate the minimal requirements to build an ACD toolchain and discuss how far the available tools are from this suggested model. We then identify a number of measures that can possibly facilitate this transition. This is followed by a survey of research to either automate some stages of HLS or introduce algorithms for systematically producing HLS-friendly high-level code with improved performance once implemented in FPGA.

A. AUTOMATIC CODE DEPLOYMENT IN DETAIL

To automate the process of HLL code implementation on FPGA-based heterogeneous platforms, the following major tasks are required:

- Automatic identification of compute-intensive *hotspot* functions
- Automatic code refactoring to make non-synthesisable code synthesisable
- Automatic transformation of the source code to optimise hardware accelerator performance. This may involve automatic insertion of vendor and platform-specific annotations for parallelism and improved performance

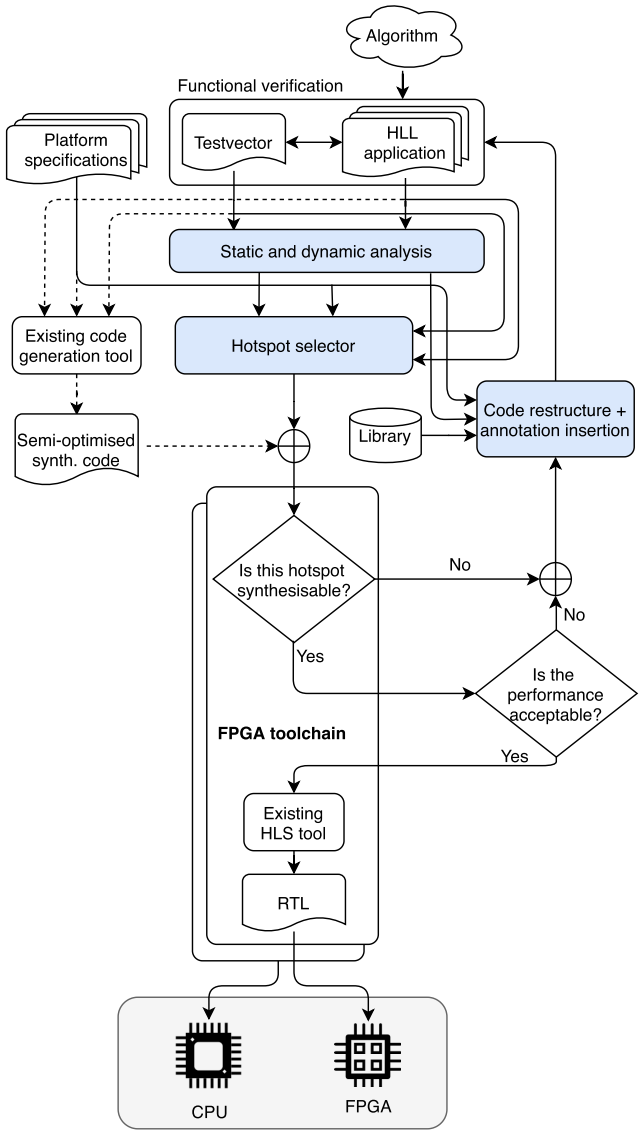


FIGURE 21. Design flow of the ACD toolchain.

To perform these tasks, we imagine a future ACD toolchain as illustrated in FIGURE 21. This would leverage existing open source and commercial tools in different stages of its flow. The blue blocks in the design flow indicate where the most research and development effort would be required, and the dotted lines indicate the optional flow in the toolchain.

1) STATIC AND DYNAMIC ANALYSIS

To identify functions (or code blocks) for offloading to FPGAs, it is necessary to first profile the application using appropriate datasets to analyse the application not only in terms of execution time but also other important metrics including memory bandwidth and power consumption per function. To identify data access patterns,

data dependencies and memory bandwidths required by the functions, the extracted information would be represented in various forms including:

- the Function Call Graph (FCG);
- the data dependency graphs at both inter- and intra-functional levels; and
- the CDFGs at both inter- and intra-functional levels.

2) HOTSPOT SELECTION

The ACD toolchain would interpret static and dynamic analysis results to identify the possible parallelism opportunities, data structures stored in memories, data dependencies and memory bandwidths required by the functions. This would involve consideration of aspects including:

- data types
- data dependencies
- data access patterns
- structure of data stored in the memory
- memory bandwidths required by code segments

It would help the toolchain to automatically select hotspots for hardware offloading. To reduce the time required to find an optimised solution, mathematical models might be developed for rapid pre-synthesis hardware performance estimates.

3) CODE RE-STRUCTURING AND ANNOTATION INSERTION

The ACD toolchain would perform source-to-source transformation on the identified hotspots and insert appropriate vendor and platform-specific annotations to generate optimised and synthesisable HLS code. This might involve tasks including:

- selecting HLS-friendly algorithms for some functions;
- inlining functions;
- partitioning functions into smaller sub-functions; and
- refactoring the code to expose parallelism or to reduce the communication overhead between parts of the application running on different computing platforms.

To accomplish this task, the tool would need to draw on detailed information obtained from the profiling step, and to address a number of challenges including:

- non-synthesisable data types
- recursive function calls
- OS and domain-specific function calls
- pointers with multiple levels of indirection
- multi-threading
- dynamic memory allocation and dynamic data types
- I/O interfacing

The tool would also need to annotate the code with directives and pragmas to guide subsequent stages in the toolchain including to specify optimisation steps, partitioning, and communication between the parts of the heterogeneous system.

B. SURVEY OF ATTEMPTS TO AUTOMATE FPGA-HLS

Although it may be some years before the vision of a complete ACD toolchain is realised, the literature does contain a considerable amount of research devoted to automating some of the challenging, tedious and time consuming steps of the code deployment process. The remainder of this section provides a survey of this work.

1) ACD FOR FORTRAN APPLICATIONS

Very recent work in [6], [7] tries to develop a compiler that can automatically transform legacy scientific applications written in FORTRAN into OpenCL programs with optimised kernels that can be deployed on FPGAs as hardware accelerators. What makes this toolflow special is that with no pragmas or human intervention, the source-to-source compiler can automatically refactor sequential and single-threaded FORTRAN 77 code a into a highly optimised OpenCL program with parallelised kernels ready for FPGA implementation.

While the presented work is able to successfully refactor the program such that all its functions can be offloaded onto the FPGA with minimised CPU-FPGA data bandwidth, it can be used only for a very limited number of software applications developed in FORTRAN.

2) LOOP PIPELINING

Loop pipelining is extensively employed as a HLS optimisation since it can improve the performance of the FPGA hardware by taking advantage of the intrinsic parallelism between successive loop iterations.

In [143], a technique is introduced that makes it possible to apply loop pipelining to loops accommodating variables with uncertain values at compile time. A series of extensions to this work [144]–[146] considers cases in which the HLS tool identifies loops with uncertain or nonuniform memory dependencies. For these a side-controller is generated using parametric polyhedral analysis to dynamically change the loop iteration speed at runtime such that these dependencies are not violated. As a result, the synthesised pipeline can be statically scheduled like any conventional pipeline.

In an attempt to improve the performance of HLS-based designs, [147], [148] introduce a combined dynamic and static technique to optimise irregular loop nests (such as loops with dynamic-bound inner loops, access to less-regular data structures and data-dependent workload, irregular memory access patterns and irregular data dependencies) through adaptive loop pipelining. Using runtime optimisation, the synthesised pipeline is equipped with extra hardware for dynamic scheduling to adapt to the program's irregular data-dependent behaviour. The paper [149] presents a methodology to automatically generate dynamically scheduled circuits from C code. It adopts several ideas from the asynchronous domain, but produces reasonably synchronous designs comparable to standard HLS techniques. The authors of the paper claim to achieve large performance improvements with the investment of more resources in comparison with static scheduling done in Vivado HLS.

One of the main challenges in synthesising high-performance loop pipelines via HLS is to implement parallel

accesses to the memory in nested loops. To provide parallel data access for the candidate nested loops, an efficient and low overhead algorithm is presented in [150] that partitions the memory via caching the reusable data into on-chip registers. The SPINE compiler [151] automatically generates FPGA accelerators based on the algorithmic species theory [152]. This approach separates the loop-nest structure from the operations to be executed, and then post-processes the synthesised loop-nest (available in HDL), automatically applying hardware specific optimisations for efficient FPGA implementation of these loop-nests. A similar approach is also proposed in [153].

