Modern High-Level Synthesis: improving productivity with a multi-level approach*

Serena Curzel[0000−0002−8202−1627]

Politecnico di Milano, Piazza Leonardo da Vinci 32 20133 Milano, Italy
serena.curzel@polimi.it

Abstract. High-Level Synthesis (HLS) tools simplify the design of hardware accelerators by automatically generating Verilog/VHDL code starting from a general-purpose software programming language. Because of the mismatch between the requirements of hardware descriptions and the characteristics of input languages, HLS tools still require hardware design knowledge and non-trivial design space exploration, which might be an obstacle for domain scientists seeking to accelerate applications written, for example, in Python-based programming frameworks. This research proposes a modern approach based on multi-level compiler technologies to bridge the gap between HLS and high-level frameworks, and to use domain-specific abstractions to solve domain-specific problems. The key enabling technology is the Multi-Level Intermediate Representation (MLIR), a framework that supports building reusable compiler infrastructure. The proposed approach uses MLIR to introduce new optimizations at appropriate levels of abstraction outside the HLS tool while still relying on years of HLS research in the low-level hardware generation steps; users and developers of HLS tools can thus increase their productivity, obtain accelerators with higher performance, and not be limited by the features of a specific (possibly closed-source) backend. The presented tools and techniques were designed, implemented, and tested to synthesize machine learning algorithms, but they are broadly applicable to any input specification written in a language that has a translation to MLIR. Generated accelerators can be deployed on Field Programmable Gate Arrays or Application-Specific Integrated Circuits, and they can reach high energy efficiency without any manual optimization of the code.

Keywords: High-Level Synthesis · FPGA · Machine Learning.

1 Introduction

The exponential growth of data science and machine learning (ML), coupled with the diminishing performance returns of silicon at the end of Moore's law and Dennard scaling, is leading to widespread interest in domain-specific architectures and accelerators [16]. Field Programmable Gate Arrays (FPGAs) and

* This research was partially supported by Pacific Northwest National Laboratory's DMC and ATSCALE Laboratory-Directed R&D Initiatives.

Application-Specific Integrated Circuits (ASICs) can provide the necessary hardware specialization with higher performance and energy efficiency than multi-core processors or Graphic Processing Units (GPUs). ASICs are the best solution in terms of performance, but they incur higher development costs; FPGAs are more accessible and can be quickly reconfigured, allowing to update accelerators according to the requirements of new applications or to try multiple configurations in a prototyping phase before committing to ASIC manufacturing.

ASICs and FPGAs are designed and programmed through hardware description languages (HDLs) such as Verilog or VHDL, which require developers to identify critical kernels, build specialized functional units and memory components, and explicitly manage low-level concerns such as clock and reset signals or wiring delays. The distance between traditional software programming and HDLs creates significant productivity and time-to-market gaps [19, 20] and traditionally required manual coding from expert hardware developers. The introduction of High-Level Synthesis (HLS) simplified this process, as HLS tools allow to automatically translate general-purpose software specifications, primarily written in C/C++, into an HDL description ready for logic synthesis and implementation [8, 7]. Thanks to HLS, developers can describe the kernels they want to accelerate at a high level of abstraction and obtain efficient designs without being experts in low-level circuit design.

Due to the mismatch between the levels of abstraction of hardware descriptions and general-purpose programming languages, HLS tools often require users to augment their input code through *pragma* annotations (i.e., compiler directives) and configuration options that guide the synthesis process, for example, towards a specific performance-area trade-off. Different combinations of pragmas and options result in accelerator designs with different latency, resource utilization, or power consumption. An exhaustive exploration of the design space requires few modifications to the input code, and it does not change the functional correctness of the algorithm, but it is still not a trivial process: the effect of combining multiple optimization directives can be unpredictable, and the HLS user needs a good understanding of their impact on the generated hardware.

