Huang L, Li DL, Wang KP *et al.* A survey on performance optimization of high-level synthesis tools. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 35(3): 697–720 May 2020. DOI 10.1007/s11390-020-9414-8

A Survey on Performance Optimization of High-Level Synthesis Tools

Lan Huang[1,2], *Distinguished Member, CCF*, Da-Lin Li[1,3], Kang-Ping Wang[1,2,*], Teng Gao[1], and Adriano Tavares[4]

[1] *College of Computer Science and Technology, Jilin University, Changchun 130012, China*

[2] *Key Laboratory of Symbol Computation and Knowledge Engineering of Ministry of Education*
Jilin University, Changchun 130012, China

[3] *Zhuhai Laboratory of Key Laboratory of Symbol Computation and Knowledge Engineering of Ministry of Education*
Zhuhai College of Jilin University, Zhuhai 519041, China

[4] *Algorithm Center, University of Minho, Guimaraes 4800058, Portugal*

E-mail: huanglan@jlu.edu.cn; dlli16@mails.jlu.edu.cn; wangkp@jlu.edu.cn; gaoteng18@mails.jlu.edu.cn
yannitavares@gmail.com

Received January 18, 2019; revised December 11, 2019.

Abstract Field-programmable gate arrays (FPGAs) have recently evolved as a valuable component of the heterogeneous computing. The register transfer level (RTL) design flows demand the designers to be experienced in hardware, resulting in a possible failure of time-to-market. High-level synthesis (HLS) permits designers to work at a higher level of abstraction through synthesizing high-level language programs to RTL descriptions. This provides a promising approach to solve these problems. However, the performance of HLS tools still has limitations. For example, designers remain exposed to various aspects of hardware design, development cycles are still time consuming, and the quality of results (QoR) of HLS tools is far behind that of RTL flows. In this paper, we survey the literature published since 2014 focusing on the performance optimization of HLS tools. Compared with previous work, we extend the scope of the performance of HLS tools, and present a set of three-level evaluation criteria, covering from ease of use of the HLS tools to promotion on specific metrics of QoR. We also propose performance evaluation equations for describing the relation between the performance optimization and the QoR. We find that it needs more efforts on the ease of use for efficient HLS tools. We suggest that it is better to draw an analogy between the HLS development process and the embedded system design process, and to provide more elastic HLS methodology which integrates FPGAs virtual machines.

Keywords evaluation criterion, field-programmable gate array (FPGA), high-level synthesis (HLS), performance optimization, quality of results (QoR)

1 Introduction

In recent years, due to a combination of physical limitations and economic factors, it is hard for Moore's Law to go forward. The high-performance computing (HPC) community starts to treat heterogeneous computing as an alternative approach for high-throughput and energy-efficient processing, as the common processing and controlling parts are designated to processors and the computing-intensive parts are accelerated by application-specific customized hardware [1].

Field-programmable gate arrays (FPGAs) have recently been an arising accelerator of the heterogeneous computing. Different from CPUs and graphics processing units (GPUs), FPGAs are integrated circuits that can run without instructions and operating systems, providing FPGAs an advantage in delay and power consumption over the processors of von Neumann architecture [2]. FPGAs are designed by hardware description languages (HDLs). Programming in HDLs is actually a process of functional circuits description, where functionalities are specified at a low

---

Survey

This work was supported by the National Natural Science Foundation of China under Grant No. 61772227, the Development Project of Jilin Province of China under Grant Nos. 20190201273JC and 2020C003, Guangdong Key Project for Applied Fundamental Research under Grant No. 2018KZDXM076, and Jilin Provincial Key Laboratory of Big Date Intelligent Computing under Grant No. 20180622002JC.

*Corresponding Author

©Institute of Computing Technology, Chinese Academy of Sciences 2020

level of abstraction. It is time-consuming and requires the algorithm designers being specialized in hardware.

In order to increase productivity and promote FPGAs to a wider user community, new design methodologies in a high-level design abstraction have been present in recent years, including FPGA High-Level Synthesis (HLS) [3] and FPGA overlay [4–6]. HLS accepts design in high-level language (e.g., C, C++, and SystemC) and generates synthesizable cycle-accurate register transfer level (RTL) through code transformations and synthesis optimizations. FPGA overlay is a coarse-grained design abstraction layer over fine-grained FPGAs resources, and the abstraction layer should be pre-implemented for dedicated applications [7–9]. The design in high-level language will be complied, scheduled, and mapped to the overlay architecture [10].

HLS has many advantages, for instance, accomplishing FPGA design in higher level abstraction with less hardware knowledge, exploring design space faster with little modification of the programs, richer and more convenient verification and debugging methods, etc. HLS also has obvious drawbacks: designers remain exposed to various aspects of hardware design; development cycles are still time consuming; the quality of results (QoR) of HLS tools is far behind that of RTL flows. To address these problems, the HLS community contributes a lot on optimizing the performance of the HLS tools. The QoR has improved with the newest generation of HLS tools. HLS has recently been applied to a variety of applications (e.g., medical imaging, convolutional neural networks, and machine learning), with significant benefits in terms of performance and energy consumption [11].

In this paper, we survey the scientific literature published since 2014 focusing on the performance improvement of HLS tools. Our work has four main contributions:

- a set of three-level evaluation criteria, covering from the ease of use of the HLS tools to promotion on specific metrics of QoR (Section 3);
- performance evaluation equations for describing the relation between the performance optimization contributions and the QoR of the HLS tools (Section 3);
- a survey of the literature suggesting research directions and ways to improve HLS (Section 4);
- a deep discussion on the challenge and develop trends of the HLS tools (Section 5).

This survey shows that HLS is a promising way to improve the productivity of FPGAs. However, it still needs more efforts on the ease of use for efficient HLS tools. We suggest that it is better to draw an analogy between the HLS development process and the embedded system design process, and to provide a more elastic HLS methodology.

2 Overview of HLS Tools

In this section, we firstly briefly review the development history of HLS tools. Secondly, we introduce the general working process of HLS tools. Thirdly, we present an overview of HLS tools in use. Lastly, for further understanding the design philosophy and the advantages/limitations of HLS tools, we compare FPGA HLS with FPGA overlay, another approach to increasing the design productivity using high-level design abstraction, in terms of design flows, development efficiency, and performance.

2.1 Brief History of HLS Tools

Until the late 1960s, the design, optimization, and lay-out of ICs were processed manually. Gate-level simulation appeared in the early 1970s. Cycle-based simulation was introduced in 1979. Automatic design approaches, such as schematic circuit capture, formal verification, place-and-route, and static timing analysis became available during 1980s. Later, the emergence of Verilog (1986) and VHDL (1987), which were named hardware description languages (HDLs), strengthened the capability of simulation tools. The first generation of commercial high-level synthesis (HLS) tools was introduced in the 2000s, which was mainly designed for data path applications [12, 13]. Almost the same period, more research work was invested in hardware-software co-design methods, which includes exploration, estimation, partitioning, communication, interfacing, co-simulation and synthesis [14]. The IP cores, which are actually reuse techniques, emerged at the same time.

Driven by the growth of the silicon capability and the increase of the application complexity, design tools and methodologies in higher abstraction levels are desired. Nowadays, the new emerging HLS tools are more oriented to raise the abstraction levels and automate the processes of the synthesis and the verification, so that the design space exploration can be more rapidly and efficiently.

2.2 Working Process of HLS Tools

With HLS tools, untimed high-level language descriptions (C, C++, SystemC, Java, etc.) are translated into a cycle-based implementation automatically

or semi-automatically. Besides the communication interfaces and the memory banks, the generated implementation is expressed in RTLs, which contains a controller and a data path (i.e., multiplexers, registers, functional modules, and buses), following the requirement of the functional specification and the constraints designed by the designer. As shown in Fig.1, the task of an HLS tool contains the following steps:

- compiling high-level language description;
- transforming functional specification into hardware controlling structures according to the directives from the designer;
- allocating hardware resources (storage components, functional units, buses, and so on);
- scheduling the operations to dedicated clock-cycles;
- binding the operations to customized or integrated functional units;
- binding variables to storage components (LUTs, flip-flops, or BRAMs);
- binding data transfers to specific buses;
- generating the RTL description.

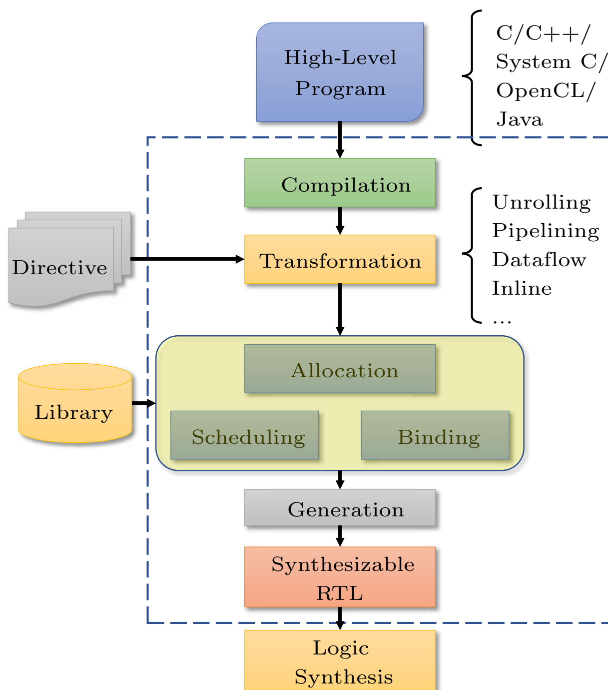


Fig.1. Typical high-level synthesis (HLS) flow.

2.3 Overview of HLS Tools in Use

After years of development, more than 30 kinds of HLS tools have been implemented [15]. For various reasons (e.g., abandoned by communities or purchased by commercial companies), some of them are no longer used. In this paper, we only introduce the HLS tools still in use when writing this article. We distinguish the HLS tools between two categories: academic ones and commercial ones. Their general information is listed in Table 1.

2.3.1 *Academic Tools*

The academic version of LegUp [16] was first released in 2011 at the University of Toronto, and its newest version is 4.0. LegUp is oriented to Altera FPGA families specifically, and it synthesizes software threads automatically into parallel-operated hardware modules through supporting Pthreads and OpenMP. Most C features are supported except memory allocation and recursion. Based on bit mask analysis and variable range of static compile-time, LegUp can automatically shrink data path widths by reducing bit width. Register removal and multi-cycle path analysis are also supported, where the registers on some paths completed in more than one cycle are removed and the constraints for the back-end of the tool flow are generated accordingly.

GAUT [17] is an open-source HLS tool which is designed specifically to digital signal processing (DSP) applications. The potential parallelism of the applications is first extracted prior to the scheduling, allocation, and binding tasks. Then a potentially pipelined architecture is generated, consisting of processing and memory modules, together with a communication unit with a GALS/LIS interface. The throughput and the clock period are mandatory constrained during the synthesis process, and the I/O timing diagram and the memory mapping are optional.

BAMBU [18] is an open source HLS tool with a modular framework, and it supports both ASIC and FPGA technologies. It exploits novel and efficient storage architectures to implement most of the constructs in C language without requirements of any three-state. Floating-point operations are also integrated. The synthesis flow, such as constraints, transformation passes, synthesis scripts and options, is customized through XML files. The test-benches are generated automatically. Finally, BAMBU proposes a validation of the results against the corresponding high-level description.

DWARV [19] translates common C-code into VHDL for unlimited applications. It is developed from the CoSy[1] framework, which constructs a compiler which

[1] http://www.ace.nl, Dec. 2018.