3) AUTOMATED DESIGN SPACE EXPLORATION

FPGA accelerator designs make extensive use of design options specified at the architecture level of abstraction as pragmas. These include pragmas to guide loop unrolling, loop pipelining and array partitioning. However, since pragmas affect the design's performance and cost in complex ways, exhaustively exploring all possible alternatives is very time consuming. For real-world designs, with time-to-market pressure, this kind of optimisation is likely to be infeasible.

Lin-Analyzer [154] is an accurate FPGA accelerator performance analyser developed to address the design space exploration challenge at a high-level of abstraction using pragmas for loop unrolling, pipelining and array partitioning. Lin-Analyzer is a pre-HLS tool: it does not need to generate RTL, and hence performs its analysis/exploration very rapidly without invoking HLS tools. Instead, it gathers information about operation dependencies from dynamic profiling to accurately predict the performance of the final hardware function.

COMBA is another pre-HLS design space explorer [155], which covers seven pragmas: loop unrolling, loop pipelining, array partitioning, function pipelining, dataflow, loop flattening and function inlining. Without invoking an HLS tool, COMBA's estimate of C/C++ applications performance is claimed to be very accurate (average error $\approx 1\%$) and quick (a few minutes for real-life applications).

The XPPE [11] is a recent machine-learning-based design space explorer that estimates the performance for FPGA-HLS. A correlation between estimated and actual speed-up of more than 0.97 is claimed. XPPE is a 3-layer fully connected neural network with 900 hidden neurons that estimates the speedup of an application once implemented on an FPGA using HLS. The neural network receives the resources utilised on the target FPGA (reported by the HLS tool), factory-made resources on the target FPGA and domain-based application characteristics (such as Machine Learning, Image/Video Processing, Cryptography and Mathematical) as input to estimate the speedup for the target FPGA compared to a CPU implementation.

Researchers in [156] introduce FLASH, a post-HLS scheduling-based simulator for FPGA accelerators, that is designed to bridge the accuracy/speed gap between HLS-based simulation and time-consuming RTL-based simulation of the design, especially for system designers from a non-FPGA background. Upon receiving the HLS output, FLASH extracts the application's scheduling information and forms a cycle-accurate simulation flow with no allocation/binding information, while maintaining the C semantics. This results in a much faster simulation compared to the RTL counterpart.

Researchers in [157] present HLScope+ (an improved version of HLScope [158]), a performance estimator/debugger, based on a source-to-source transformation for FPGA-HLS. It takes advantage of a high-level DRAM access model for a typical FPGA architecture and a HLS-specific automatic code instrumentation technique targeting high-level cycle estimation via software simulation. Typical HLS tools only provide a high-level report of the synthesised design that is not very useful for identifying regions in the HDL where performance needs to be improved. HLScope+ fills that gap by providing module execution times according to a simulation-based analysis, without the need for RTL. This augments the reports generated by Vivado HLS to help the developer to more efficiently explore the design. This analyser also reports on the sources of a wide range of different stalls using its stall analysis network that monitors the on-board execution of dataflow modules.

The FlexCL [159] is a quick and accurate performance and power estimator for OpenCL kernels offloaded onto FPGAs via HLS. It estimates performance by running a model for global memory access alongside computation models for processing components. It estimates the static power consumption of the FPGA device and global memory. For dynamic power, FlexCL breaks the kernel execution into hierarchical phases and applies a CDFG model to every level in the hierarchy. It uses this to estimate the signal switching probabilities at each phase level and then sums them up to obtain the overall average dynamic power.

An analytic design exploration approach for FPGA-HLS is introduced in [160] which represents the design space as a lattice [161]. It provides a guided directive (pragma) selection and directive value-assignment methodology for very large design spaces by locally searching in the n-dimensional lattice space. This results in close approximations of the Pareto-optimal FPGA realisations, while requiring less workload and synthesis runs compared to conventional FPGA-HLS approaches.

Design space explorers that only insert unroll and pipeline pragmas do not achieve high quality results for applications with variable loop bounds in their innermost loops. The work presented in [162] is an FPGA-HLS optimiser and design space explorer that addresses this shortcoming using a series of code transformations based on partial unrolling and pipelining. It also uses a resource/cycle estimation model for variable loops by interpolating a small number of synthesis reports from Xilinx Vivado HLS resulting from dynamic software profiling with real workloads.

A high-level but accurate analytic hardware resource estimator for FPGA HLS accelerators is presented in [163].

The estimator uses models for FPGA hardware resources (such as DSPs, BRAMs, LUTs and Flip-flops) to explore the trade-off between HLS optimisation directives and the area occupied on the target FPGA with no need to generate the HDL. MPSeeker [164] and the framework presented in [165] are other examples of HDL-independent fast design space explorers, which accurately estimate resource utilisation and performance of the functions offloaded onto the FPGA via HLS.

The Supporting uTilities for Heterogeneous EMbedded (STHEM) image processing platform [10] automates iterative, and otherwise tedious and time consuming FPGA HLS steps such as performance evaluation, design space exploration and vendor-specific HLS tool configuration. The goal is to allow the system designer to concentrate on just the main application development steps. STHEM consists of vendor tools, including Xilinx SDSoC 2017.4 and HIPPEROS [166], and utilities including: an analysis utility, which provides the designer the means to improve the performance of the software and HLS code (automatic visual static analysis based on the code's CFG, runtime, visual power/energy profiling using power measurement utility, and automatic design space explorer through different parts of the HLS design flow); a power measurement utility, which is a piece of hardware designed to simultaneously measure the current for a number of modules and to sample the program counters of the cores available on the SDSoC board under test; a dynamic partial reconfiguration utility, which helps the system developer to come up with a reconfigurable design; and HiFlipVX, an image processing library providing basic image processing functions for streaming applications.

A very recent project [167] seeks to address the major concerns of HLS developers when evaluating HLS designs, inaccurate timing and resource utilisation summaries provided by the HLS tools, by introducing *Pyramid*, a machine learning framework, that is claimed to evaluate the performance and FPGA resource utilisation of an HLS design more precisely.

Giorgi *et al.* [168] modify the conventional HLS tool flow by introducing a new modelling step prior to the design space exploration stage. This pre-exploration modelling, which takes advantage of HP-Labs COTSon full-system simulator [169], can narrow down the design candidates for FPGA-HLS implementation before they are sent for design space exploration, resulting in considerable reduction in the architecture selection process time.

A considerable number of works, such as those reported in [158], [170]–[174], introduce automated high-level performance in-circuit debugging tools in hardware and/or software, mostly via source-to-source compilation, for designs developed via the FPGA-HLS process.

4) AUTOMATIC PARTITIONING

In [175], a programming framework for legacy C applications is presented that exploits a *gcc*-plugin to automatically extract candidate code segments from the C code for implementation as hardware functions on the FPGA, and then to synthesise the selected C regions to the final RTL code as accelerators that can be invoked by the software part of the application.

Through a series of experiments the authors of [8] develop a model that can estimate the potential candidates for hardware offloading using a number of software performance measures such as predictable branches, L1 cache miss, number of instruction dispatched, main and second execution unit instructions, and Load/Store instructions (provided by the CPU's performance monitoring unit) along with L2 cache data hits and requests (accessed via the L2 cache controller). Via a series of thorough studies it is found that, in most cases, making an accurate pre-synthesis estimate of the FPGA accelerators performance is not possible just by using software metrics unless the prediction model can also benefit from scheduling and resource allocation information generated by the HLS tool, as well as information about the system memory architecture and the program's CDFG (produced via instrumented profiling the program on the CPU using real benchmarks).

5) DATA TRANSFER AND MEMORY INTERFACE ANALYSIS

In HLS, for cases where explicit data communication to and from the FPGA is preferable (in contrast to exploiting the shared memory), the amount of data to be exchanged between the FPGA and the rest of the computing system must be available at design time. Providing this information is very troublesome, if not impossible, especially when pointers and dynamic data structures are used in the HLL application. In [9] a semi-automatic methodology is presented that analyses the source code (or its IR) to identify the size of data structures referred to by pointers during execution of the C application. This static tool first analyses the source program to calculate the maximum possible size of each data structure passed by a pointer parameter. It is claimed to achieve near perfect correctness (real size never exceeds the maximum) and accuracy (real size is very close to the maximum).

An ''automatic data placement framework'' for heterogeneous SoC platforms, supporting FPGAs and CPUs, is proposed in [176]. The framework can be integrated into the Xilinx Vivado HLS. By solving an integer linear programming (ILP) formulation, the framework finds the optimum allocation for array objects, proving that data placement approaches conventionally employed for software-controlled on-chip memories in CPU/GPU do not provide optimal solutions. The ILP calculations take into account factors such as how FPGA computation is performed and what the memory access patterns are.