Data scientists who develop and test algorithms in high-level, Python-based programming frameworks (e.g., TensorFlow [1] or PyTorch [18]) typically do not have any hardware design expertise: therefore, the abstraction gap that needs to be overcome is not anymore from C/C++ software to HDL (covered by mature commercial and academic HLS tools), but from Python to annotated C/C++ for HLS. The issue is exacerbated by the rapid evolution of data science and ML, as no accelerator can be general enough to support new methods efficiently, and a manual translation of each algorithm into HLS code is highly impractical.

The aim of this research is to bridge the gap between high-level frameworks and HLS through a multi-level, compiler-based approach. The key enabling technology is the Multi-Level Intermediate Representation (MLIR) [17], a reusable and extensible infrastructure in the LLVM project for the development of domain-specific compilers. MLIR allows defining specialized intermediate representations (IRs) called *dialects* to implement analysis and transformation passes at different levels of abstraction, and it can interface with multiple software programming frameworks. An MLIR-based approach is a “modern” solution to automate the design of hardware accelerators for high-level applications through HLS, as opposed to “classic” approaches that rely on hand-written template libraries [11, 14, 4].

A practical realization of the proposed approach is the SOftware Defined Architectures (SODA) Synthesizer [6, 2], an open-source hardware compiler composed of an MLIR frontend [5] and an HLS backend [15]. SODA provides an end-to-end agile development path from high-level software frameworks to FPGA and ASIC accelerators, supports the design of complex systems, and allows to introduce and explore optimizations at many different levels of abstraction, from high-level algorithmic transformations to low-level hardware-oriented ones. Translation across different levels of abstraction is performed through progressive lowering between IRs, allowing each step to leverage information gathered in other phases of the compilation. In the frontend, domain-specific MLIR dialects allow developers to work on specialized abstractions to address system-level concerns and pre-optimize the code. The integration of an open-source tool in the backend allows to exploit years of HLS research and to introduce new features in the low-level hardware generation steps when necessary. The rest of the paper will focus on the main features of SODA (Section 2) and describe the results it allowed to obtain (Section 3).

2 The SODA Synthesizer

The SODA Synthesizer (Figure 1) is an open-source, modular, compiler-based toolchain that uses a multi-level approach, able to generate optimized FPGA and ASIC accelerators for ML through MLIR and HLS. It can accept as inputs pre-trained ML models developed in a high-level framework such as TensorFlow or PyTorch and translated into an MLIR representation. The SODA frontend (SODA-OPT) provides a search and outlining methodology to automatically extract accelerator kernels and their data dependencies from the input specification; the kernels are then optimized through a set of compiler passes that can be tuned to explore different design points, while host code containing calls to the kernel functions can be compiled by a standard LLVM compiler. SODA-OPT provides a default optimization pipeline that privileges passes resulting in faster accelerators (e.g., passes that increase instruction- and data-level parallelism or remove unnecessary operations), but many others exist that can be individually enabled or disabled, such as the ones listed in Table 1. Optimized kernels are synthesized by the backend HLS tool to generate FSMD accelerators and later composed in multi-accelerator systems; when using the Bambu HLS backend, the SODA Synthesizer is fully open-source from the algorithm to the HDL description. The outputs of SODA-OPT are fully tool-agnostic LLVM IRs that do not contain anything specific to Bambu, so they can also be synthesized through recent versions of Vitis HLS [3].