Table 1. HLS Tools in Use

License	Application Domain	Compiler	Owner	Input	Output	Year	Test Bench	FP	FixP
Commercial	All	VivadoHLS	Xilinx	C/C++/ SystemC	VHDL/Verilog/ SystemC	2013	Yes	Yes	Yes
		FPGA SDK for OpenCL	Intel	C	VHDL/Verilog/ System-Verilog	2013	Yes	Yes	Yes
		LegUp	LegUp Computing Inc.	C	Verilog	2015	Yes	Yes	No
		Cyber-WorkBench	NEC	BDL	VHDL/Verilog	2011	Cycle/ Formal	Yes	Yes
		eXCite	Y Explorations	C	VHDL/Verilog	2001	Yes	No	Yes
		Catapult-C	Calypto Design Systems	C/C++/ SystemC	VHDL/Verilog/ SystemC	2004	Yes	No	Yes
		Stratus	Cadence	C/C++/ SystemC	Verilog	2004	Yes	Yes	Yes
		Bluespec	BluSpec Inc.	BSV	System-Verilog	2007	No	No	No
	Dataflow	MaxCompiler	Maxeler	MaxJ	RTL	2010	No	Yes	No
	DSP	Synphony	Synopsys	C/C++/ SystemC	VHDL/Verilog/ SystemC	2010	Yes	No	Yes
	Streaming	DK Design Suite	Mentor Graphics	Handel-C	VHDL/Verilog	2009	No	Yes	No
Academic	DSP	GAUT	U.Bretagne	C/C++	VHDL	2008	Yes	No	Yes
	Streaming	ROCCC	UC.Riverside	C subset	VHDL	2010	No	Yes	No
	All	LegUp	U.Toronto	C	Verilog	2011	Yes	Yes	No
		Bambu	PoliMi	C	Verilog	2012	Yes	Yes	No
		DWARV	TU.Delft	C subset	VHDL	2012	Yes	Yes	Yes

Note: The "application domain" column represents which application domain the tools are designed for, such as, dataflow languages, DSP or data streaming applications. The "all" in this column indicates the tools can be used in all application domains including but not limited to the previous mentioned specific ones. The "test bench" column shows whether the automatic test bench generation is supported by the tools. The "FP" and "FixP" in the last two columns demonstrate the capability of supporting floating-point and fixed-point arithmetic respectively.

is highly modularized and can integrate standard or customized optimizations easily.

2.3.2 Commercial Tools

Vivado HLS② was first developed by AutoESL, named AutoPilot [20]. Xilinx purchased AutoPilot in 2011 and released the first Vivado HLS based on LLVM [21] in 2013. Vivado HLS takes C, C++, and SystemC as input and generates HDL descriptions in Verilog, VHDL, and SystemC. The generation process can be fine-tuned through the integrated design environment and the abundant features provided by Vivado HLS. Multiple optimization options are proposed, such as loop pipelining, loop unrolling, operation chaining, and memory mapping. Both streaming and shared memory type interfaces are supported for simplifying the integration among accelerators.

FPGA SDK for OpenCL was first published by Altera. It provides a heterogeneous parallel programming environment based on the enhanced OpenCL standards [22]. In October 2018, Intel published Intel® FPGA SDK for OpenCL™ Pro Edition③. FPGA SDK for OpenCL divides the applications into two main parts: the host program managing the application and FPGA accelerator, and the FPGA programming bitstream(s) [23]. During the compiling, the OpenCL Compiler compiles the OpenCL kernels to an image file used by the host program for programming the FPGA. The host-side C compiler compiles the host program and links it to the Intel FPGA SDK for OpenCL runtime libraries. The FPGA compiler automatically unrolls the loops, while the unrolling also can be carried out manually by directives if the automatic unrolling results are not satisfied.

From 2015, the commercial version LegUp④ has been proposed with the additional characteristics of eclipse-based GUI, multi-FPGA vendor support, and cloud application acceleration.

Synphony⑤, formerly PICO [24], is an HLS tool for

②https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf, Dec. 2018.

③https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf, Dec. 2018.

④http://www.legupcomputing.com/docs/legup-6.3-docs/index.html, Dec. 2018.

⑤https://www.synopsys.com/Tools/Implementation/RTLSynthesis/Pages/SynphonyC-Compiler.aspx, Dec. 2018.

DSP design released by Synopsys. It supports both streaming and memory interfaces and provides fine-tuned performance optimizations (e.g., loop unrolling and loop pipelining). Synphony only supports fixed-point arithmetic.

CyberWorkBench[25] is a system-level design tool set consisting of synthesis, verification, and simulation. It takes the behavioral description language (BDL) as input, which is a superset of the C language with extended constructs for hardware concept expression, e.g., variables of customized bitwidth, designation of concurrency or synchronization, and explicit clock boundary definition.

In eXCite[6], for describing the communication between the software and the hardware, users have to insert the communication channels manually, which can be blocking, streaming, or indexed (e.g., for array transmission).

Catapult-C[7] is a commercial HLS tool which is flexible in choosing the objective technology and libraries, setting the cycle frequency, and mapping function parameters to either streaming interfaces, registers, RAM, or ROM. Now, it mainly focuses on low-power FPGA solution.

Stratus[8] supports formal verification between RTL and gates, and power analysis. It also provides a number of optimizations, e.g., support for FP operations with IEEE-754 single/double precision and industry-standard IEEE 1666 SystemC, C, and C++.

Bluespec Compiler (BSC)[9] takes Bluespec System Verilog (BSV) as the input language. Inspired by Haskell, BSV is a high-level functional HDL derived from Verilog, as functional modules are implemented as a set of rules under Verilog grammar. The rules are named guarded atomic actions which express behavior through concurrent cooperation finite state machines (FSMs)[26]. BSC requires designers with specific expertise.

MaxCompiler[10] is a dataflow-oriented tool, which accepts MaxJ, a Java-based language, as input. It produces synthesizable HDL description for the dataflow engines running on Maxeler's hardware platform. MaxCompiler splits applications into three components: kernel(s), manager configuration, and CPU application. The first component takes charge of the computational parts of the application on FPGAs. The second component connects kernels to the CPUs, engine RAMs, other kernels and other dataflow engines through MaxRing. The last component communicates with the dataflow engines for transferring data to the kernels and engine RAMs.

DK Design Suite[11] involves Handel-C as the input language. Handel-C is a subset of the C language and extended with hardware-specific constructs. However, the designers are required to specify timing specifications and to code the concurrence and synchronization components definitely. Besides, the users also have to map data to different storage elements manually. Hence, the designers need advanced hardware knowledge due to these additional features.

The ROCCC[12] mainly focuses on the parallelization of the applications with heavy-compute-density and little control. This limits applying it to streaming applications. And also only a subset of the C language is accepted as the design language. For instance, only integer array operations and perfectly nested loops with fixed stride are permitted.

2.4 Comparison Between FPGA HLS and FPGA Overlay

FPGA HLS and FPGA overlay are new design methodologies using high-level design abstractions to promote FPGA to more user communities than the traditional HDL methods. Both of them treat C-like high-level language programs as inputs. However, they transform and deploy the programs on the FPGAs through different approaches[27, 28].

2.4.1 Design Flows

High-Level Synthesis (*HLS*). As shown in Fig.2, an HLS tool directly maps programs to FPGAs through a design flow which mainly includes scheduling, binding, and control logic extraction[13]. Scheduling plans the

[6] http://www.yxi.com/products.php, Dec. 2018.

[7] https://www.mentor.com/hls-lp/catapult-high-level-synthesis/, Dec. 2018.

[8] https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/digital-design-signoff/stratus-ds.pdf, Dec. 2018.

[9] https://bluespec.com/54621-2/, Dec. 2018.

[10] https://www.maxeler.com/products/software/maxcompiler/, Dec. 2018.

[11] http://www.mentor.com/ products/fpga/handel-c/dk-design-suite, Dec. 2018.

[12] http://roccc.cs.ucr.edu/DOCUMENTATION-0.7.6/ROCCC-Overview.pdf, Dec. 2018.

[13] https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf, Dec. 2018.

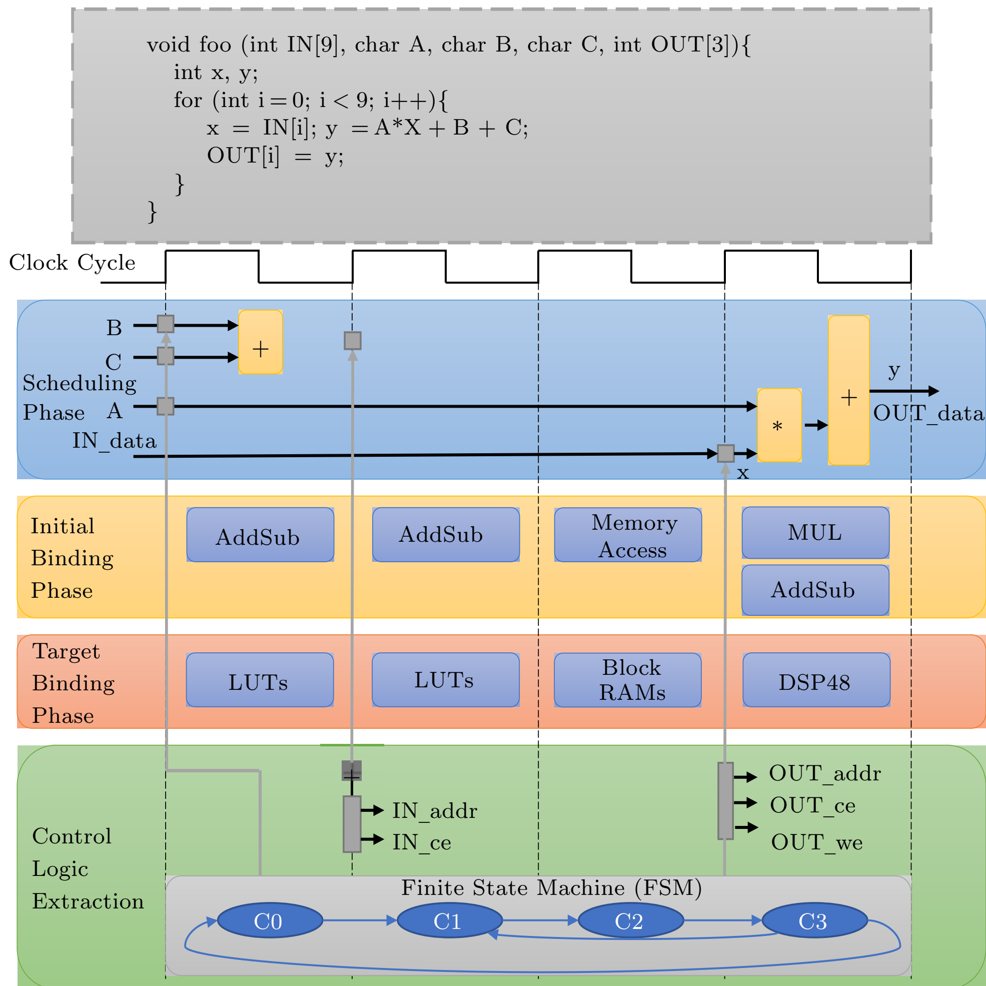


Fig.2. Scheduling, binding, and control logic extraction phases of HLS design flow.

operations of each clock cycle; binding allocates hardware resources for the scheduled operations; control logic extraction creates a finite state machine (FSM) to sequence the operations in the RTL design. In the end, all the operations and data are mapped to hardware resources of the FPGA chip, and the mapping strategy is mainly directed by the designer.

FPGA Overlay. Different from HLS, as shown in Fig.3, an overlay tool first maps the high-level language programs to the overlay which consists of an array of pre-implemented directly connected simple configurable processing elements (CPEs). Each CPE performs primitive compute operations according to a small local sequencer at each clock cycle, and the operations and data are translated into instructions of CPEs by the scheduler. After the mapping process, the overlay tools output the RTL implementation of the overlay. The designer need not consider the structure of the hardware, but only direct how many hardware resources will be involved through a parallelism factor.