In [177], a prediction model for optimum partitioning of any given workload between a CPU and FPGA is presented. This model, which incorporates the communication overhead (CPU time plus data transfer latency), provides the system developer an understanding of the workload size for which using the FPGA can be beneficial.

6) AUTOMATIC CODE REFACTORING

The designer of a HLS system may find that some parts of the code are not synthesisable due to data dependencies (for example between successive iterations of a loop). Alternatively, code segments may be synthesisable, but when offloaded the overall performance of the final realisation on FPGA and CPU still does not meet the application's requirements.

''Code refactoring'' or ''code restructuring'' can be a solution for transforming such code segments into HLS-friendly code. While the concept of code refactoring/restructuring can be considered to mean any possible change to code, it is mainly practised for rewriting the program based on: arithmetic or algebraic identities (such as associativity and distributively); reshaping memory access patterns (for example to reduce memory bandwidth); eliminating unsupported pointer-based memory accesses; control flow optimisations (e.g. partial loop unrolling); inlining functions; and substituting synthesisable algorithms for required functionalities (for example by using equivalent hardware-friendly function libraries or IPs).

A very recent work reported in [178] develops machine learning models, trained using high-level source code, to predict possible routing congestion in the hardware implementation via FPGA-HLS, without analysing post-implementation details after Place and Route. This approach then backtracks the predicted congestion measures to the HLS IR so they can be used to spot the congestion causes in the source code.

While using HLS has lowered the design effort for programming FPGAs, many HLL coding techniques and features conventionally exploited by software developers cannot be used by system designers for FPGA-HLS. On the other hand, effective hardware realisation on FPGA platforms may need extensive use of hardware specific structures including FIFOs, CAMs and shift registers. `hlslib` [179], a collection of open source software tools, plug-in hardware modules and sample programs has been developed to improve the efficiency of HLS-based FPGA. While at the moment only designs targeting Xilinx FPGA platforms can benefit from `hlslib`'s full features, a number of Intel FPGA OpenCL elements are also covered by the library. For example, using the `hlslib::DataPack` class defined in `hlslib/xilinx/DataPack.h` the designer can easily define wide data paths and make use of SIMD parallelisation to implement vector operation in the HLS code. Other features provided by hlslib are common compile-time functions used in hardware development such as $\log_2$, a streaming implementation of accumulation, and `hlslib::Stream` class, an alternative to Xilinx's native `hls::Stream`, that provides a richer interface for HLS streams.

While using libraries of high performance application-specific IPs is a common practice among RTL designers developing high-end FPGA realisations, high-level languages may potentially provide limited space for HLS implementation of unconventional computations. Uguen et.al. in [180] try to shrink this gap by developing a plugin for the source-to-source compiler GeCoS [181] to transform selected floating-point operations (such as summation, accumulation and sum-of-products) into sequences of simpler operators using non-standard arithmetic formats, via IR manipulation instructed by compiler directives. The workflow receives required domain-specific information such as the variables ranges and the application's acceptable accuracy, and in turn, by exploiting new application-specific intermediate formats, an alternative HLS-friendly HLL code is generated that trades computation accuracy with latency and implementation area by relaxing some tight IEEE-754 and C11 standards restrictions.

The increasing tendency to implement machine learning algorithms on FPGAs has been the main motivation behind developing *hls4ml* [182], a deep neural network translation library that automatically compiles a machine learning model into HLS-friendly code. hls4ml supports common machine learning models and architectures and provides opportunity for designers to reconfigure the architectures to explore different trade-offs between latency, initiation interval, and resource utilisation, based on the application's requirements. hls4ml is able to port fully connected networks trained from open-source software packages Keras and PyTorch to a Xilinx FPGA, supporting a wide variety of activation functions such as ReLu, tanh, sigmoid and softmax, targeting. By using automated hls4ml, prototyping a neural network to the target FPGA can be done much faster than by directly designing the network architecture for that FPGA via either HDL or HLS.

The source-to-source optimiser toolset introduced in [183] automatically rewrites the HLL program to improve the application latency, while maintaining accuracy and resource usage within in acceptable ranges. This tool, which suits numerically intensive applications, automatically identifies opportunities for ''expression balancing'' (rearranging operators into a balanced tree to exploit parallelism and construct more efficient pipelines). It applies the transformations to the program in different ways to construct a variety of functionally equivalent program candidates and then develops a 4-dimensional Pareto frontier that the system developer can use to find the trade-off between execution time, accuracy, FPGA's LUTs and DSP usage, that most suits most the application's requirements, when realising the design via Xilinx Vivado HLS.

A framework called AFFIX is presented in [184] that automatically generates FPGA accelerators for a diverse range of computer vision algorithms based on OpenVX kernels. Previous similar research projects just covered a small subset of OpenVX functions and were able to convert only simple vision-based algorithms through a non-automated flow to non-optimised FPGA accelerators. It takes as its input the vision algorithm developed as a DAG by the system designer. It then converts OpenCV functions used in the algorithm into optimised OpenCV blocks that can be efficiently synthesised onto the FPGA. The tool develops an OpenCV version of the OpenVX library with reconfigurable kernels that can co-run on the CPU and the FPGA, while the kernels and are able to be exploited even outside the AFFIX framework.

VI. CONCLUSION

The motivation for this article was a vision of a software tool that could automatically deploy sections of code, originally written for a conventional CPU, to FPGA accelerators to achieve an implementation advantage, whether that be latency, throughput, energy or some other optimisation goal. How close are existing tools to this delivering this vision, and where are the capability gaps?

To survey and evaluate existing tools we provided a classification of design flows in FIGURE 1. This classification has neatly expressed the relationships between many different hardware synthesis tools surveyed under three broad approaches: manual re-coding, behavioural synthesis, and dataflow synthesis, as well as variations of these.

Sections III and IV surveyed many commercial and research tools for code deployment, organised according to the framework in FIGURE 1. Wherever possible we have identified the pedigree of these tools and their relationship to other tools in the survey. For the more widely used or surveyed tools we have provided an overview of salient features and capabilities.

None of the existing tools are able to fulfil the vision of fully automatic deployment of general C/C++ code to a heterogeneous system of FPGAs and CPUs. Capability gaps include the generation of synthesisable HLS code from HLL code that uses pointers to pointers or functions, recursive functions, or dynamic memory allocations. Other challenges include efficient partitioning of the code, optimisation of generated hardware, and design space exploration. All of these are the subject of active research efforts as surveyed in Section V-B.

This work has also identified a trend that is somewhat orthogonal to the idea of automatic code deployment. There is currently significant work in approaches to express the application in a high level language that is more amenable for execution on diverse platforms. Examples include the growing proliferation of domain-specific languages or dataflow representations.

REFERENCES

[1] P. A. Gargini, "Roadmap evolution: From NTRS to ITRS, from ITRS 2.0 to IRDS," in *Proc. Int. Conf. Extreme Ultraviolet Lithography*, vol. 10450, P. A. Gargini, P. P. Naulleau, K. G. Ronse, and T. Itani, Eds. Bellingham, WA, USA: SPIE, 2017, pp. 191–205, doi: 10.1117/12.2280803.

[2] A. Cornu, S. Derrien, and D. Lavenier, "HLS tools for FPGA: Faster development with better performance," in *Reconfigurable Computing: Architectures, Tools and Applications*, A. Koch, R. Krishnamurthy, J. McAllister, R. Woods, and T. El-Ghazawi, Eds. Berlin, Germany: Springer, 2011, pp. 67–78.

[3] D. J. Pagliari, M. R. Casu, and L. P. Carloni, "Accelerators for breast cancer detection," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 3, pp. 1–25, Jul. 2017, doi: 10.1145/2983630.

[4] X. Liu, Y. Chen, T. Nguyen, S. Gurumani, K. Rupnow, and D. Chen, "High level synthesis of complex applications: An H.264 video decoder," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, Feb. 2016, pp. 224–233.

[5] M. Pelcat, C. Bourrasset, L. Maggiani, and F. Berry, "Design productivity of a high level synthesis compiler versus HDL," in *Proc. Int. Conf. Embedded Comput. Syst., Archit., Modeling Simulation (SAMOS)*, Jul. 2016, pp. 140–147.