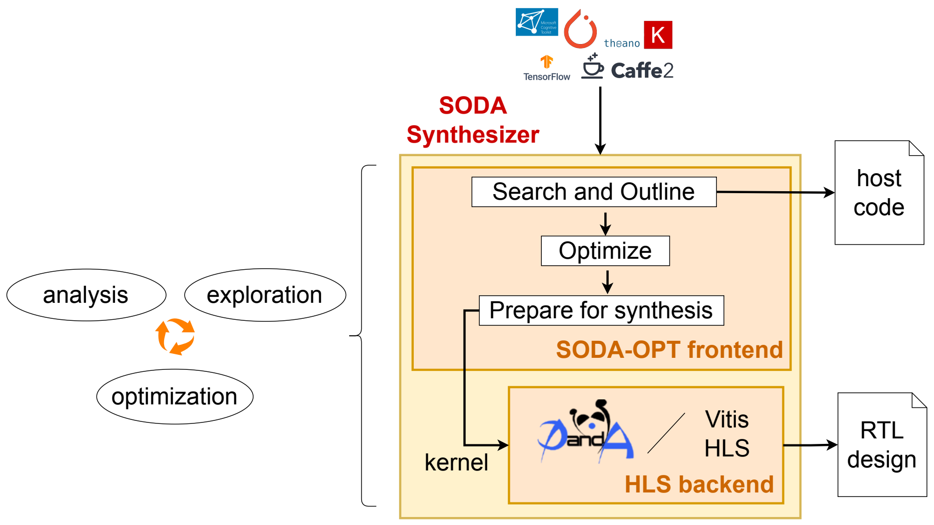


Fig. 1: The SODA Synthesizer: and end-to-end toolchain from ML algorithms to hardware accelerators through MLIR and HLS.

Table 1: Partial list of high-level optimizations available in SODA-OPT.

Optimization pass	**Effect**	**Default**
Loop unrolling	Expose instruction-level parallelism	yes
Loop tiling	Balance computation and memory transfer	no
Loop pipelining	Parallelize loop iterations	no
If-conversion	Speculative execution of if-else blocks	yes
Results forwarding	Remove unnecessary memory transfers	yes
Temporary buffer allocation	Reduce accesses to external memory	yes
Common sub-expression elimination	Remove unnecessary operations	yes

The multi-level, MLIR-based structure of the SODA Synthesizer provides ample opportunities to explore high-level compiler transformations that can improve the quality of HLS results without needing to modify the HLS tool itself [12]. Such "higher-level" optimizations can improve the performance of the generated accelerators, the portability across HLS tools (since they do not introduce tool-specific annotations or code patterns), and the productivity of users and developers: optimizations can be explored more easily and safely through compiler passes than through manual code rewriting, and there is no need to access the backend HLS code nor to be expert in low-level synthesis techniques. Moreover, dedicated MLIR dialects can be built and exploited to solve domain-specific optimization problems: for example, the `soda` dialect has been introduced to support the outlining process for accelerator kernels, and many SODA-OPT passes exploit the `affine` dialect to apply loop optimizations.

Following this approach, a new loop pipelining pass has been introduced in SODA-OPT leveraging the MLIR `affine` dialect, implementing high-level code optimizations that provide a pre-scheduled input description to HLS [13]. The `affine` dialect provides structures and methods to analyze and transform loops (in fact, it was initially introduced to support polyhedral optimizations for ML frameworks), and the higher level of abstraction allows to identify more complex dependencies than what is possible on an LLVM IR or low-level HLS IR. The proposed implementation can analyze dependencies between operations in the loop body of an `affine.for` operation and schedule them to overlap the execution of loop iterations, following standard software pipelining techniques; it can forward results from one iteration to the other, support loops with variable bounds, and speculate execution of if-else blocks.

The SODA Synthesizer also integrates a low-level synthesis methodology for the generation of complex system-on-chip (SoC) architectures composed of multiple kernels, either connected to a central microcontroller, or directly to each other in a custom dataflow architecture [9]. In fact, large and compute-intensive deep neural networks frequently represent a challenge for HLS tools, and they need to be manually broken down into smaller kernels; the issue is especially evident when the model needs to process streaming inputs in a pipelined fashion, as the complexity of the finite state machine (FSM) driving the execution becomes unmanageable. In a SoC with a central general-purpose microcontroller driving multiple accelerators, the data movement between the host microcontroller, the accelerators, and memory quickly becomes a performance bottleneck. For this reason, the SODA Synthesizer has been extended to support the generation of a second type of system: a dynamically scheduled architecture where custom accelerators are composed in a dataflow system and are driven by a distributed controller. In this architecture, multiple accelerators can perform computations in parallel on different portions of streaming input data without requiring orchestration from the host microcontroller, and can communicate with each other without going through external memory. Analysis and transformation passes in the MLIR frontend have access to high-level representations that explicitly describe the flow of data through operators and memory in a computational graph, removing the need for complex alias analysis in the HLS backend and thus simplifying the low-level generation steps.