2.4.2 Development Efficiency

High-Level Synthesis (HLS). The Xilinx HLS tool first compiles the high-level language programs into intermediate representation with a high-level language compiler, and then synthesizes the representation into the RTL code. At last, the RTL code is mapped to FPGAs chips through implementation process. All the processes are automatic and can be directed by the users [29]. However, there are two main drawbacks. Firstly, the synthesis and implementation cost a long time. Secondly, the users are still exposed to various aspects of hardware design. For example, if the users want to design a high-efficient pipeline, they have to find out and relieve the data dependencies, reconstruct the dataflow of the programs to be suitable for the FPGAs architecture (e.g., moving if statement into loops),

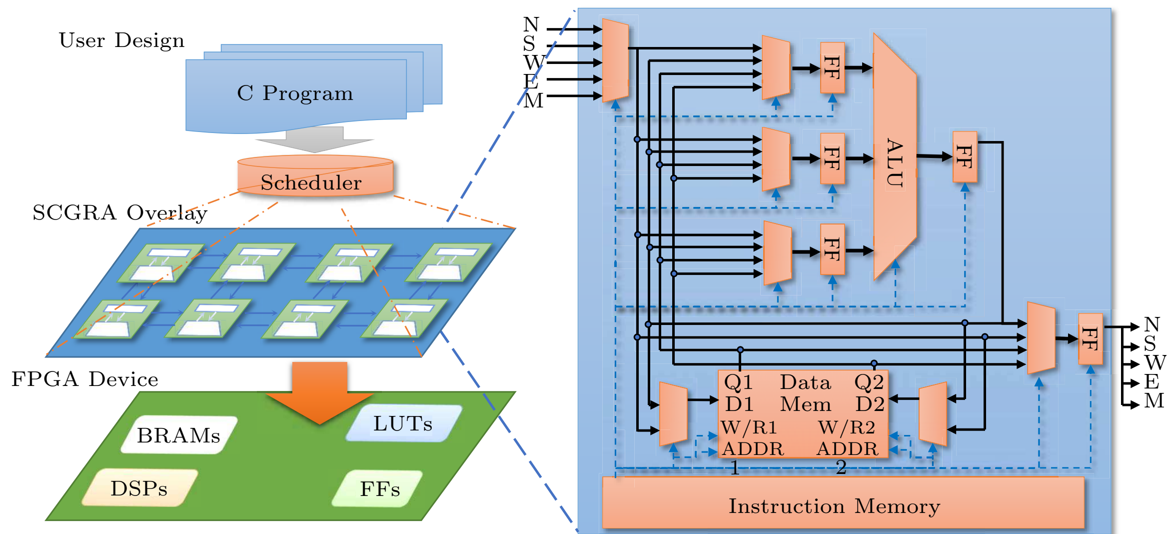


Fig.3. Overlay compiling flow. FF: flip-flops.

and elaborate on the combination of BRAMs[14] for the required bandwidth.

FPGA Overlay. Until now, no commercial FPGA overlay tool has been released. The authors in [30] provided an overlay architecture for pipelined execution of data flow graphs (DFGs) and a tool mapping DFGs to overlays. The design flow is divided into two parts: extracting DFGs from high-level language programs, placing and routing the DFG nodes onto the overlay. There is no need to consider aspects of hardware design and the FPGAs architecture when extracting DFGs. DFGs can be mapped to overlays in seconds with the provided tool, and the functional units (FUs) on the overlay are pre-compiled and configurable, which shorten the whole FPGA design time. However, the users have to get the DFGs manually or through third-party tools, and the FUs should be pre-designed according to the function of the high-level programs.

2.4.3 Performance

In [31], the authors provided a performance comparison between Vivado HLS[32] and ArchSyn[28], an FPGA overlay tool. This comparison focuses on computation intensive scenarios, and two benchmarks are tested, i.e., matrix-matrix multiplication (50×50 and 25×25 in size, respectively) and fast Fourier transform (8 192 and 1 024 in length, respectively). The evaluation metrics consist of computation latency, dynamic power consumption, power-delay product, and energy-delay product. The results show that FPGA overlay is 8x–39x faster than FPGA HLS in computation performance. But FPGA HLS performs better in dynamic power consumption metric, achieving up to 17x lower power consumption than FPGA overlay. Under the provided benchmarks, FPGA overlay has better performance than FPGA HLS, but more hardware resource must be invested than that in FPGA HLS for the same function. In addition, the authors[31] only used loop unrolling directive in Vivado HLS to exploit computation parallelism, so that the ultimate performance of Vivado HLS has not been reached.

3 Evaluation Criteria and Optimization Equation of High-Level Synthesis Tools

Evaluation criteria are used for evaluating the performance of solutions designed by the HLS tools. The HLS tools are designed to promote the abstraction level of FPGAs programming so that the designers can specify hardware functionalities with software specifications. Therefore, the criteria should evaluate not only the QoR of the synthesized implementations[33], but also the ease of use of the HLS tools. More efficient tools and development process can help designers find the optimal solution in shorter time.

The drawbacks of HLS tools are mainly reflected in three aspects: designers remain exposed to various aspects of hardware design, development cycles are still

[14] https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug871-vivado-high-level-synthesis-tutorial.pdf, Dec. 2018.

time consuming, and the quality of results (QoR) of HLS tools is far behind that of RTL flows. Performance optimization work should improve these defects.

To evaluate the performance optimization contributions on them, we propose a set of three-level evaluation criteria, as shown in Fig.4. These three levels are named ease of use, development cycles optimization, and promotion on specific metrics.

In order to reveal the relation between the performance of the HLS tools and the three-level evaluation criteria, we propose a set of performance evaluation equations. In these equations, we use a QoR trail function to describe the relation between the performance optimization of the HLS tools and the QoR of the HLS tools. We set each item of the criteria as one factor of the trail function, so that designers can use it to evaluate the possibility of achieving higher QoR and to find chance to improve the performance of HLS tools in the future.

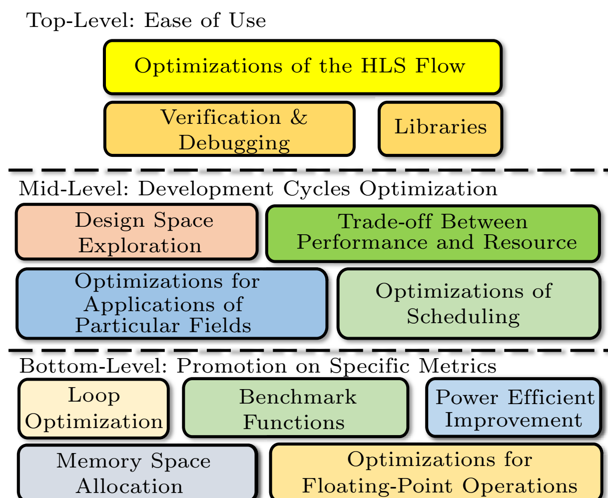


Fig.4. Three-level evaluation criteria.

3.1 Three-Level Evaluation Criteria

The three-level evaluation criteria target different stage of designing with HLS tools.

3.1.1 Top-Level: Ease of Use

The top-level consists of three parts: optimizations of the HLS flows, verification and debugging, and libraries. With HLS tools, the gap between algorithm design and hardware design becomes narrow, which opens hardware design to algorithm designers who have little hardware design experience. However, the HLS tools do not take over all the hardware-related tasks. The designers still need to think in hardware elements level, e.g., registers, clocks, and fan-out. The design process is more like embedded system design [34] than high-level software design.

In order to further reduce the workload of designers, it is better to provide more design assistant approaches. An optimized HLS tools flow can improve coding efficiency with less consideration on the hardware structures. Advanced verification and debugging tools can accelerate convergence of program stability. Libraries package specific functions, which can both speed up coding efficiency and performance of the generated circuits.

3.1.2 Mid-Level: Development Cycles Optimization

The development cycles of HLS tools consist of three steps [35]. First, the designers describe the algorithms in high-level languages, and design a set of synthesis directive based on the considerations (e.g., latency, area, power consumption). Then, the high-level language designs are compiled into RTL designs by the HLS tools. At last, the RTL implementations are transformed into device configuration through logic synthesis and implementation by the general FPGA design suites. The last two steps are automatical but time consuming, and the designers can only get the performance of the designs when the last step finished. In order to find a better solution, the designers modify the description and the directive according to the reports in the last step, and begin a new round of the last two steps. As a result, the development cycles are very long.

The goal of development cycles optimization is to find the best solution with the lowest possible time cost. Therefore, we design the mid-level of our evaluation criteria including design space exploration, optimizations of scheduling, optimizations for applications of particular fields, and trade-off between performance and resource. Because the last step of development cycles is mainly decided by the FPGA design suites, most of the contributions optimize the performance of the HLS tools from the perspective of the first two steps.

3.1.3 Bottom-Level: Promotion on Specific Metrics

The QoR of the designs are influenced by many factors [36]. We extract the most significant parts of them to construct the bottom level of our evaluation criteria. As shown in Fig.4, there are loop optimization, memory space allocation, power efficient improvement, optimizations for floating-point operations, and benchmark functions in this level. The implementation of the loop affects the delay and the throughput of designs. A

better memory space allocation can promote bandwidth and increase the scale of the designs. Floating-point operations affect the area of the solutions. The power efficiency can be promoted through many approaches, e.g., decreasing the resource for the designs, and optimizing the layout to involve less clock regions. Benchmark functions provide baselines for evaluating various solutions.

3.2 Performance Evaluation Equations of HLS Tools

As stated at the beginning of this section, the performance of the HLS tools should consider both the QoR and the ease of use (EoU). Therefore, in our work, the performance of HLS tools is defined as

$$P_{\mathrm{HLS}} = (QoR, EoU).$$

We also define QoR as

$$QoR = (F, L, T, R, P),$$

where F is the frequency, L is the latency, T is the throughput, R is the resource utilization, and P is the power efficiency, all of which are main concerns of FPGA designs and are obtained from the reports of the HLS tools [37]. In order to evaluate the performance of HLS tools, we set the QoR of manual RTL flows as the optimal QoR ($(QoR)_o$), and define the QoR satisfying performance requirements as $(QoR)_r$, and the QoR of HLS tools as $(QoR)_h$.

Normally, the designs satisfying performance requirements are obtained through multiple iterations. Based on the discussion in Subsection 3.1, we define the trial function of $(QoR)_h$ as

$$\begin{aligned} T^i_{HLS} &= e\boldsymbol{C}(\boldsymbol{x}^i)^{\mathrm{T}} \\ &= e\,(D, TO, A, S, LP, B, PE, M, FF) \\ &\quad (x^i_D, x^i_{TO}, x^i_A, x^i_S, x^i_{LP}, x^i_B, x^i_{PE}, x^i_M, x^i_{FF})^{\mathrm{T}}, \end{aligned}$$

where i is the i-th iteration, e is the coefficient quantified by EoU, $\boldsymbol{C}$ is the capability vector of the HLS tools, D, TO, A, S, LP, B, PE, M and FF represent the capabilities of design space exploration, trade-off between performance and resource, optimization for applications of particular fields, optimization of scheduling, loop optimization, benchmark functions, power efficient improvement, memory space allocation, and optimization for floating-point operations in our three-level evaluation criteria, respectively, $\boldsymbol{x}^i$ is the elaboration vector of the designers, and x^i_D, x^i_{TO}, x^i_A, x^i_S, x^i_{LP}, x^i_B, x^i_{PE}, x^i_M and x^i_{FF} represent the efforts of the designers in the corresponding aspects, such as refactoring the codes to match the hardware structure, and adjusting the bit width of data according to the accuracy requirements of specific problems. The value of T^i_{HLS} is proportional to the value of $(QoR)_h$.

$(QoR)_h$ is generated in each iteration, and then the designers modify one or several components of $\boldsymbol{x}$ to improve $(QoR)_h$ in the next iteration. A larger value of e, which is promoted by the EoU of the tools, can also accelerate the convergence of $(QoR)_h$.

We define the error between $(QoR)_o$ and the $(QoR)_h$ of the i-th iteration as

$$E = (QoR)_o - (QoR)_h,$$

and the termination conditions for HLS iteration as

$$\begin{cases} E = 0, \\ (QoR)_r = (QoR)_h. \end{cases}$$

The iteration terminates when any of the equations in the termination conditions is satisfied.

The performance of the HLS tools is reflected by e and $\boldsymbol{C}$ in the trial function. HLS tools with higher performance converge faster to the final designs. When the iteration terminates, the smaller the value of i is, the better the performance of the HLS tool is.

4 Performance Optimization of HLS Tools

HLS accelerates the FPGA design by raising the level of design abstraction beyond RTL, which significantly promotes design productivity and reduces engineering cost. However, the performance of the HLS tools, such as the ease of use and the quality of results, is still behind those of manual RTL flows. In order to narrow the performance gap between them, a lot of studies are carried out to improve HLS. Some of them are inspired by the compiler community, while the others are dedicated for hardware design. In this section, we present a survey of literatures in this field.

4.1 Ease of Use

Easy-to-use tools should have the following features: easy to understand, smooth learning curve, strong user assistance, etc. [38] HLS promises designers a higher level abstraction for FPGA designing than the traditional HDL. However, the designers now remain exposed to various aspects of hardware design. They have to be experienced in hardware for high-performance

HLS designs. Therefore, the performance improvement work should elaborate on the above problems. In this paper, we consider the ease of use of HLS tools in the following aspects: optimizations of the HLS flow, verification and debugging, and libraries.

4.1.1 *Optimizations of the HLS Flow*

In order to decrease the considerations on hardware, an efficient performance optimization approach is introducing new steps to the HLS design flow.

In [39], the authors used sequential model based optimization (SMBO) methods to select directive setting combinations automatically. They also optimized SMBO so that the proposed tool can further improve the convergence rate over the standard method. However, the proposed framework only can select the current directive settings. The directive settings of previous design process cannot be inherited, which decreases the optimization efficiency.