[6] W. Vanderbauwhede and G. Davidson, "Domain-specific acceleration and auto-parallelization of legacy scientific code in FORTRAN 77 using source-to-source compilation," *Comput. Fluids*, vol. 173, pp. 1–5, Sep. 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0045793018302950

[7] W. Vanderbauwhede and S. Waqar Nabi, "Towards automatic transformation of legacy scientific code into OpenCL for optimal performance on FPGAs," 2018, *arXiv:1901.00416*. [Online]. Available: http://arxiv.org/abs/1901.00416

[8] B. A. Syrowik, B. Fort, and S. D. Brown, "Use of CPU performance counters for accelerator selection in HLS-generated CPU-accelerator systems," in *Proc. 9th Int. Symp. Highly-Efficient Accel. Reconfigurable Technol.*, New York, NY, USA, Jun. 2018, pp. 1–6, doi: 10.1145/3241793.3241805.

[9] M. Lattuada, F. Ferrandi, and M. Perrotin, "Data transfers analysis in computer assisted design flow of FPGA accelerators for aerospace systems," *IEEE Trans. Multi-Scale Comput. Syst.*, vol. 4, no. 1, pp. 3–16, Jan. 2018.

[10] A. Sadek, A. Muddukrishna, L. Kalms, A. Djupdal, A. Podlubne, A. Paolillo, D. Goehringer, and M. Jahre, "Supporting utilities for heterogeneous embedded image processing platforms (STHEM): An overview," in *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, N. Voros, M. Huebner, G. Keramidas, D. Goehringer, C. Antonopoulos, and P. C. Diniz, Eds. Cham, Switzerland: Springer, 2018, pp. 737–749.

[11] H. M. Makrani, H. Sayadi, T. Mohsenin, S. Rafatirad, A. Sasan, and H. Homayoun, "XPPE: cross-platform performance estimation of hardware accelerators using machine learning," in *Proc. 24th Asia South Pacific Design Autom. Conf.*, New York, NY, USA, Jan. 2019, pp. 727–732, doi: 10.1145/3287624.3288756.

[12] MaxCompiler. *Maxeler Technologies*. Accessed: Sep. 22, 2020. [Online]. Available: https://www.maxeler.com/products/software/maxcompiler/

[13] J. Eker and J. W. Janneck, "CAL language report: Specification of the CAL actor language, version: 1.0," Dept. EECS, Univ. California, Berkeley, CA, USA, Tech. Rep., 2003. [Online]. Available: https://ptolemy.berkeley.edu/papers/03/Cal/

[14] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Design Test Comput.*, vol. 26, no. 4, pp. 8–17, Jul./Aug. 2009.

[15] L. Daoud, D. Zydek, and H. Selvaraj, "A survey of high level synthesis languages, tools, and compilers for reconfigurable high performance computing," in *Advances in Systems Science*, J. Swiętek, A. Grzech, P. Swiętek, and J. M. Tomczak, Eds. Cham, Switzerland: Springer, 2014, pp. 483–492.

[16] *LLVM Compiler Infrastructure*. Accessed: Sep. 22, 2020. [Online]. Available: http://www.llvm.org

[17] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Des. Test Comput.*, vol. 26, no. 4, pp. 18–25, Jul. 2009.

[18] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 4, pp. 473–491, Apr. 2011.

[19] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong, *AutoPilot: A Platform-Based ESL Synthesis System*. Dordrecht, The Netherlands: Springer, 2008, pp. 99–112, doi: 10.1007/978-1-4020-8588-8_6.

[20] J. M. P. Cardoso, P. C. Diniz, and M. Weinhardt, "Compiling for reconfigurable computing: A survey," *ACM Comput. Surveys*, vol. 42, no. 4, pp. 1–65, Jun. 2010, doi: 10.1145/1749603.1749604.

[21] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt, "An overview of today's high-level synthesis tools," *Design Autom. Embedded Syst.*, vol. 16, no. 3, pp. 31–51, 2012.

[22] J. Andrade, N. George, K. Karras, D. Novo, F. Pratas, L. Sousa, P. Ienne, G. Falcao, and V. Silva, "Design space exploration of LDPC decoders using high-level synthesis," *IEEE Access*, vol. 5, pp. 14600–14615, 2017.

[23] Xilinx. *Vivado High-Level Synthesis*. Accessed: Sep. 22, 2020. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[24] Intel. *Intel FPGA SDK for OpenCL*. Accessed: Sep. 22, 2020. [Online]. Available: https://www.intel.com.au/content/www/au/en/software/programmable/sdk-for-opencl/overview.html

[25] S. Ravi and M. Joseph, "Open source HLS tools: A stepping stone for modern electronic CAD," in *Proc. IEEE Int. Conf. Comput. Intell. Comput. Res. (ICCIC)*, Dec. 2016, doi: 10.1109/ICCIC.2016.7919615.

[26] C. Pilato and F. Ferrandi, "Bambu: A modular framework for the high level synthesis of memory-intensive applications," in *Proc. 23rd Int. Conf. Field Program. Log. Appl.*, Sep. 2013, pp. 1–4, doi: 10.1109/FPL.2013.6645550.
[27] R. Nane, V.-M. Sima, B. Olivier, R. Meeuws, Y. Yankova, and K. Bertels, "DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler," in *Proc. 22nd Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2012, pp. 619–622.
[28] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-level synthesis for FPGA-based processor/accelerator systems," in *Proc. 19th ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, 2011, pp. 33–36.
[29] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 10, pp. 1591–1604, Oct. 2016.
[30] M. Meredith, *High-Level SystemC Synthesis with Forte's Cynthesizer*. Dordrecht, The Netherlands: Springer, 2008, pp. 75–97, doi: 10.1007/978-1-4020-8588-8_5.
[31] *Cadence C-to-Silicon Compiler*. Accessed: Feb. 18, 2020. [Online]. Available: http://www.cadence.com/products/sd/silicon_compiler/pages/default.aspx
[32] Cadence. *Stratus High-Level Synthesis*. Accessed: Sep. 22, 2020. [Online]. Available: https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html
[33] eXCite. *Y Explorations*. Accessed: Sep. 22, 2020. [Online]. Available: http://www.yxi.com/products.php
[34] M. B. Gokhale and J. M. Stone, "NAPA C: Compiling for a hybrid RISC/FPGA architecture," in *Proc. IEEE Symp. FPGAs Custom Comput. Mach.*, Apr. 1998, pp. 126–135.
[35] Intel. *Intel HLS Compiler*. Accessed: Sep. 22, 2020. [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html
[36] Mentor Graphics. *Catapult High-Level Synthesis*. Accessed: Sep. 22, 2020. [Online]. Available: https://www.mentor.com/hls-lp/catapult-high-level-synthesis/
[37] W. A. Najjar, J. Villarreal, and R. J. Halstead, *ROCCC 2.0*. Cham, Switzerland: Springer, 2016, pp. 191–204, doi: 10.1007/978-3-319-26408-0_11.
[38] ROCCC. *University of California Riverside*. Accessed: Sep. 22, 2020. [Online]. Available: http://roccc.cs.ucr.edu
[39] P. Coussy, C. Chavet, P. Bomel, D. Heller, E. Senn, and E. Martin, *GAUT: A High-Level Synthesis Tool for DSP Applications*. Dordrecht, The Netherlands: Springer, 2008, pp. 147–169, doi: 10.1007/978-1-4020-8588-8_9.
[40] Université de Bretagne-Sud. *GAUT–High-Level Synthesis Tool*. Accessed: Sep. 22, 2020. [Online]. Available: http://hls-labsticc.univ-ubs.fr
[41] S. Singh and D. J. Greaves, "Kiwi: Synthesis of FPGA circuits from parallel programs," in *Proc. 16th Int. Symp. Field-Programmable Custom Comput. Mach.*, Apr. 2008, pp. 3–12, doi: 10.1109/FCCM.2008.46.
[42] University of Cambridge. *Kiwi HLS and Kiwi Scientific Acceleration*. Accessed: Sep. 22, 2020. [Online]. Available: https://www.cl.cam.ac.uk/research/srg/han/hprls/orangepath/kiwic.html
[43] NEC Corporation. *Cyberworkbench*. Accessed: Sep. 22, 2020. [Online]. Available: https://www.nec.com/en/global/prod/cwb/index.html
[44] K. Wakabayashi, *Cyber: High Level Synthesis System from Software into ASIC*. Boston, MA, USA: Springer, 1991, pp. 127–151, doi: 10.1007/978-1-4615-3966-7_6.
[45] Altium. *Altium Designer*. Accessed: Sep. 22, 2020. [Online]. Available: https://www.altium.com/altium-designer/
[46] Synopsys. *Press Release: Synopsys Introduces Synphony High Level Synthesis*. Accessed: Sep. 22, 2020. [Online]. Available: https://news.synopsys.com/index.php?s=20295&item=123096
[47] V. Kathail, S. Aditya, R. Schreiber, B. Ramakrishna Rau, D. C. Cronquist, and M. Sivaraman, "PICO: Automatically designing custom computers," *Computer*, vol. 35, no. 9, pp. 39–47, Sep. 2002.
[48] MathWorks. *HDL Coder*. Accessed: Sep. 22, 2020. [Online]. Available: https://au.mathworks.com/products/hdl-coder.html
[49] LegUp Computing Inc. *LegUp: Software IDE for Programming FPGAs*. Accessed: Sep. 22, 2020. [Online]. Available: https://www.legupcomputing.com
[50] M. Aubury, I. Page, G. Randall, J. Saul, and R. Watts, "Handel-C language reference guide," in *Computing Laboratory*. Oxford, U.K.: Oxford Univ., 1996.
[51] Mentor Graphics. *Handel-C Synthesis Methodology*. Accessed: Sep. 22, 2020. [Online]. Available: https://www.mentor.com/products/fpga/handel-c/
[52] (2010). *DK Design Suite Manual*, Mentor Graphics. [Online]. Available: http://s3.mentor.com/public_documents/datasheet/products/fpga/handel-c/dk-design-suite/dk-ds.pdf
[53] R. Nane, V. M. Sima, C. P. Quoc, F. Goncalves, and K. Bertels, "High-level synthesis in the delft workbench Hardware/Software co-design tool-chain," in *Proc. 12th IEEE Int. Conf. Embedded Ubiquitous Comput.*, Aug. 2014, pp. 138–145.
[54] D. Chen, J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang, "xPilot: A platform-based behavioral synthesis system," SRC TechCon, OR, USA, Tech. Rep., 2005, vol. 5.
[55] Xilinx. *Vivado Design Suite–HLx Editions*. Accessed: Sep. 22, 2020. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html
[56] Xilinx. *SDSoC Development Environment*. Accessed: Sep. 22, 2020. [Online]. Available: https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html
[57] Xilinx. *SDAccel: Enabling Hardware-Accelerated Software*. Accessed: Sep. 22, 2020. [Online]. Available: https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html
[58] Xilinx. *Vitis Unified Software Platform*. Accessed: Sep. 22, 2020. [Online]. Available: https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html
[59] Khronos Group Inc. *OpenCL Overview*. Accessed: Sep. 22, 2020. [Online]. Available: https://www.khronos.org/opencl/
[60] *Clang: A C language Family Frontend for LLVM*. Accessed: Sep. 22, 2020. [Online]. Available: http://clang.llvm.org/
[61] Intel. *Intel Quartus Prime Design Software Suite*. Accessed: Sep. 22, 2020. [Online]. Available: https://www.intel.com.au/content/www/au/en/software/programmable/quartus-prime/overview.html
[62] T. Bollaert, *Catapult Synthesis: A Practical Introduction to Interactive C Synthesis*. Dordrecht, The Netherlands: Springer, 2008, pp. 29–52, doi: 10.1007/978-1-4020-8588-8_3.
[63] Calypto. *Catapult C Synthesis*. Accessed: Sep. 22, 2020. [Online]. Available: http://calypto.agranderdesign.com/catapult_c_synthesis.php/
[64] Mentor Graphics. *AC Datatypes*. Accessed: Sep. 22, 2020. [Online]. Available: https://www.mentor.com/hls-lp/downloads/ac-datatypes
[65] K. Wakabayashi and T. Okamoto, "C-based SoC design flow and EDA tools: An ASIC and system vendor perspective," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 19, no. 12, pp. 1507–1522, Dec. 2000.
[66] K. Wakabayashi and B. C. Schafer, *'All-in-C' Behavioral Synthesis and Verification with CyberWorkBench*. Dordrecht, The Netherlands: Springer, 2008, pp. 113–127, doi: 10.1007/978-1-4020-8588-8_7.
[67] M. Aldham, J. Anderson, S. Brown, and A. Canis, "Low-cost hardware profiling of run-time and energy in FPGA embedded processors," in *Proc. 22nd IEEE Int. Conf. Appl.-Specific Syst., Architectures Processors (ASAP)*, Sep. 2011, pp. 61–68.
[68] Politecnico di Milano. *PandA Project*. Accessed: Sep. 22, 2020. [Online]. Available: https://panda.dei.polimi.it/
[69] Lattice Semiconductor. *Lattice Diamond Software*. Accessed: Sep. 22, 2020. [Online]. Available: http://www.latticesemi.com/latticediamond
[70] Mentor. *Modelsim*. Accessed: Sep. 22, 2020. [Online]. Available: https://www.mentor.com/products/fv/modelsim/
[71] Veripool. *Verilator*. Accessed: Sep. 22, 2020. [Online]. Available: https://www.veripool.org/wiki/verilator
[72] *Verilog Icarus*. Accessed: Sep. 22, 2020. [Online]. Available: http://iverilog.icarus.com/
[73] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, and M. S. Lam, "SUIF: An infrastructure for research on parallelizing and optimizing compilers," *ACM SIGPLAN Notices*, vol. 29, no. 12, pp. 31–37, 1994.
[74] Z. Guo, W. Najjar, and B. Buyukkurt, "Efficient hardware code generation for FPGAs," *ACM Trans. Archit. Code Optim.*, vol. 5, no. 1, pp. 1–26, May 2008, doi: 10.1145/1369396.1369402.

[75] Synopsys. *Synopsys Design Compiler*. Accessed: Sep. 22, 2020. [Online]. Available: https://www.synopsys.com/support/training/rtl-synthesis/design-compiler-rtl-synthesis.html

[76] A. M. Zaidi, "Exposing ILP in custom hardware with a dataflow compiler IR," in *Proc. 22nd Int. Conf. Parallel Architectures Compilation Techn.*, Sep. 2013, p. 411.

[77] J. de Fine Licht, M. Besta, S. Meierhans, and T. Hoefler, "Transformations of high-level synthesis codes for high-performance computing," 2018, *arXiv:1805.08288*. [Online]. Available: http://arxiv.org/abs/1805.08288

[78] Silexica. *SLX FPGA*. Accessed: Sep. 22, 2020. [Online]. Available: https://www.silexica.com/products/slx-fpga/

[79] R. Leupers and J. Castrillon, "MPSoC programming using the MAPS compiler," in *Proc. 15th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2010, pp. 897–902.

[80] Merlin Compiler. *Falcon Computing Solutions*. Accessed: Sep. 22, 2020. [Online]. Available: https://www.falconcomputing.com/merlin-fpga-compiler/

[81] S. Skalicky, J. Monson, A. Schmidt, and M. French, "Hot & spicy: Improving productivity with Python and HLS for FPGAs," in *Proc. IEEE 26th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr./May 2018, pp. 85–92, doi: 10.1109/FCCM.2018.00022.

[82] E. Del Sozzo, R. Baghdadi, S. Amarasinghe, and M. D. Santambrogio, "A common backend for hardware acceleration on FPGA," in *Proc. IEEE Int. Conf. Comput. Design (ICCD)*, Nov. 2017, pp. 427–430.

[83] M. Rabozzi, G. Natale, E. Del Sozzo, A. Scolari, L. Stornaiuolo, and M. D. Santambrogio, "Heterogeneous exascale supercomputing: The role of CAD in the exaFPGA project," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2017, pp. 410–415.

[84] R. Meeuws, C. Galuzzi, and K. Bertels, "High level quantitative hardware prediction modeling using statistical methods," in *Proc. Int. Conf. Embedded Comput. Syst., Archit., Modeling Simulation*, Jul. 2011, pp. 140–149.

[85] S. A. Ostadzadeh, R. J. Meeuws, C. Galuzzi, and K. Bertels, "Quad–a memory access pattern analyser," in *Reconfigurable Computing: Architectures, Tools and Applications*. Berlin, Germany: Springer, 2010, pp. 269–281.

[86] E. Del Sozzo, R. Baghdadi, S. Amarasinghe, and M. D. Santambrogio, "A unified backend for targeting FPGAs from DSLs," in *Proc. IEEE 29th Int. Conf. Appl.-Specific Syst., Architectures Processors (ASAP)*, Jul. 2018, pp. 1–8, doi: 10.1109/ASAP.2018.8445108.

[87] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, "Decoupling algorithms from schedules for easy optimization of image processing pipelines," *ACM Trans. Graph.*, vol. 31, no. 4, pp. 1–12, Aug. 2012, doi: 10.1145/2185520.2185528.