The authors of [40] used the embedded domain-specific language (DSL) of Haskell as the higher-order abstract expression layer. The parallel structure of the programs is first expressed in DSL, and then compiled to low level virtual machine intermediate representation (LLVM IR), which is then integrated into the HLS tools. However, the proposed method can only deal with MapReduce process, and it also needs to be improved in time scheduling.

Another rapid prototyping method to reduce the design time is introduced in [41]. The method converts the problem of hardware structure compiling to the problem of graphic searching. However, the designers have to pre-design the hardware models for the graphic nodes, which cannot be predicted when facing new problems.

Template-based approaches are provided in [42, 43], where composable and parameterizable templates of common computation patterns are optimized for parallel hardware structure. With these templates, designers can further shorten programming time. However, these studies only promote the efficiency of code generation when the parameters in the templates are changed. The performance of the generated hardware implements through these approaches still needs to be improved.

4.1.2 *Verification and Debugging*

Verification and debugging remain time-consuming parts of any design projects. Hence, it is of great importance that the HLS tools support the verification and debugging throughout the design process. Furthermore, improving their efficiency is also valuable.

HLS tools provide efficient module-level behavioral verification, but the non-behavioral factors of the generated RTL still have to be verified, e.g., the results of interface synthesis and component integration. To tackle this problem, the authors of [44] provided a methodology of reducing the RTL verification requirements, which refines the control and data-path by using semantic stubs as well as high-level control models. With this methodology, the cover goals of both high level and RTL verification are met in a semantically sound fashion, without the need of re-establishment in RTL. However, the proposed method only implements the function verification. Another important factor, time sequence, is not mentioned in this work.

A source-level debugging framework for HLS is offered in [45]. The framework supports gdb-like software debugging environment, inspection of the values of logic signals in the hardware from the perspective of C source code, and consistence verification between the logic signal in the hardware and the corresponding variable of the logic signal in software. The proposed method focuses on the source code consistence before and after synthesis. However, the time sequence debugging is still not mentioned.

In [46], a debugging approach, Hybrid Quick Error Detection (H-QED), is presented. H-QED runs software and hardware versions of the same function separately, and generates software and hardware signatures during the execution. Finally, the sequence of software signatures is compared with that of hardware signatures, and any mismatch indicates bug detection. The authors proposed an effective approach to keep the consistency between the hardware and the software codes, but the debugging processing is time consuming.

The authors of [47] provided a method to improve the debugging visibility of HLS tools through extending the source-level visibility into in-system simulation, co-simulation, and hardware execution. The method can help to find bugs with little or without effect on the resource usage, timing, latency, or throughput of the design. However, due to the source-to-source transformation adopted in the work, the original sources have to be verified again when the debugging process finished.

To further reduce bug detection latency, a just-in-time (JIT) trace-based debugging framework is proposed in [48]. This framework upgrades HLS debugging by JIT traces and automated insertion of verification code into the generated RTL. The earliest instance

of execution mismatch can be identified quickly; meanwhile, the framework provides detailed information on the faulty signal and its corresponding variable in the application source. However, because of the inserted debugging code, the original codes have to be verified again before being deployed on FPGAs chips.

4.1.3 Libraries

Library is another approach to improving the efficiency and quality of HLS designs. Normally, the libraries concentrate on one specific area, such as image processing, floating-point arithmetic. They provide interfaces for dedicated functions and are parameterizable. Moreover, they are always deeply optimized in terms of hardware structure implementation. Thus, users can save developing time and be less experienced for favorable implementations.

A C-based library for accelerating image processing was proposed in [49]. It is composed of generic building blocks which are applicable of multitude of image processing applications. The library is compactly represented so that the control performance and resource requirements are achieved easily. The performance of the library is optimized through an efficient memory architecture that facilitates easy integration of point and local image processing operators. This work proposes a flexible library for image processing, and provides a reference for the libraries dedicated in other fields.

The authors of [50] provided a C++ based image processing library which focuses on imagine processing applications that can be expressed as stream-based dataflow graphs (DFGs). The library provides users a coding template, and the users can use this template to define a DFG via functions in the library with fewer considerations of implementation details. Also, image border handling modes can be selected just by a template parameter and the whole algorithm can efficiently be parallelized with a global constant. However, the performance of the generated designs still need to be improved.

In [51], the authors provided a set of design steps to implement scalable spatial FPGA algorithms with HLS, and introduced the open source library hlslib[15] to aid productivity. Hlslib is designed from the perspective of high-performance computing (HPC), providing classes of host (CPUs) side and of device (FPGAs) side respectively. In the device side, hlslib contains classes such as DataPack for easing SIMD parallelization, Flatten for flatten loop nests, AXI for AXI stream interface and so on. Experimental results show that the applications developed with hlslib outperform the others in power consumption. However, the latency of the implementations is not mentioned in the work, which could be a weakness of the performance due to the design philosophy of hlslib.

Besides libraries, the authors of [52] developed an approach implementing HLS based upon parameterizable templates that can be composed using common data access patterns. The templates are divided into multi-levels. The hardware kernels are packaged as level 0 templates by the hardware experts, and the upper level 1 are composed of the level 0 templates. If needed by more complex functions, the level 1 templates can be used for constructing level 2 templates. Experimental results show that the template approach always run faster than the Optimized Software Code (OSC) approach. This work optimizes the HLS design flow by the hierarchy method, which proposes a good reference for the HLS tools.

4.2 Development Cycles Optimization

Development cycles cover from designing algorithm to verifying products[53, 54]. Especially for FPGAs designing, it is time consuming to find the optimal solution. The higher level abstraction of HLS promises designers to explore the optimal solution with lower time cost. In addition, the HLS community contributes a lot to further optimizing the development cycles.

4.2.1 Design Space Exploration Capabilities

HLS tools promise to reach better productivity compared with manual RTL flows, measured by the time for an acceptable solution. They usually generate an RTL description corresponding to a given software algorithm, considering specific performance targets (frequency, latency, throughput, area, power consumption, etc.), which is defined as design space exploration (DSE). As a result of the DSE on the given algorithm, only the designs with which designers are concerned are the Pareto-optimal designs. Pareto-optimality can be loosely defined as those designs which are impossible to improve one performance goal without making at least another performance goal worse.

In this paper, we define the design space as the QoR discussed in Subsection 3.2. The expression of our design space is shown as follows:

$$DS = (F, L, T, R, P),$$

[15]https://github.com/definelicht/hlslib, Dec. 2018.

where DS is the design space, F is the frequency, L is the latency, T is the throughput, R is the resource utilization, and P is the power efficiency. The values of these parameters vary according to the specific requirements. Sometimes only one of the parameters is concerned.

The process of DSE is normally described as shown in Fig.5. All the concerned constraints are entered from the left side. According to the constraints, potentially infinite state space is generated. There are different kinds of space exploration algorithms in the middle box, such as machine learning, evolutionary algorithms, pruning, or graph searching. After the exploration, the alternative solutions are filtered.

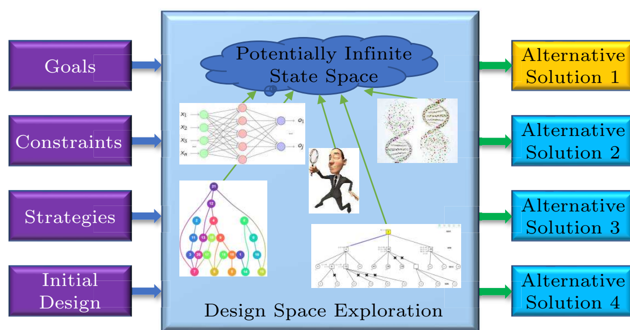


Fig.5. Design space exploration.

However, the DSE process of current HLS tools is usually done manually and acceptable for well-informed users. The researchers of the HLS community have tried a lot on improving the capability of DSE.

In [55], the authors proposed a design space exploration framework named "Design-Trotter". This framework explores the design space before the high-level synthesis. There are two sub-goals in the framework. The first one is to demonstrate all the possible parallelism of the application through a graph representation. The second one is to conduct the high-level synthesis processes by means of dynamic estimates, which are represented by parallelism vs delay trade-off curves. A point in the curves represents a possible solution in terms of parallelism options for both processing (ALUs) and data-transfer (memory access) operations. These curves could be used as a guide for the designers to choose better implementations while high-level synthesizing. However, the work constructs the design space through traversing, which makes the design space too large to find the optical solution in limited time.

The authors of [56] also presented a design space exploration methodology considering the metrics of computation parallelism and memory bandwidth. Based on an area/delay tradeoff estimation, the authors provided a pruning method to mitigate the complexity of the exploration process. The pruning method proposed in this work accelerates the exploration, which provides performance improvement during DSE.

Focusing on minimizing the hardware area while meeting a minimum throughput constraint through pipelines, the authors of [57] explored the design space by module selection and resource sharing. Furthermore, they took data introduction interval and system frequency as the performance metrics. To simplify design space, two methods are used for reducing the complexity of the module selection and the scheduling algorithm, respectively. The optical solutions selected by the proposed method tend to minimize the area, whereas the bandwidth is limited.

A predictive model DSE method based on machine learning is presented in [58], which first constructs a predictive model from a training set until a specified error threshold is achieved and then explores the design space using the model. The provided method is faster than genetic algorithm (GA) and simulated annealer (SA) based DSEs with comparable results. However, it is hard to construct the training set.

In order to promote the prediction accuracy of DSE, a performance analysis tool named Lin-Analyzer is proposed in [59]. Lin-Analyzer provides an accurate performance estimation of FPGA designs using the dynamic data dependence graph (DDDG), which can avoid time-consuming HLS runs and the false data dependencies generated from the static analysis methodologies adopted in most existing techniques including commercial HLS tools. This work proposes a graph-based method for the solution exploration, which provides an acceleration of the exploration process.

Without making the decision manually, an autonomous DSE flow is proposed in [60], which uses an iterative and greedy strategy to ensure a fast convergence towards a satisfactory solution in a short time. The proposed flow strictly respects user constraints about resource usage and target frequency. By handling profiling annotations, the DSE always focuses on the most critical sections, and the flow presents a good scalability. This work provides an automatic method for DSE. However, there is not enough optimization for the exploration strategy, which leads to a long exploration time.

To further involve the HLS tools in the DSE, the authors of [61] directly used the directive knobs of the HLS tools to construct the design space. The DSE is accelerated by first classifying these knobs according to the underly implications of each of these knobs

and then exploring them sequentially. In order to further accelerate the DSE, a probabilistic method is proposed, which computes the probability of each micro-architecture generated from constraints exploration period. As a result, new dominating designs are discovered, and the following exploration only focuses those with the highest probabilities. This work provides an efficient exploration strategy. However, the definition of the underly implications of the knobs is still time consuming.

The area information of the reports of HLS tools is inaccurate, and should be confirmed through logic synthesis. The work in [62] presents a pruning-based DSE method, which uses adaptive windowing methods to choose the candidate designs for logic synthesis. The adaptive windowing is inspired by the Rival Penalized Competitive Learning (RPCL) model and is used for classifying designs which need to be synthesized to search for the true Pareto-optimal designs. Experimental results show that the pruning-based DSE method generates similar quality trade-off curves and decreases the exploration time compared with the methods which execute a logic synthesis as long as a new design generated. The proposed method decreases the DSE time, but the optimal design has to be confirmed manually in a second round of research.

The authors of [63] proposed a cluster-based heuristic DSE method, which accelerates the DSE through decreasing the solutions needed to be synthesized. This method first clusters solutions which have a high degree of similarity, and then combines only the most effective HLS directives characterizing each cluster. With this method, only a subset of possible configurations for an HLS design need to be explored, and the close approximation of the Pareto Frontier can be achieved. However the cluster-based heuristic algorithm has to be pre-designed.

Evolutionary algorithms perform well in solving multi-objective optimization problems. Researchers also involve them in DSE. The genetic algorithm (GA) is used for DSE in [64], and the two presented encoding methods, named priority-based encoding and binding-base encoding, achieve better performance than traditional approaches. A multi-structure genetic algorithm is introduced for optimizing DSE in [65], with a cost function evaluating execution time and power consumption considering registers, functional units, demultiplexers, multiplexers, and clock frequency oscillator. This method performs better than a previous GA based heuristic approach. Non-dominated Sorting Genetic Algorithm II (NSGA II) is also introduced in [66], and is able to converge to the true Pareto front obtained from exhaustive search. Another work [67] applies simulated annealer algorithm (SA) on DSE for searching the smallest and the fastest designs, and experimental results showed that it can reduce the total runtime by an average of 66% compared with a brute force approach. Due to the property of evolutionary algorithms [68], the final solution proposed by these methods may not be optimal.