[88] R. Baghdadi, J. Ray, M. Ben Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, "Tiramisu: A polyhedral compiler for expressing fast and portable code," 2018, *arXiv:1804.10694*. [Online]. Available: http://arxiv.org/abs/1804.10694

[89] L. Di Tucci, M. Rabozzi, L. Stornaiuolo, and M. D. Santambrogio, "The role of CAD frameworks in heterogeneous FPGA-based cloud systems," in *Proc. IEEE Int. Conf. Comput. Design (ICCD)*, Nov. 2017, pp. 423–426.

[90] E. A. Deiana, M. Rabozzi, R. Cattaneo, and M. D. Santambrogio, "A multiobjective reconfiguration-aware scheduler for FPGA-based heterogeneous architectures," in *Proc. Int. Conf. ReConFigurable Comput. FPGAs (ReConFig)*, Dec. 2015, doi: 10.1109/ReConFig.2015.7393328.

[91] C. B. D. Oliveira, J. M. P. Cardoso, and E. Marques, "High-level synthesis from c vs. A DSL-based approach," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, May 2014, pp. 257–262.

[92] N. George, H. Lee, D. Novo, T. Rompf, K. J. Brown, A. K. Sujeeth, M. Odersky, K. Olukotun, and P. Ienne, "Hardware system synthesis from domain-specific languages," in *Proc. 24th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2014, doi: 10.1109/FPL.2014.6927454.

[93] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "Polymage: Automatic optimization for image processing pipelines," *ACM SIGARCH Comput. Archit. News*, vol. 43, no. 1, pp. 429–443, Mar. 2015.

[94] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 519–530, Jun. 2013, doi: 10.1145/2499370.2462176.

[95] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan, "Darkroom: Compiling high-level image processing code into hardware pipelines," *ACM Trans. Graph.*, vol. 33, no. 4, pp. 1–11, Jul. 2014, doi: 10.1145/2601097.2601174.

[96] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek, "Terra: A multi-stage language for high-performance computing," in *Proc. 34th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, New York, NY, USA, 2013, pp. 105–116, doi: 10.1145/2491956.2462166.

[97] O. Reiche, M. A. Özkan, R. Membarth, J. Teich, and F. Hannig, "Generating FPGA-based image processing accelerators with Hipacc: (Invited paper)," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD)*, Nov. 2017, pp. 1026–1033.

[98] C. H. Yu, P. Wei, M. Grossman, P. Zhang, V. Sarker, and J. Cong, "S2FA: An accelerator automation framework for heterogeneous computing in datacenters," in *Proc. 55th ACM/ESDA/IEEE Design Autom. Conf. (DAC)*, Jun. 2018, pp. 1–6, doi: 10.1109/DAC.2018.8465827.

[99] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated End-to-End optimizing compiler for deep learning," 2018, *arXiv:1802.04799*. [Online]. Available: http://arxiv.org/abs/1802.04799

[100] J. Hegarty, R. Daly, Z. DeVito, J. Ragan-Kelley, M. Horowitz, and P. Hanrahan, "Rigel: Flexible multi-rate image processing hardware," *ACM Trans. Graph.*, vol. 35, no. 4, pp. 1–11, Jul. 2016, doi: 10.1145/2897824.2925892.

[101] R. Ierusalimschy, L. H. de Figueiredo, and W. C. Filho, "Lua—An extensible extension language," *Softw., Pract. Exper.*, vol. 26, no. 6, pp. 635–652, 1996.

[102] A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Wu, A. R. Atreya, K. Olukotun, T. Rompf, and M. Odersky, "OptiML: An implicitly parallel domain-specific language for machine learning," in *Proc. 28th Int. Conf. Int. Conf. Mach. Learn.* New York, NY, USA: Omnipress, 2011, pp. 609–616. [Online]. Available: http://dl.acm.org/citation.cfm?id=3104482.3104559

[103] H. Lee, K. Brown, A. Sujeeth, H. Chafi, T. Rompf, M. Odersky, and K. Olukotun, "Implementing domain-specific languages for heterogeneous parallel computing," *IEEE Micro*, vol. 31, no. 5, pp. 42–53, Sep. 2011.

[104] A. K. Sujeeth, T. Rompf, K. J. Brown, H. Lee, H. Chafi, V. Popic, M. Wu, A. Prokopec, V. Jovanovic, M. Odersky, and K. Olukotun, "Composition and reuse with compiled domain-specific languages," in *ECOOP 2013–Object-Oriented Programming*, G. Castagna, Ed. Berlin, Germany: Springer, 2013, pp. 52–78.

[105] L. Tsoeunyane, S. Winberg, and M. Inggs, "Software-defined radio FPGA cores: Building towards a domain-specific language," *Int. J. Reconfigurable Comput.*, vol. 2017, Jul. 2017, Art. no. 3925961, doi: 10.1155/2017/3925961.

[106] N. Chugh, V. Vasista, S. Purini, and U. Bondhugula, "A DSL compiler for accelerating image processing pipelines on FPGAs," in *Proc. Int. Conf. Parallel Architectures Compilation*, Sep. 2016, pp. 327–338.

[107] V. Benara, "Generating power and area efficient image processing pipelines through a DSL compiler on FPGAs using automated bitwidth tuning," Ph.D. dissertation, Dept. Electron. Commun. Eng., Int. Inst. Inf. Technol. Hyderabad, Hyderabad, India, 2019.

[108] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang, "HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, New York, NY, USA, Feb. 2019, pp. 242–251, doi: 10.1145/3289602.3293910.

[109] Stanford VLSI Research Group. *Halide for Heterogeneous Computing*. Accessed: Sep. 22, 2020. [Online]. Available: http://vlsiweb.stanford.edu/projects/h2hw/

[110] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz, "Programming heterogeneous systems from an image processing DSL," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 3, pp. 1–25, Sep. 2017, doi: 10.1145/3107953.

[111] A. Ishikawa, N. Fukushima, A. Maruoka, and T. Iizuka, "Halide and GENESIS for generating domain-specific architecture of guided image filtering," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2019, pp. 1–5, doi: 10.1109/iscas.2019.8702260.

[112] Fixstars Inc. *Halide to FPGA*. Accessed: May 13, 2020. [Online]. Available: https://www.halide2fpga.com/

[113] J. Weng. (2017). *Halide-SDSoC*. [Online]. Available: https://github.com/were/Halide-SDSoC

[114] S.-W. Liao, S.-Y. Kuang, C.-L. Kao, and C.-H. Tu, "A halide-based synergistic computing framework for heterogeneous systems," *J. Signal Process. Syst.*, vol. 91, nos. 3–4, pp. 219–233, Sep. 2017.
[115] T. Carlson and E. Van Wyk, "Building parallel programming language constructs in the AbleC extensible c compiler framework: A PPoPP tutorial," in *Proc. 24th Symp. Princ. Pract. Parallel Program.*, New York, NY, USA, Feb. 2019, pp. 443–446, doi: 10.1145/3293883.3302574.
[116] O. Shacham, S. Galal, S. Sankaranarayanan, M. Wachs, J. Brunhaver, A. Vassiliev, M. Horowitz, A. Danowitz, W. Qadeer, and S. Richardson, "Avoiding game over: Bringing design to the next level," in *Proc. DAC Design Automat. Conf.*, Jun. 2012, pp. 623–629.
[117] J. B. Dennis, "First version of a data flow procedure language," in *Programming Symposium*, B. Robinet, Ed. Berlin, Germany: Springer, 1974, pp. 362–376.
[118] K. Gilles, "The semantics of a simple language for parallel programming," *Inf. Process.*, vol. 74, pp. 471–475, Aug. 1974.
[119] J. von Neumann, "First draft of a report on the EDVAC," *IEEE Ann. Hist. Comput.*, vol. 15, no. 4, pp. 27–75, 1993.
[120] *Multiscale Dataflow Programming–Tutorials*, 17th ed., Maxeler Technol., Palo Alto, CA, USA, 2017.
[121] E. Bezati, S. C. Brunet, M. Mattavelli, and J. W. Janneck, "High-level system synthesis and optimization of dataflow programs for MPSoCs," in *Proc. 50th Asilomar Conf. Signals, Syst. Comput.*, Nov. 2016, pp. 417–421.
[122] Openspl. *Maxeler Technologies*. [Online]. Available: http://www.openspl.org/
[123] J. Sérot, F. Berry, and S. Ahmed, *CAPH: A Language for Implementing Stream-Processing Applications on FPGAs*. New York, NY, USA: Springer, 2013, pp. 201–224, doi: 10.1007/978-1-4614-1362-2_9.
[124] University of Clermont Auvergne. *CAPH–High Level Dataflow Programming for FPGAs*. Accessed: Sep. 22, 2020. [Online]. Available: http://caph.univ-bpclermont.fr/CAPH/CAPH.html
[125] J. W. Janneck, I. D. Miller, D. B. Parlour, G. Roquier, M. Wipliez, and M. Raulet, "Synthesizing hardware from dataflow programs," *J. Signal Process. Syst.*, vol. 63, no. 2, pp. 241–249, May 2011, doi: 10.1007/s11265-009-0397-5.
[126] M. Wipliez, "Compilation infrastructure for dataflow programs," M.S. thesis, School INSA de Rennes, IETR, Mullana, India, Dec. 2010. [Online]. Available: https://tel.archives-ouvertes.fr/tel-00598914
[127] *ISO/IEC 23001-4:2009 Information Technology—MPEG Systems Technologies—Part 4: Codec Configuration Representation*, Standard ISO/IEC 23001-4:2009. [Online]. Available: https://www.iso.org/standard/50556.html
[128] O. Lindtjorn, R. Clapp, O. Pell, H. Fu, M. Flynn, and O. Mencer, "Beyond traditional microprocessors for geoscience high-performance computing applications," *IEEE Micro*, vol. 31, no. 2, pp. 41–49, Mar. 2011.
[129] Maxeler Technologies. *Maxeler Products*. Accessed: Sep. 22, 2020. [Online]. Available: https://www.maxeler.com/products/
[130] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. von Platen, M. Mattavelli, and M. Raulet, "OpenDF: A dataflow toolset for reconfigurable hardware and multicore systems," *ACM SIGARCH Comput. Archit. News*, vol. 36, no. 5, pp. 29–35, Dec. 2008, doi: 10.1145/1556444.1556449.
[131] E. Bezati, "High-level synthesis of dataflow programs for heterogeneous platforms design flow tools and design space exploration," Ph.D. dissertation, School Eng., École Polytechnique Fédérale Lausanne, Lausanne, Switzerland, 2015. [Online]. Available: http://infoscience.epfl.ch/record/207992
[132] M. Abid, K. Jerbi, M. Raulet, O. Déforges, and M. Abid, "Efficient system-level hardware synthesis of dataflow programs using shared memory based FIFO," *J. Signal Process. Syst.*, vol. 90, no. 1, pp. 127–144, Jan. 2018, doi: 10.1007/s11265-017-1226-x.
[133] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, and K. Olukotun, "Spatial: A language and compiler for application accelerators," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 296–311, Dec. 2018, doi: 10.1145/3296979.3192379.
[134] F. Peverelli, M. Rabozzi, E. Del Sozzo, and M. D. Santambrogio, "OXiGen: A tool for automatic acceleration of c functions into dataflow FPGA-based kernels," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2018, pp. 91–98.
[135] D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun, "Automatic generation of efficient accelerators for reconfigurable hardware," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 115–127.
[136] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *Proc. DAC Design Automat. Conf.*, Jun. 2012, pp. 1212–1221.
[137] R. Stewart, K. Duncan, G. Michaelson, P. Garcia, D. Bhowmik, and A. Wallace, "RIPL: A parallel image processing language for FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 11, no. 1, pp. 1–24, Mar. 2018, doi: 10.1145/3180481.
[138] P. Grigoras, X. Niu, J. G. F. Coutinho, W. Luk, J. Bower, and O. Pell, "Aspect driven compilation for dataflow designs," in *Proc. IEEE 24th Int. Conf. Appl.-Specific Syst., Architectures Processors*, Jun. 2013, pp. 18–25.
[139] J. M. P. Cardoso, T. Carvalho, J. G. F. Coutinho, W. Luk, R. Nobre, P. Diniz, and Z. Petrov, "LARA: An aspect-oriented programming language for embedded systems," in *Proc. 11th Annu. Int. Conf. Aspect-Oriented Softw. Develop. (AOSD)*, New York, NY, USA, 2012, pp. 179–190, doi: 10.1145/2162049.2162071.
[140] Stanford University. *Spatial*. Accessed: Sep. 22, 2020. [Online]. Available: https://spatial-lang.org/starting
[141] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. Cambridge, MA, USA: MIT Press, 1991.
[142] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proc. IEEE*, vol. 83, no. 5, pp. 773–801, May 1995.
[143] J. Liu, S. Bayliss, and G. A. Constantinides, "Offline synthesis of online dependence testing: Parametric loop pipelining for HLS," in *Proc. IEEE 23rd Annu. Int. Symp. Field-Program. Custom Comput. Mach.*, May 2015, pp. 159–162.
[144] J. Liu, J. Wickerson, and G. A. Constantinides, "Loop splitting for efficient pipelining in high-level synthesis," in *Proc. IEEE 24th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, May 2016, pp. 72–79.
[145] J. Liu, J. Wickerson, S. Bayliss, and G. A. Constantinides, "Run fast when you can: Loop pipelining with uncertain and non-uniform memory dependencies," in *Proc. 51st Asilomar Conf. Signals, Syst., Comput.*, Oct. 2017, pp. 126–130.
[146] J. Liu, J. Wickerson, S. Bayliss, and G. A. Constantinides, "Polyhedral-based dynamic loop pipelining for high-level synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 9, pp. 1802–1815, Sep. 2018.
[147] M. Tan, G. Liu, R. Zhao, S. Dai, and Z. Zhang, "ElasticFlow: A complexity-effective approach for pipelining irregular loop nests," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD)*, Nov. 2015, pp. 78–85.
[148] S. Dai, G. Liu, R. Zhao, and Z. Zhang, "Enabling adaptive loop pipelining in high-level synthesis," in *Proc. 51st Asilomar Conf. Signals, Syst., Comput.*, Oct. 2017, pp. 131–135.
[149] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically scheduled high-level synthesis," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, New York, NY, USA, Feb. 2018, pp. 127–136, doi: 10.1145/3174243.3174264.
[150] J. Su, F. Yang, X. Zeng, D. Zhou, and J. Chen, "Efficient memory partitioning for parallel data access in FPGA via data reuse," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 36, no. 10, pp. 1674–1687, Oct. 2017.
[151] M. Wijtvliet, S. Fernando, and H. Corporaal, "SPINE: From c loop-nests to highly efficient accelerators using algorithmic species," in *Proc. 25th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2015, pp. 1–6, doi: 10.1109/fpl.2015.7294015.
[152] C. Nugteren, R. Corvino, and H. Corporaal, "Algorithmic species revisited: A program code classification based on array references," in *Proc. IEEE 6th Int. Workshop Multi-/Many-Core Comput. Syst. (MuCoCoS)*, Sep. 2013, pp. 1–8, doi: 10.1109/mucocos.2013.6633604.
[153] S. D. Fernando, M. Wijtvliet, C. Nugteren, A. Kumar, and H. Corporaal, "$(AS)^2$: Accelerator synthesis using algorithmic skeletons for rapid design space exploration," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Grenoble, France, Mar. 2015, pp. 305–308.
[154] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar, "Lin-analyzer: A high-level performance analysis tool for FPGA-based accelerators," in *Proc. 53rd Annu. Design Autom. Conf. (DAC)*, New York, NY, USA, 2016, doi: 10.1145/2897937.2898040.
[155] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, "COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD)*, Nov. 2017, pp. 430–437.

[156] Y. Chi, Y.-K. Choi, J. Cong, and J. Wang, "Rapid cycle-accurate simulator for high-level synthesis," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, New York, NY, USA, Feb. 2019, pp. 178–183, doi: 10.1145/3289602.3293918.

[157] Y.-K. Choi, P. Zhang, P. Li, and J. Cong, "HLScope+,: Fast and accurate performance estimation for FPGA HLS," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD)*, Piscataway, NJ, USA, Nov. 2017, pp. 691–698. [Online]. Available: http://dl.acm.org/citation.cfm?id=3199700.3199792

[158] Y.-K. Choi and J. Cong, "HLScope: high-level performance debugging for FPGA designs," in *Proc. IEEE 25th Annu. Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM)*, Apr. 2017, pp. 125–128.

[159] Y. Liang, S. Wang, and W. Zhang, "FlexCL: A model of performance and power for OpenCL workloads on FPGAs," *IEEE Trans. Comput.*, vol. 67, no. 12, pp. 1750–1764, Dec. 2018.