4.2.2 *Optimizations for Applications of Particular Fields*

FPGA performs superior performance in some special application areas, such as imagine and signal processing, sorting. To further improve the performance of FPGA in these areas, many efforts are invested.

In [69], the authors proposed Data-Level Parallelism (DLP) method for imagine processing, where loop coarsening is used for improving the hardware efficiency on top of loop tiling. In [70], for promoting the performance of real-time image processing, a source code and directive manipulation strategy is presented. The strategy improves performance through carefully designing the order of different optimization forms. In order to speed up the development process, a very high-level synthesis method is presented in [71]. With this method, designers can fast prototype and verify the image processing designs in the MATLAB environment. These three studies propose optimizations from different stages of image processing problems.

To improve the portability and maintainability of sorting algorithms implemented on FPGA, the work in [72] provides a framework which is composed of 10 basic sorting architectures. Users with not enough experience in hardware can construct hybrid sorting architectures with these basic architectures quickly. The proposed framework constructs new sorting architectures with pre-defined blocks. However, the connections between blocks are not optimized, which will limit the performance when the scale of the problems increases.

Memory intensive algorithms are another common problem with FPGAs. In [73], a memory interface that supports parallel memory subsystems and enables implementing atomic memory operations is designed to optimize the memory access. In [74], the authors promoted the performance through a new design methodology, based on dedicated application- and data array-specific caches. These studies focus on the bandwidth of memory, but the memory reuse is not mentioned.

4.2.3 Optimizations of Scheduling

As described in Subsection 2.4, scheduling allocates clock circles for the operations, which decides the sequence of operations, and affects the performance of the HLS-generated circuits. HLS tools in use almost universally generate statically scheduling, which implies that circuits generated by HLS tools have a hard time exploiting parallelism in code with potential memory dependencies, with control-dependent dependencies in inner loops, or where performance is limited by long latency control decisions. Researchers do many efforts to promote the performance of scheduling. We summarize the main work as described in the follows.

Multi-cycling is a well-known strategy to improve performance in digital design. In [75], the multi-cycling is introduced in HLS, and the software profiling is also used for guiding multi-cycling optimizations, which optimizes the scheduling utilizing hardware design tricks. Efficient pipeline scheduling is another difficulty for scheduling. Taking the area minimization of throughput-constrained, mapping-aware pipeline scheduling problem as the primary objective, an exact formulation is proposed in [76] so as to mitigate the pessimism inherent in static delay estimates and improve resource utilization. However, these studies only proposed the static loop dependency analysis.

In [77, 78], dynamic dependency analysis is introduced for resource allocation which is executed before scheduling. Dynamic dependency analysis can improve the qualify of the resource allocation, and then simplify and accelerate the scheduling. However, the performance of the proposed dynamic dependency is still poor, which leads to a long scheduling time.

In [79], the authors presented the dynamical scheduling by describing the implementation of a prototype synthesizer which generates a particular form of latency-insensitive synchronous circuits. Compared with static scheduling, the performance of the circuits generated from the dynamical scheduling is very significantly improved at an affordable cost. This work proposes an excellent reference on optimizing the scheduling, further to improve the synthesis and development cycles.

4.2.4 Trade-off Between Performance and Resource

An important approach to improve the performance of the FPGA design is to invest more resources. However, the amount of resources is limited. Therefore, there should be a trade-off between performance and resources. The trade-off can be affected by many factors, such as the experience of the designers, the optimization of the compliers, and so on. The community also makes a lot effort on optimizing it.

In order to achieve a better system-level throughput of the applications with multiple nested loops under a given area budget, the authors of [80] developed an algorithm to determine the optimal resource usage and initiation intervals for each loop in the applications, which obtains an average of 31% performance speedup over state-of-the-art HLS solutions according to the experimental results. However, the proposed method only can deal with the loops without dependency.

Considering the communication between parallel modules, a memory management method is described in [81], which reduces memory contention among hardware units that operate in parallel on the basis of saving connection resources among the memory banks. However, this method pays more memory resource for saving the connections.

To promote the performance of the algorithms with complicated data structures (such as heaps, trees, and so on), a method for decoupling complex data structures with a latency-insensitive interface is presented in [82]. As shown in the experimental results, this method is capable of achieving very promising speedups without causing significant area overhead. However, the decoupling process increases the risk of the stability of the programs, and designers have to verify the programs with more test cases.

Before synthesis, the source code must be compiled by the high-level language compiler. The work in [83] exploits the effect of the optimization configurations of the LLVM compiler, and presents a new performance optimization approach that interferes the compiling process manually according to the prejudgment of the compilation results. In [84], the authors proposed an automatic compiling optimization method based on machine learning method. These two studies optimize the performance of the HLS through interfering the compiling optimization process, which proposes good exploration on optimizing HLS tools.

4.3 Promotion on Specific Metrics

As discussed in Subsection 3.2, the metrics of QoR contain frequency, latency, throughput, resource utilization, power efficient, etc. FPGA design requirements usually expect solutions to be optimal in several of these metrics. Therefore, designers elaborate on the

designs so that the designs can match the requirements. The HLS community has also contributed many efforts for optimizing the HLS tools, which can help the FPGA designs to achieve better performance on these metrics without extra efforts of the designer.

4.3.1 Loop Optimization

Loop is the most commonly operation in algorithms. A high-efficient hardware acceleration of loop can greatly optimize the runtime of algorithms.

There are three factors need to be considered for loop design optimization as shown in Fig.6. We define them as the loop design space.

$$DS_{\text{loop}} = (LPU, LPP, II),$$

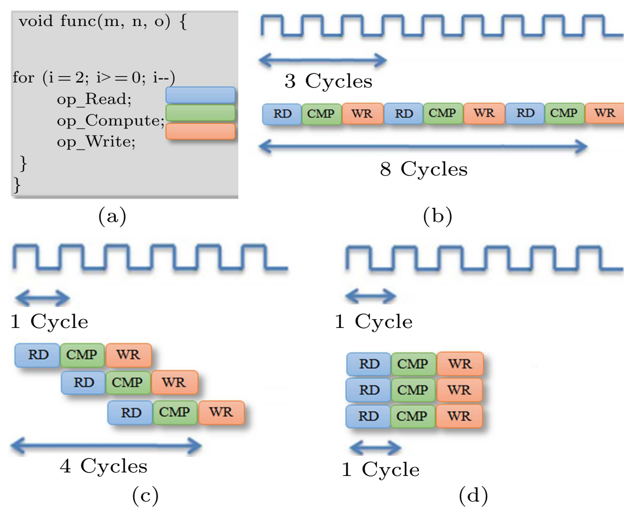


Fig. 6. Loop optimization. (a) For loop. (b) Without loop pipeline. (c) With loop pipeline. (D) Loop unroll.

where LPU is the loop unroll factor, LPP is the loop pipeline factor, and II is defined as the initiation interval for loop pipeline. As shown in Fig.6, loop unroll achieves a SIMD-like effect. Loop pipeline allows starting a new loop iteration before the completion of its predecessor, which improves the utilization of computing components and the throughput. If loop pipeline is adopted, II must be considered. It is the number of clock cycles between two adjacent loop iterations. In order to promote the throughput, II should be as small as possible, ideally 1, where a new loop iteration starts every cycle. Resource constraints and loop-carried dependencies are the two key factors for the minimum II. Resource constraints are mainly caused by input-output conflictions. For example, there are one load operation and two store operations in a loop body. In this scenario, it is impossible to achieve an II less than 2 if the memory has two ports, as there are three memory operations in each loop iteration. With respect to loop-carried dependencies, reducing II is restricted if a loop iteration cannot start without the result computed in its prior iteration, which is named data dependency.

Considering loop-carried dependencies among multiple nested loops, the authors in [85] found the optimized loop acceleration solutions through DSE. An algorithm used for deriving the Pareto-optimal curve is developed, and it can effectively prune the dominated points in the design space. However, the provided optimal solution searching strategy is not efficient enough, and the synthesis process is still time consuming.

To reduce II of loop pipelining, a customized affine+ISS (ISS) algorithm, where ISS is the acronym for Index-Set Splitting, is developed in [86]. The authors used ISS as a complementary transformation to extract additional loop-level parallelism and further reduced II of affine programs which are a class of programs with regular loop bounds and array accesses. ISS is then optimized based on memory port conflict detection for separating out conflict-free loop iterations leading to further latency improvements. However, the method proposed in this work cannot deal with loop dependency scenario.

Another method to reduce II is introduced in [87]. The method is based on an automated source-to-source code transformation, which first finds the iterations violating loop-carried dependencies, and then inserts statically-determined parametric pipeline breaks to split the loop into pieces. This can optimize the dependence patterns and improve the throughput. This work proposes a method for loop dependency, but the proposed method will increase the scale of the generated circles.

Focusing on the dependencies between loops and arrays, the framework proposed in [88] represents these dependencies as a graph and finds the optimized loop pipelining through traversing the combinations of the access and partition pattern of arrays. However, the proposed method only can deal with static dependencies between loops and arrays.

In [89], an open-source program optimizer named SOAP3 is proposed, which can automatically rewrite a given program to alleviate data dependencies between loop iterations. SOAP3 takes runtime, accuracy, and area as metrics for the rewritten programs, and minimizes the latency of loops with controlled accuracy of floating-point computations. This work proposes an effective solution on resolving the loop dependencies.

Although loop unrolling improves the throughput, it can also lead to schedule additional operations, at the cost of more resource sharing, larger MUX size, and delay. A recent work[90] devotes to exploiting the best loop unrolling factor in a behavioral specification, which uses design space pruning methods to predict controller and datapath delays. However, the proposed method only can deal with loops without dependency.

Since it is challenging to analyze the loops with unbalanced workload, irregular dependence patterns, and irregular memory accesses, most of the HLS tools implement the pipeline of the loops conservatively, which sacrifices performance for maintaining presumed regularity. The authors of [91] expanded loop pipeline with dynamic scheduling to adapt to data-dependent behaviors, while employing static compile-time optimizations to minimize the hardware overhead associated with runtime optimization. Another work in [92] addresses this problem based on application-specific dynamic hazard resolution. The method first generates an aggressive pipeline at compile-time, and then resolves hazards with memory port arbitration and squash-and-replay at run-time. These two studies propose methods on improving the performance of the pipelines.

4.3.2 *Memory Space Allocation*

In order to enhance the storage capability, FPGAs are equipped with multiple memory banks in the form of distributed block RAMs (BRAMs), which can speed up memory accesses at low latency via partitioning and mapping software data structures onto dedicated BRAMs. Consequently, multiple memory operations can be scheduled in one cycle if they access different data in the same cycle without confliction. BRAMs expand the available parallelism. However, the customization of memory accesses should be carefully planned due to the limited memory ports of BRAMs.

There are four aspects that need to be considered for memory allocation. We define them as the design space of memory:

$$DS_{\text{mem}} = (BW, BD, FO, LT),$$

where BW is the bit width, BD is the bandwidth, FO is the fanout, and LT is the latency. Bit width is decided by the data type of the algorithms. If the precision meets the requirements of the algorithms, a smaller bit width is preferred. Bandwidth mainly varies according to the degree of parallelism. Fanout is defined as the number of connections of the same storage units, and its value is decided by the number of read/write operations to the same variable at the same time. There is a fanout limitation for the storage units. If the connections exceed the limitation, the routing algorithms use LUTs to expand the connections. This costs additional FPGA resource. Therefore, the read/write of the variables in the algorithm needs elaboration. By taking the BRAMs as the storage units, there will be an additional delay cycle for writing. This must be considered during the time sequential design.

A generalized memory-partitioning framework for high data throughput of on-chip memories is proposed in [93]. The framework generalizes cyclic partitioning into block-cyclic partitioning to expand the design space. In addition, the framework also provides a conflict detection algorithm on polytope emptiness testing, and uses integer points counting in polytope for intra-bank offset generation. According to the experimental results, the framework can save more BRAMs while achieving comparable throughput. This work is the first to use a polyhedral model to formulate and solve the bank access conflict problem, which is a novel approach for improving the bandwidth of BRAMs.