[160] L. Ferretti, G. Ansaloni, and L. Pozzi, "Lattice-traversing design space exploration for high level synthesis," in *Proc. IEEE 36th Int. Conf. Comput. Design (ICCD)*, Oct. 2018, pp. 210–217.

[161] G. Birkhoff, *Lattice Theory*, vol. 25. Providence, RI, USA: American Mathematical Society, 1940.

[162] Y.-K. Choi and J. Cong, "HLS-based optimization and design space exploration for applications with variable loop bounds," in *Proc. Int. Conf. Computer-Aided Design*, New York, NY, USA, Nov. 2018, pp. 1–8, doi: 10.1145/3240765.3240815.

[163] M. Makni, M. Baklouti, S. Niar, and M. Abid, "Hardware resource estimation for heterogeneous FPGA-based SoCs," in *Proc. Symp. Appl. Comput. (SAC)*, New York, NY, USA, 2017, pp. 1481–1487, doi: 10.1145/3019612.3019683.

[164] G. Zhong, A. Prakash, S. Wang, Y. Liang, T. Mitra, and S. Niar, "Design space exploration of FPGA-based accelerators with multi-level parallelism," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Leuven, Belgium, Mar. 2017, pp. 1141–1146. [Online]. Available: http://dl.acm.org/citation.cfm?id=3130379.3130649

[165] S. Liu, F. C. Lau, and B. C. Schafer, "Accelerating FPGA prototyping through predictive model-based HLS design space exploration," in *Proc. 56th Annu. Design Autom. Conf.*, New York, NY, USA, Jun. 2019, pp. 1–6, doi: 10.1145/3316781.3317754.

[166] *HIPPEROS SA: The Embedded Software Company*. Accessed: Sep. 22, 2020. [Online]. Available: https://www.hipperos.com/

[167] H. Mohammadi Makrani, F. Farahmand, H. Sayadi, S. Bondi, S. M. Pudukotai Dinakarrao, H. Homayoun, and S. Rafatirad, "Pyramid: Machine learning framework to estimate the optimal timing and resource usage of a high-level synthesis design," in *Proc. 29th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2019, pp. 397–403.

[168] R. Giorgi, F. Khalili, and M. Procaccini, "Translating timing into an architecture: The synergy of COTSon and HLS (domain expertise—Designing a computer architecture via HLS)," *Int. J. Reconfigurable Comput.*, vol. 2019, Nov. 2019, Art. no. 2624938, doi: 10.1155/2019/2624938.

[169] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega, "COTSon: Infrastructure for full system simulation," *ACM SIGOPS Operating Syst. Rev.*, vol. 43, no. 1, pp. 52–61, Jan. 2009, doi: 10.1145/1496909.1496921.

[170] J. S. Monson and B. L. Hutchings, "Using source-level transformations to improve high-level synthesis debug and validation on FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, New York, NY, USA, 2015, pp. 5–8, doi: 10.1145/2684746.2689087.

[171] J. Goeders and S. J. E. Wilton, "Quantifying observability for in-system debug of high-level synthesis circuits," in *Proc. 26th Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2016, pp. 1–11, doi: 10.1109/fpl.2016.7577371.

[172] P. K. Bussa, J. Goeders, and S. J. E. Wilton, "Accelerating in-system FPGA debug of high-level synthesis circuits using incremental compilation techniques," in *Proc. 27th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2017, pp. 1–4, doi: 10.23919/fpl.2017.8056800.

[173] M. B. Ashcraft and J. Goeders, "Unified on-chip software and hardware debug for HLS-accelerated programs," in *Proc. Int. Conf. Field-Program. Technol. (FPT)*, Dec. 2018, pp. 354–357.

[174] J. S. Monson and B. L. Hutchings, "Enhancing debug observability for HLS-based FPGA circuits through source-to-source compilation," *J. Parallel Distrib. Comput.*, vol. 117, pp. 148–160, Jul. 2018.

[175] M. Vogt, G. Hempel, J. Castrillon, and C. Hochberger, "GCC-plugin for automated accelerator generation and integration on hybrid FPGA-SoCs," 2015, *arXiv:1509.00025*. [Online]. Available: http://arxiv.org/abs/1509.00025

[176] S. Li, Y. Wei, and L. Ju, "Automatic data placement for CPU-FPGA heterogeneous multiprocessor System-on-Chips," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 1379–1384.

[177] A. Kroh and O. Diessel, "A short-transfer model for tightly-coupled CPU-FPGA platforms," in *Proc. Int. Conf. Field-Program. Technol. (FPT)*, Dec. 2018, pp. 366–369.

[178] J. Zhao, T. Liang, S. Sinha, and W. Zhang, "Machine learning based routing congestion prediction in FPGA high-level synthesis," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 1130–1135.

[179] J. de Fine Licht and T. Hoefler, "Hlslib: Software engineering for hardware design," 2019, *arXiv:1910.04436*. [Online]. Available: http://arxiv.org/abs/1910.04436

[180] Y. Uguen, F. de Dinechin, and S. Derrien, "Bridging high-level synthesis and application-specific arithmetic: The case study of floating-point summations," in *Proc. 27th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2017, pp. 1–8, doi: 10.23919/fpl.2017.8056792.

[181] A. Floc'h, T. Yuki, A. El-Moussawi, A. Morvan, K. Martin, M. Naullet, M. Alle, L. L'Hours, N. Simon, S. Derrien, F. Charot, C. Wolinski, and O. Sentieys, "GeCoS: A framework for prototyping custom hardware design flows," in *Proc. IEEE 13th Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, Sep. 2013, pp. 100–105.

[182] N. Tran, B. Kreis, J. Duarte, E. Kreinar, and Z. Wu "HLS-FPGa-machine-learning/hls4ml: v0.1.5," Eur. Org. Nucl. Res. (CERN), Geneva, Switzerland, Tech. Rep. v0.1.5, Aug. 2019, doi: 10.5281/zenodo.3359214.

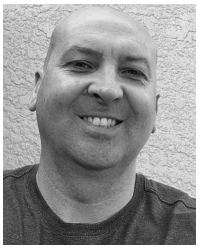
[183] Y. Gao and P. Zhang, "A survey of homogeneous and heterogeneous system architectures in high performance computing," in *Proc. IEEE Int. Conf. Smart Cloud (SmartCloud)*, Nov. 2016, pp. 170–175.

[184] S. Taheri, P. Behnam, E. Bozorgzadeh, A. Veidenbaum, and A. Nicolau, "AFFIX: Automatic acceleration framework for FPGA implementation of OpenVX vision algorithms," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, New York, NY, USA, Feb. 2019, pp. 252–261, doi: 10.1145/3289602.3293907.

MOSTAFA W. NUMAN received the B.Sc. degree in computer science from the Shahjalal University of Science and Technology, Bangladesh, in 2006, the M.Sc. degree in electrical, electronic and systems engineering from Universiti Kebangsaan Malaysia, Malaysia, in 2010, and the Ph.D. degree from The University of Adelaide, in 2017, focusing on network-on-chip (NoC) design for AI and cognitive computing. Since 2018, he has been working with The University of Adelaide as a Postdoctoral Researcher. His current research interest includes automatic software code deployment onto reconfigurable computing platforms (e.g., FPGAs).

BRADEN J. PHILLIPS (Member, IEEE) received the B.Sc. degree in mathematics and computer science and the B.E. and Ph.D. degrees in electrical and electronic engineering from The University of Adelaide, in 1992, 1993, and 2000, respectively.

He is currently the Interim Deputy Dean of Learning and Teaching with the Faculty of Engineering, Computer and Mathematical Sciences, The University of Adelaide. His current research interests include digital microelectronics, computer architecture, and high-level logic synthesis.



GAVIN S. PUDDY (Associate Member, IEEE) received the B.E. degree in computer systems engineering from the University of South Australia, in 2005. He is currently pursuing the Ph.D. degree in performance prediction of evolving systems. He joined the Defence Science and Technology Organisation, in 2005, to work on combat system architecture research for surface and subsurface platforms. He also leads up a research program on early evaluation methods into nonfunctional performances of resource constraint real-time systems within the defence domain.

KATRINA FALKNER is currently the Executive Dean of the Faculty of Engineering, Computer and Mathematical Sciences and a Professor with the School of Computer Science. She also leads the Computer Science Education Research Group and the Modeling and Analysis Program within the Centre for Distributed and Intelligent Technologies. She has extensive experience in industry consultation, including policy development and the establishment of national equity campaigns, crossing both the areas of computer science education, and distributed systems and modeling.