In [94], the authors proposed a memory optimization method on improving the area efficiency of the memory components. First, they constructed a memory controller to schedule the data accesses of all the processes on the same physical memories. Second, they designed an algorithm to automatically determine an optimal architecture of the memory subsystem under a formal description of the data exchanges among the accelerator processes. Once the description contains multiple accelerators which operate in time-multiplexing, the algorithm derives opportunities to enforce the sharing of physical memories among the accelerator processes to reduce the overall memory footprint. The proposed method transforms the memory optimization problem into the DSE problem. However, the work does not propose an efficient DSE strategy.

LegUp[95] is an HLS tool which permits synthesizing multi-threaded software into parallel hardware. In order to promote the memory bandwidth of the parallelized threads, the authors of [96] designed an approach which can automatically partition the arrays of the multi-threaded programs into sub-arrays. This approach is accomplished with trace-based profiling, and allocates dedicated memories to each thread, which reduces memory access contention and arbitration. However, the work focuses less on memory reuse, and the efficiency of memory utilization can be further improved.

It will be another difficulty for HLS tools if dynamic memory allocation or dynamic, pointer-based data structures are involved in applications. The authors of [97] tackled this difficulty in two steps. First, they performed a static analysis for pointer-manipulating programs where heap-allocated data structures are automatically split into disjoint, independent regions. Second, they designed an algorithm to automatically transform source code for loop parallelization and memory partitioning using off-the-shelf HLS tools. However, in order to avoid the memory access arbitration, the automatic transformation algorithm decreases the performance of the algorithms.

4.3.3 Optimizations of Floating-Point Operations

Floating-point operations are common in algorithms, which are resource-intensive and operational order sensitive. The high-level languages used by HLS tools inherit the limitations when they are used for general purpose processors, such as bit width (32 or 64 bits only), established standards (C11 or IEEE-754).

In order to handle floating-point operations in a completely non-standard way, the work in [98] presents an attempt on generating efficient designs. The compliance with the C11 and IEEE-754 standards is replaced by the restrict of a high-level accuracy specification. According to the experiments on floating-point summation-reduction pattern, the proposed work optimizes both accuracy and latency by an order of magnitude for comparable area.

In [11], the authors exhibited the performance of the C language described floating-point cores provided in LegUp, which can be customized to non-compliant variants with superior performance and area features, for instance, reduced-precision floating point, or cores without full IEEE 754 exceptions support. Compared with the optimized RTL FP cores, the software-specified HLS-generated cores are close to them in terms of area/performance.

Both of the HLS tools implement limited operators for floating-point data type, which need to be enriched in the future work.

4.3.4 Power Efficient Improvements

Power consumption is a very important metric for the digital circuits, which is mainly proportional to the complexity of the algorithms. We express the relation among power consumption affecting factors as follows:

$$PC = Fre \times N \times p,$$

where PC is the power consumption of FPGAs designs, Fre is working frequency, N is the number of working units (LUTs, DSPs, BRAMs, IOs, etc.) during power calculation, and p is the average power consumption per unit.

There are also special efforts in the community to reduce the power consumption of the circuits generated by HLS tools.

In [99], a centralized and fine-grained microarchitecture-level clock gating method is proposed for lowering the power of the designs generated by HLS tools. The method mainly utilizes the existing signals of finite state machine to control the datapath clock network, where only the clock sub-tree of the current state is enabled and the other clock sub-trees are disabled.

The authors of [100] provided an automated Low Power-High Level Synthesis (LP-HLS) methodology, which aims to integrate low power techniques, specifically power shut-off (PSO), within a model-based hardware flow. LP-HLS realizes automatic low power design mainly through deriving power intent information using set of pragmas and a directive file.

[99] and [100] lower the power-consumption through dynamically disabling the unused elements of FPGAs chips, which also decrease the stability of the FPGAs.

4.3.5 Benchmark Functions

Standard benchmarks are of great importance while quantitatively evaluating the new emerged ideas and algorithms. Communities need benchmark functions to facilitate comparisons between tools, evaluate and stress-test new synthesis techniques, and establish meaningful performance baselines to track progress of the HLS technology.

A C-based benchmark library, named CHStone, is proposed in [101]. There are 12 programs in CHStone, including multiple application fields, e.g., media processing, arithmetic, microprocessor, security, and so on. CHStone is now widely used by the HLS communities [46–48].

S2CBench, a benchmark suite compiled with SystemC language, is introduced in [102]. There are four main features in S2CBench: programs selected from multiple application fields, specific test methods for the optimization of the programs, easy comparison of QoR among HLS tools, and completeness test of the HLS tools.

Different from the previous benchmarks suits which are primarily comprised of small textbook-style function kernels, Rosetta [103] provides fully-developed com-

plex applications. These applications are associated with realistic performance constraints, and optimized by advanced features of modern HLS tools.

All of the proposed benchmark functions focus on evaluating the parallel capability of FPGAs, and do not provide good test cases for exploring the pipeline capability of FPGAs.

4.4 Summary of the Performance Optimization Work of HLS Tools

In this subsection, we summarize the contributions and insufficiencies of the performance optimization work of HLS tools in Table 2. With this table, we hope to provide a perspective of the performance op-

Table 2. Summary of the Performance Optimization Work of HLS Tools

Criteria Level	Sub-Metric	Work	Contribution	Insufficiency
Ease of use	Optimizations of the HLS Flow	Kuga *et al.* [40], Josipovic *et al.* [42], Kastner *et al.* [43]	Providing higher design abstract level with DSL and templates, etc; therefore, designers can consider less on hardware structures	The methods are only dedicated for problems of specific modes, e.g., problems with summation
	Verification & debugging	Doucet and Kurshan [44], Calagar *et al.* [45], Campbell *et al.* [46], Yang *et al.* [48]	Transplanting high-level language debugging approaches (tools and modes) to hardware debugging to improve FPGAs debugging efficiency	The methods debugs only functions, without timing; the final hardware functions have to be confirmed before deployment due to the insertion of hardware debugging codes
	Libraries	Schmid *et al.* [49], Özkan *et al.* [50], Licht *et al.* [51]	Providing library functions for common application domains (graphics and maths, etc.); therefore, designers pay less attention to the details of the implementations	The quality of circuits synthesized from library functions needs to be further improved
Development cycles optimization	Design space exploration	Moullec *et al.* [55], Bilavarn *et al.* [56], Liu and Schäfer [62], Ferretti *et al.* [63], Schafer *et al.* [67], Liu *et al.* [68]	Automating the process of exploring the optimal solution under the user defined performance targets	Before the automatic process starts, designers have to do many preparation works manually; most of the proposed methods cost long running time.
	Optimizations for applications of particular fields	Schmid *et al.* [69], Li *et al.* [70], Matai *et al.* [72], Minutoli *et al.* [73]	Improving the performance (bandwidth, throughput, etc.) for specific problems (graphic, sorting, etc.)	The resource costs caused by the performance improvements are not assessed in the work
	Optimizations of scheduling	Garibotti *et al.* [77], Garibotti *et al.* [78], Josipovic *et al.* [79]	Improving scheduling through dynamic dependency analysis	The performance of dynamic dependency analysis algorithms needs to be improved to decrease the time cost
	Trade-off between performance and resource	Li *et al.* [80], Choi *et al.* [81], Dai *et al.* [84]	Achieving the trade-off through optimizing the results of the high-level language compilers, reducing the initiation intervals of loops, and saving routing resource	Only one of the resources (timing or routing) can be traded off, even with the cost of other kind of resource (storage)
Promotion on specific metrics	Loop optimization	Liu *et al.* [87], Pham *et al.* [88], Gao *et al.* [89], Panda *et al.* [90]	Improving pipeline and unrolling of loops through optimizing loop-carried dependency and data dependency	The codes have to be rewritten, which may affect the accuracy of the algorithms
	Benchmark functions	Hara *et al.* [101], Schafer and Mahapatra [102]	Providing baselines for comparison between tools, evaluation of new synthesis technologies	The proposed benchmark functions mainly focus on the parallel capability of HLS, but less on other capabilities
	Power efficient improvement	Alam *et al.* [99], Qamar *et al.* [100]	Lowering the power consumption through dynamically disabling the unused elements of FPGAs chips	The running stability of the FPGAs is decreased
	Memory space allocation	Wang *et al.* [93], Chen and Anderson [96], Winterstein [97]	Improving memory usage by decomposing memory access and enhancing memory reuse	The decomposing and reuse should be accomplished by code conversion, which may affect the accuracy of the algorithms
	Optimizations for floating-point operations	Uguen *et al.* [98], Zhang *et al.* [11]	Replacing standard floating-point (IEEE-754) with customized floating-point, which saves storage and computation resource	The proposed studies only implement limited floating-point operators

timization work and a reference for the future work of readers.

5 Discussion

In the common processor (e.g., CPUs and GPUs) programming field, the high-level language gradually replaces the assembly language along with the growth of the scale and complexity of the software. Taking this as a reference, the FPGA community considers that HLS is a promising way to resolve the problems in FPGA design. However, there are still research challenges remaining open.

First of all, the high-level languages are designed for procedure description on processors with instruction systems, whereas the FPGA designs are descriptions of circuit structures. This make the designers still have to understand the rules of hardware design and to learn how to describe hardware structures with high-level language. Most of the designers start by learning the high-level languages for procedure description, and it is hard for them to change their thinking habits.

Secondly, different from the compilers of the common processors, the compilers of HLS cannot completely take over the optimization of the designs. Most of the HLS tools provide directives for the users to manually direct the optimizing process, which makes the optimization process time-consuming and highly depend on the hardware experience of the designers. Even worse, the designers have to reconstruct the programs to match the rescheduling rules, which helps the synthesizer to perform better during feature extraction. Due to the lack of rules documents, the program reconstruction is again converted to a DSE problem.

According to our survey of the literature, the HLS community has contributed a lot on the performance optimization of the HLS tools, for instance, making the exploring of the design space faster and automatically (discussed in Subsection 4.2.1), establishing libraries to improve the code reuse (discussed in Subsection 4.1.3), improving the efficiency of verification and debugging (discussed in Subsection 4.1.2), exploring better parallelism approaches (discussed in Subsection 4.3.1, Subsection 4.3.2, and Subsection 4.2.3), and providing predefined programming templates (discussed in Subsection 4.1.1). In addition, most of the academic tools are source open, and the commercial tools strive to provide richer documents. The QoR of the HLS tools has been significantly improved with the newest generation of HLS tools.

Based on our research on HLS and FPGA design, we also provide some advice on the future optimization of HLS tools.

First, the performance optimization of HLS tools should not only focus on the performance of the generated circuits, such as the area, the latency, and the throughput, but also consider the ease of use of the HLS tools, for example, automatic code generation, automatic test framework generation, and more visible debugging. After all, programming takes up most of the HLS design.

Second, in order to decrease the requirements of hardware knowledge for designers, it is better to draw an analogy between the HLS design process and the embedded system design process (e.g., system design on ARM-based systems[16]). The embedded system design tools normally integrate the hardware operations into open source application programming interfaces (APIs) packages. The designers specify functionalities by the APIs without consideration on hardware structures, so that they can focus on the algorithms [104]. For fine-grained operations, designers can operate the hardware directly by the instructions, or modify the source code of the libraries to match the functionality and performance requirements. In the future generation of HLS tools, similar ideas can be offered. API packages are provided through open source libraries. Designers with less hardware knowledge can complete FPGAs designs in a low time cost. Different from the previous libraries [105], the APIs should be realized in RTL languages to guarantee the performance. If the performance optimization is needed, the designers can get into lower level operations and resource management through the open source of the libraries. In this way, more resilient HLS tools will be provided.

Finally, FPGAs virtual machines can be provided, which provide unified programming models. The virtual machines should be a higher level abstraction of the processing abilities of FPGAs, consisting of customizable parallelized or pipelined processing units. With the virtual machines, designers turn their attentions to the unified programming models, and implement the algorithms from the perspective of parallel computing or heterogeneous computing [106, 107]. The FPGA overlay discussed in Subsection 2.4 is considered as a kind of FPGA virtual machines. However, the abstraction level

[16]http://infocenter.arm.com/help/index.jsp, Dec. 2018.

is not enough. The kernels still have to be pre-designed by the designers.

6 Conclusions

In this paper, we proposed a survey of literature on the performance improvement of HLS tools. First of all, a set of three-level evaluation criteria was proposed, which includes the ease of use of the HLS tools, development cycles optimization, and promotion on specific metrics. The literature was then classified according to the evaluation criteria. After a deep analysis of the literature, we found that the main work of optimization of HLS tools is focusing on improving the QoR. Insufficient attention has been paid to improve the ease of use of the HLS tools. In order to propose our advice on the future optimization of HLS tools, we discussed the challenge of the HLS tools, and drew an analogy between the HLS and the embedded system in the design process. In the end, we suggested that more elastic HLS methodology based on open source libraries should be proposed.

References

[1] DiCecco R, Lacey G, Vasiljevic J, Chow P, Taylor G, Areibi S. Caffeinated FPGAs: FPGA framework for convolutional neural networks. In *Proc. the 2016 International Conference on Field-Programmable Technology*, December 2016, pp.265-268.

[2] Ahmed E, Rose J. The effect of LUT and cluster size on deep-submicron FPGA performance and density. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2004, 12(3): 288-298.

[3] Coussy P, Adam M. High-level Synthesis: From Algorithm to Digital Circuits. Springer, 2008.

[4] Jain A, Fahmy S A, Maskell D L. Efficient overlay architecture based on DSP blocks. In *Proc. the 23rd IEEE International Symposium on Field-Programmable Custom Computing Machines*, May 2015, pp.25-28.

[5] Koch D, Beckhoff C, Lemieux G F. An efficient FPGA overlay for portable custom instruction set extensions. In *Proc. the 23rd International Conference on Field Programmable Logic and Applications*, September 2013, Article No. 43.

[6] Capalija D, Abdelrahman T S. Tile-based bottom-up compilation of custom mesh-of-functional-units FPGA overlays. In *Proc. the 24th International Conference on Field Programmable Logic and Applications*, September 2014, Article No. 79.

[7] Lin C Y, So K H. Energy-efficient dataflow computations on FPGAs using application-specific coarse-grain architecture synthesis. *ACM SIGARCH Computer Architecture News*, 2012, 40(5): 58-63.

[8] Abdelfattah M S, Han D, Bitar A *et al.* DLA: Compiler and FPGA overlay for neural network inference acceleration. In *Proc. the 28th International Conference on Field Programmable Logic and Applications*, August 2018, pp.411-418.

[9] Najem M, Bollengier T, Lann J L, Lagadec L. Extended overlay architectures for heterogeneous FPGA cluster management. *Journal of Systems Architecture: Embedded Software Design*, 2017, 78: 1-14.

[10] Jain A K, Maskell D L, Fahmy S A. Resource-aware Just-in-Time OpenCL compiler for coarse-grained FPGA overlays. arXiv:1705.02730, 2017. https://rxiv.org/abs/1705.02730, Dec. 2019.

[11] Zhang C, Li P, Sun G, Guan Y, Xiao B, Cong J. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proc. the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, February 2015, pp.161-170.

[12] Knapp D W. Behavioral Synthesis: Digital System Design Using the Synopsys Behavioral Compiler (Har/Dskt edition). Prentice Hall, 1996.

[13] Elliott J P. Understanding Behavioral Synthesis: A Practical Guide to High-Level Design. Kluwer Academic Publishers, 1999.

[14] Wolf W. A decade of hardware/software codesign. *IEEE Computer*, 2003, 36(4): 38-43.

[15] Nane R, Sima V M, Pilato C *et al.* A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2016, 35(10): 1591-1604.

[16] Najjar W A, Böhm A P W, Draper B A, Hammes J, Rinker R, Beveridge J R, Chawathe M, Ross C. High-level language abstraction for reconfigurable computing. *IEEE Computer*, 2003, 36(8): 63-69.

[17] Coussy P, Chavet C, Bomel P, Heller D, Senn E, Martin E. GAUT: A high-level synthesis tool for DSP applications. In *High-Level Synthesis: From Algorithm to Digital Circuit*, Coussy P, Morawiec A (eds.), Springer Netherlands, 2008, pp.147-169.

[18] Pilato C, Ferrandi F. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *Proc. the 23rd International Conference on Field Programmable Logic and Applications*, September 2013, Article No. 56.

[19] Nane R, Sima V M, Olivier B, Meeuws R, Yankova Y, Bertels K. DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler. In *Proc. the 22nd International Conference on Field Programmable Logic and Applications*, August 2012, pp.619-622.

[20] Cong J, Liu B, Neuendorffer S, Noguera J, Vissers K, Zhang Z. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2011, 30(4): 473-491.

[21] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. the 2nd IEEE/ACM International Symposium on Code Generation and Optimization*, March 2004, pp.75-86.

[22] Muslim F B, Ma L, Roozmeh M, Lavagno L. Efficient FPGA implementation of OpenCL high-performance computing applications via high-level synthesis. *IEEE Access*, 2017, 5: 2747-2762.

[23] Kobayashi R, Oobata Y, Fujita N, Yamaguchi Y, Boku T. OpenCL-ready high speed FPGA network for reconfigurable high performance computing. In *Proc. the International Conference on High Performance Computing in Asia-Pacific Region*, January 2018, pp.192-201.

[24] Kathail V, Aditya S, Schreiber R, Rau B R, Cronquist D C, Sivaraman M. PICO: Automatically designing custom computers. *IEEE Computer*, 2002, 35(9): 39-47.

[25] Wakabayashi K, Okamoto T. C-based SoC design flow and EDA tools: An ASIC and system vendor perspective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2000, 19(12): 1507-1522.

[26] Nikhil R. Bluespec system Verilog: Efficient, correct RTL from high level specifications. In *Proc. the 2nd ACM/IEEE International Conference on Formal Methods and MODELS for Co-Design*, June 2004, pp.69-70.

[27] Dang V, Skadron K. Acceleration of frequent itemset mining on FPGA using SDAccel and Vivado HLS. In *Proc. the 28th IEEE International Conference on Application-Specific Systems, Architectures and Processors*, July 2017, pp.195-200.

[28] Lin Y. ArchSyn: An energy-efficient FPGA high-level synthesizer [Ph.D. Thesis]. University of Hong Kong, Hong Kong, 2012.

[29] Oussama K, Naeem R, Abbes A, Khalida G, Fatima C. Design and evaluation of Vivado HLS-based compressive sensing for ECG signal analysis. In *Proc. the 16th IEEE International Conference on Dependable, Autonomic and Secure Computing, the 16th Int. Conf. Pervasive Intelligence and Computing, the 4th Int. Conf. Big Data Intelligence and Computing and the 3rd IEEE Cyber Science and Technology Congress*, August 2018, pp.457-461.

[30] Capalija D, Abdelrahman T S. A high-performance overlay architecture for pipelined execution of data flow graphs. In *Proc. the 23rd International Conference on Field Programmable Logic and Applications*, September 2013, Article No. 119.

[31] Lin C Y, Jiang Z H, Fu C, So K H, Yang H. FPGA high-level synthesis versus overlay: Comparisons on computation kernels. *ACM SIGARCH Computer Architecture News*, 2017, 44(4): 92-97.

[32] Neuendorffer S, Martinez-Vallina F. Building zynq® accelerators with Vivado® high level synthesis. In *Proc. the 2013 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, February 2013, pp.1-2.

[33] Nane R, Sima V M, Pilato C *et al.* A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2016, 35(10): 1591-1604.

[34] Farhat W, Sghaier S, Faiedh H, Souani C. Design of efficient embedded system for road sign recognition. *Journal of Ambient Intelligence & Humanized Computing*, 2019, 10(2): 491-507.

[35] Goeders J, Wilton S J E. Allowing software developers to debug HLS hardware. arXiv: 1508.06805, 2015. https://arxiv.org/abs/1508.06805, Dec. 2019.

[36] Lahti S, Sjovall P, Vanne J, Hämäläinen T D. Are we there yet? A study on the state of high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018, 38(5): 898-911.

[37] Schäfer B C. Enabling high-level synthesis resource sharing design space exploration in FPGAs through automatic internal bitwidth adjustments. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2017, 36(1): 97-105.

[38] Mathieson K, Keil M. Beyond the interface: Ease of use and task/technology fit. *Information & Management*, 1998, 34(4): 221-230.

[39] Lo C, Chow P. Model-based optimization of high level synthesis directives. In *Proc. the 26th International Conference on Field Programmable Logic and Applications*, August 2016, Article No. 60.

[40] Kuga M, Fukuda K, Amagasaki M, Iida M, Sueyoshi T. High-level synthesis based on parallel design patterns using a functional language. In *Proc. the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, June 2017, Article No. 23.

[41] Mori J Y, Werner A, Fricke F, Hüebner M. A rapid prototyping method to reduce the design time in commercial high-level synthesis tools. In *Proc. the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops*, May 2016, pp.253-258.

[42] Josipovic L, George N, Ienne P. Enriching C-based high-level synthesis with parallel pattern templates. In *Proc. the 2016 International Conference on Field-Programmable Technology*, December 2016, pp.177-180.

[43] Kastner R, Matai J, Neuendorffer S. Parallel programming for FPGAs. arXiv:1805.03648, 2018. https://arxiv.org/abs/1805.03648, Dec. 2019.

[44] Doucet F, Kurshan R. A methodology to take credit for high-level verification during RTL verification. *Formal Methods in System Design*, 2017, 51(2): 395-418.

[45] Calagar N, Brown S D, Anderson J H. Source-level debugging for FPGA high-level synthesis. In *Proc. the 24th International Conference on Field Programmable Logic and Applications*, September 2014, Article No. 124.

[46] Campbell K A, Lin D, Mitra S, Chen D. Hybrid quick error detection (H-QED): Accelerator validation and debug using high-level synthesis principles. In *Proc. the 52nd Annual Design Automation Conference*, June 2015, Article No. 53.

[47] Monson J S, Hutchings B L. Using source-level transformations to improve high-level synthesis debug and validation on FPGAs. In *Proc. the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, February 2015, pp.5-8.

[48] Yang L, Ikram M, Gurumani S, Fahmy S A, Chen D, Rupnow K. JIT trace-based verification for high-level synthesis. In *Proc. the 2015 International Conference on Field Programmable Technology*, December 2016, pp.228-231.

[49] Schmid M, Apelt N, Hannig F, Teich J. An image processing library for C-based high-level synthesis. In *Proc. the 24th International Conference on Field Programmable Logic and Applications*, September 2014, Article No. 47.

[50] Özkan M A, Reiche O, Hannig F, Teich J. A highly efficient and comprehensive image processing library for C++-based high-level synthesis. In *Proc. the 4th International Workshop on FPGAs for Software Programmers*, August 2017, pp.2-10.

[51] Licht J D F, Blott M, Hoefler T. Designing scalable FPGA architectures using high-level synthesis. In *Proc. the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, February 2018, pp.403-404.

[52] Matai J, Lee D, Althoff A, Kastner R. Composable, parameterizable templates for high-level synthesis. In *Proc. Design, Automation & Test in Europe Conference & Exhibition*, August 2016, pp.744-749.

[53] Kneuper R. Sixty years of software development life cycle models. *IEEE Annals of the History of Computing*, 2017, 39(3): 41-54.

[54] Yadav H B , Yadav D K. Defects prediction of early phases of software development life cycle using fuzzy logic. In *Proc. the 4th International Conference on Confluence: The Next Generation Information Technology Summit*, Sept. 2013, pp.2-6.

[55] Moullec Y L, Diguet J P, Gourdeaux T, Philippe J L. Design-Trotter: System-level dynamic estimation task a first step towards platform architecture selection. *Journal of Embedded Computing*, 2005, 1(4): 565-586.

[56] Bilavarn S, Gogniat G, Philippe J L, Bossuet L. Design space pruning through early estimations of area/delay tradeoffs for FPGA implementations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2006, 25(10): 1950-1968.

[57] Sun W, Wirthlin M J, Neuendorffer S. FPGA pipeline synthesis design exploration using module selection and resource sharing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2007, 26(2): 254-265.

[58] Schafer B C, Wakabayashi K. Machine learning predictive modeling high-level synthesis design space exploration. *IET Computers & Digital Techniques*, 2012, 6(3): 153-159.

[59] Zhong G, Prakash A, Liang Y, Mitra T, Niar S. Lin-Analyzer: A high-level performance analysis tool for FPGA-based accelerators. In *Proc. the 53rd Annual Design Automation Conference*, June 2016, Article No. 136.

[60] Prost-Boucle A, Muller O, Rousseau F. Fast and standalone design space exploration for high-level synthesis under resource constraints. *Journal of Systems Architecture*, 2014, 60(1): 79-93.

[61] Schäfer B C. Probabilistic multi knob high-level synthesis design space exploration acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2016, 35(3): 394-406.

[62] Liu D, Schäfer B C. Efficient and reliable high-level synthesis design space explorer for FPGAs. In *Proc. the 26th International Conference on Field Programmable Logic and Applications*, August 2016, Article No. 72.

[63] Ferretti L, Ansaloni G, Pozzi L. Cluster-based heuristic for high level synthesis design space exploration. *IEEE Transactions on Emerging Topics in Computing*. doi:10.1109/TETC.2018.2794068.

[64] Ferrandi F, Lanzi P L, Loiacono D, Pilato C, Sciuto D. A multi-objective genetic algorithm for design space exploration in high-level synthesis. In *Proc. IEEE Computer Society Symposium on VLSI*, April 2008, pp.417-422.

[65] Sengupta A, Sedaghat R. Integrated scheduling, allocation and binding in high level synthesis using multi structure genetic algorithm based design space exploration. In *Proc. the 12th International Symposium on Quality Electronic Design*, March 2011, pp.486-494.

[66] Ram D S H, Bhuvaneswari M C, Logesh S M. A novel evolutionary technique for multi-objective power, area and delay optimization in high level synthesis of datapaths. In *Proc. IEEE Computer Society Symposium on VLSI*, July 2011, pp.290-295.

[67] Schafer B C, Takenaka T, Wakabayashi K. Adaptive simulated annealer for high level synthesis design space exploration. In *Proc. the 2009 International Symposium on VLSI Design, Automation and Test*, April 2009, pp.106-109.

[68] Liu C, Zhao Q, Yan B, Elsayed S M, Ray T, Sarker R A. Adaptive sorting-based evolutionary algorithm for many objective optimization. *IEEE Transactions on Evolutionary Computation*, 2019, 23(2): 247-257.

[69] Schmid M, Reiche O, Hannig F, Teich J. Loop coarsening in C-based high-level synthesis. In *Proc. the 26th IEEE International Conference on Application-Specific Systems, Architectures and Processors*, July 2015, pp.166-173.

[70] Li C, Bi Y, Benezeth Y, Ginhac D, Yang F. High-level synthesis for FPGAs: Code optimization strategies for real-time image processing. *Journal of Real-Time Image Processing*, 2017, 27(9): 31-42.

[71] Li C, Bi Y, Benezeth Y, Ginhac D, Yang F. Fast FPGA prototyping for real-time image processing with very high level synthesis. *Journal of Real-Time Image Processing*, 2017, 27(9): 1-18.

[72] Matai J, Richmond D, Lee D, Blair Z, Wu Q, Abazari A, Kastner R. Resolve: Generation of high-performance sorting architectures from high-level synthesis. In *Proc. the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, February 2016, pp.195-204.

[73] Minutoli M, Castellana V G, Tumeo A, Lattuada M, Ferrandi F. Enabling the high level synthesis of data analytics accelerators. In *Proc. the 11th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, October 2016, Article No. 15.

[74] Liang M, Lavagno L, Lazarescu M T, Arif A. Acceleration by inline cache for memory-intensive algorithms on FPGA via high-level synthesis. *IEEE Access*, 2017, 5: 18953-18974.

[75] Hadjis S, Canis A, Sobue R, Hara-Azumi Y, Tomiyama H, Anderson J. Profiling-driven multi-cycling in FPGA high-level synthesis. In *Proc. the 2015 Design, Automation & Test in Europe Conference & Exhibition*, March 2015, pp.31-36.

[76] Zhao R, Tan M, Dai S, Zhang Z. Area-efficient pipelining for FPGA-targeted high-level synthesis. In *Proc. the 52nd Annual Design Automation Conference*, June 2015, Article No. 157.

[77] Garibotti R, Reagen B, Shao Y S, Wei G Y, Brooks D. Using dynamic dependence analysis to improve the quality of high-level synthesis designs. In *Proc. IEEE International Symposium on Circuits and Systems*, May 2017.

[78] Garibotti R, Reagen B, Shao Y S, Wei G, Brooks D. Assisting high-level synthesis improve SpMV benchmark through dynamic dependence analysis. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2018, 65-II(10): 1440-1444.

[79] Josipovic L, Ghosal R, Ienne P. Dynamically scheduled high-level synthesis. In *Proc. the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, February 2018, pp.127-136.

[80] Li P, Zhang P, Pouchet L N, Cong J. Resource-aware throughput optimization for high-level synthesis. In *Proc. the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, February 2015, pp.200-209.

[81] Choi J, Brown S, Anderson J. Resource and memory management techniques for the high-level synthesis of software threads into parallel FPGA hardware. In *Proc. the 2015 International Conference on Field Programmable Technology*, December 2015, pp.152-159.

[82] Zhao R, Liu G, Srinath S, Batten C, Zhang Z. Improving high-level synthesis with decoupled data structure optimization. In *Proc. the 53rd Annual Design Automation Conference*, June 2016, Article No. 137.

[83] Huang Q, Lian R, Canis A, Choi J, Xi R, Calagar N, Brown S, Anderson J. The effect of compiler optimizations on high-level synthesis-generated hardware. *ACM Transactions on Reconfigurable Technology & Systems*, 2015, 8(3): Article No. 14.

[84] Dai S, Zhou Y, Zhang H, Ustun E, Young E F, Zhang Z. Fast and accurate estimation of quality of results in high-level synthesis with machine learning. In *Proc. the 26th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, April 2018, pp.129-132.

[85] Zhong G, Venkataramani V, Liang Y, Mitra T, Niar S. Design space exploration of multiple loops on FPGAs using high level synthesis. In *Proc. the 32nd IEEE International Conference on Computer Design*, October 2014, pp.456-463.

[86] Li P, Pouchet L, Cong J. Throughput optimization for high-level synthesis using resource constraints. In *Proc. the 4th International Workshop on Polyhedral Compilation Techniques*, January 2014.

[87] Liu J, Wickerson J, Constantinides G A. Loop splitting for efficient pipelining in high-level synthesis. In *Proc. the 24th IEEE International Symposium on Field-Programmable Custom Computing Machines*, May 2016, pp.72-79.

[88] Khanh P N, Singh A K, Kumar A, Aung K M K. Exploiting loop-array dependencies to accelerate the design space exploration with high level synthesis. In *Proc. the 2015 Design, Automation & Test in Europe Conference & Exhibition*, March 2015, pp.157-162.

[89] Gao X, Wickerson J, Constantinides G A. Automatically optimizing the latency, area, and accuracy of C programs for high-level synthesis. In *Proc. the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, February 2016, pp.234-243.

[90] Panda P R, Sharma N, Kurra S, Bhartia K A, Singh N K. Exploration of loop unroll factors in high level synthesis. In *Proc. the 31st International Conference on VLSI Design and the 17th International Conference on Embedded Systems*, January 2018, pp.465-466.

[91] Dai S, Liu G, Zhao R, Zhang Z. Enabling adaptive loop pipelining in high-level synthesis. In *Proc. the 51st Asilomar Conference on Signals, Systems, and Computers*, October 2017, pp.131-135.

[92] Dai S, Zhao R, Liu G, Srinath S, Gupta U, Batten C, Zhang Z. Dynamic hazard resolution for pipelining irregular loops in high-level synthesis. In *Proc. the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, February 2017, pp.189-194.

[93] Wang Y, Li P, Cong J. Theory and algorithm for generalized memory partitioning in high-level synthesis. In *Proc. the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, February 2014, pp.199-208.

[94] Pilato C, Mantovani P, Guglielmo G D, Carloni L P. System-level memory optimization for high-level synthesis of component-based SoCs. In *Proc. the 2014 International Conference on Hardware/Software Codesign & System Synthesis*, October 2014, Article No. 18.

[95] Canis A, Choi J, Aldham M, Zhang V, Kammoona A, Czajkowski T, Brown S D, Anderson J H. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems*, 2013, 13(2): Article No. 24.

[96] Chen Y T, Anderson J H. Automated generation of banked memory architectures in the high-level synthesis of multi-threaded software. In *Proc. the 27th International Conference on Field Programmable Logic and Applications*, September 2017, Article No. 99.

[97] Winterstein F J. Separation Logic for High-level Synthesis. Springer, 2017.

[98] Uguen Y, de Dinechin F, Derrien S. Bridging high-level synthesis and application-specific arithmetic: The case study of floating-point summations. In *Proc. the 27th International Conference on Field Programmable Logic and Applications*, September 2017, Article No. 38.

[99] Alam M R, Salehi M, Fakhraie S M. Power efficient high level synthesis by centralized and fine-grained clock gating. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2015, 34(12): 1954-1963.

[100] Qamar A, Muslim F B, Iqbal J, Lavagno L. LP-HLS: Automatic power-intent generation for high-level synthesis based hardware implementation flow. *Microprocessors & Microsystems*, 2017, 50: 26-38.

[101] Hara Y, Tomiyama H, Honda S, Takada H, Ishii K. CHStone: A benchmark program suite for practical C-based high-level synthesis. In *Proc. the 2008 IEEE International Symposium on Circuits and Systems*, May 2008, pp.1192-1195.

[102] Schäfer B C, Mahapatra A. S2CBench: Synthesizable SystemC benchmark suite for high-level synthesis. *IEEE Embedded Systems Letters*, 2014, 6(3): 53-56.

[103] Zhou Y, Gupta U, Dai S *et al*. Rosetta: A realistic high-level synthesis benchmark suite for software programmable FPGAs. In *Proc. the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, February 2018, pp.269-278.

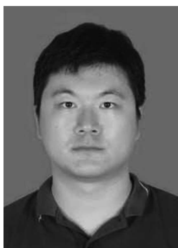
[104] Rodríguez A, Valverde J, Portilla J, Otero A, Riesgo T, de la Torre E. FPGA-based high-performance embedded systems for adaptive edge computing in cyber-physical systems: The ARTICo^3 framework. *Sensors*, 2018, 18(6): 1877.

[105] O'Loughlin D, Coffey A, Callaly F, Lyons D, Morgan F. Xilinx Vivado high level synthesis: Case studies. In *Proc. the 25th IET Irish Signals & Systems Conference and 2014 China-Ireland International Conference on Information and Communities Technologies*, June 2014, pp.352-356.

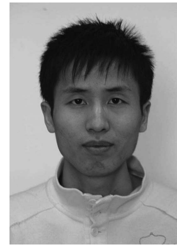
[106] Beaumont O, Becker B A, DeFlumere A M, Eyraud-Dubois L, Lambert T, Lastovetsky A L. Recent advances in matrix partitioning for parallel computing on heterogeneous platforms. *IEEE Transactions on Parallel & Distributed Systems*, 2017, 59(99): 218-229.

[107] Vesper M, Koch D, Vipin K, Fahmy S A. JetStream: An open-source high-performance PCI Express 3 streaming library for FPGA-to-Host and FPGA-to-FPGA communication. In *Proc. the 26th International Conference on Field Programmable Logic and Applications*, August 2016, Article No. 36.

Lan Huang is currently a professor and a supervisor of Ph.D. candidates at Jilin University, Changchun. She received her B.S., M.S. and Ph.D. degrees in computer science and technology from Jilin University, Changchun, in 1994, 1999, and 2003 respectively. She is mainly engaged in intelligent computing, data mining theory and application research, and high-performance computing. She is a distinguished member of CCF.



Da-Lin Li received his B.S. and M.S. degrees in mechatronic engineering from the Liaoning Technical University, Fuxin, in 2005 and 2008 respectively. He is currently pursuing his Ph.D. degree in computer science from the College of Computer Science and Technology, Jilin University, Changchun. His current research interests include high-performance computing, swarm intelligence and machine learning.



Kang-Ping Wang received his B.S., M.S. and Ph.D. degrees in computer science and technology from Jilin University, Changchun, in 2000, 2003 and 2008 respectively. He is now a faculty of the College of Computer Science and Technology at Jilin University, Changchun. In past several years, his research interest includes heterogeneous computing and deep learning.

Teng Gao received his B.S. degree in computer science and technology from China University of Geosciences, Changchun, in 2018. He is currently a Master student in Jilin University, Changchun. His research interest is FPGAs.

Adriano Tavares is currently an associate professor at University of Minho, Braga. He received his B.S. degree in informatics, M.S. degree in information technology, both from University of Coimbra, Coimbra, in 1990 and 1993 respectively. He received his Ph.D. degree in industrial electronics from University of Minho, Braga, in 2000. His main research interests are embedded systems modeling and design, system software design, system-on-chip design and engineering education. He published more than 100 book chapters and papers in international conferences and journals related to embedded systems design and two books on assembly and C programming.