3 Experimental results

A multi-level approach to HLS improves productivity, portability, and performance for users that want to accelerate high-level applications and do not have hardware design expertise. While productivity is not a feature that can be precisely measured, there are evident advantages when comparing the SODA Synthesizer with other state-of-the-art design flows based on HLS: unlike hls4ml [14] and FINN [4], SODA does not require to maintain a library of templated operators, so it is more easily adapted to new classes of input applications; SODA

Table 2: Execution times of accelerators optimized with different synthesis tools.

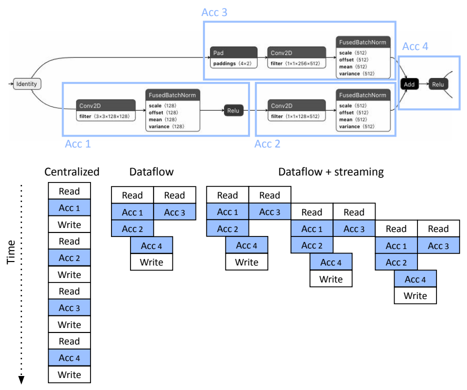
Kernel	**Backend**	**Frontend**	**2x2**	**4x4**	**8x8**	**16x16**	**Avg. speedup**
2mm	Bambu	none	176	1375	11218	87842	66.38x
		SODA-OPT	25	43	98	784	
	Vitis HLS	none	43	115	599	4239	3.67x
		SODA-OPT	26	48	106	848	
	Vivado HLS	none	162	1138	9698	75586	72.94x
		ScaleHLS	38	63	114	410	
3mm	Bambu	none	220	1743	14042	111410	35.24x
		SODA-OPT	22	40	320	2560	
	Vitis HLS	none	37	109	593	4233	3.73x
		SODA-OPT	23	45	103	824	
	Vivado HLS	none	207	1467	12723	99939	54.86x
		ScaleHLS	57	97	169	797	
gemm	Bambu	none	103	794	6538	42514	50.43x
		SODA-OPT	16	28	71	568	
	Vitis HLS	none	24	52	140	5635	6.78x
		SODA-OPT	15	29	71	259	
	Vivado HLS	none	99	669	5593	42801	43.29x
		ScaleHLS	19	27	56	Error	
syr2k	Bambu	none	99	706	4834	35650	3.82x
		SODA-OPT	19	270	1417	8835	
	Vitis HLS	none	97	367	2627	18179	4.90x
		SODA-OPT	50	159	509	1785	
	Vivado HLS	none	73	265	1089	4225	0.73x
		ScaleHLS	93	353	1665	Error	

also generates backend-agnostic low-level code, while ScaleHLS [21] focuses on extracting performance from one specific HLS tool.

Table 2 presents execution times obtained with SODA and ScaleHLS on PolyBench kernels[1], highlighting for every kernel and every input size which is the frontend/backend combination that resulted in the lowest number of clock cycles (more results are available in [5]). To avoid focusing on performance differences that derive solely from capabilities of different HLS backends, the table also reports separate baselines that are obtained without frontend optimizations. The experiments were run targeting a Xilinx Virtex7 FPGA with 100 MHz frequency; errors sometimes occurred when Verilog code generated by ScaleHLS required more resources than the ones available in the target FPGA.

Looking at absolute numbers of clock cycles, SODA outperforms ScaleHLS in 12 kernels out of 16, through either the Bambu or the Vitis HLS backend. The SODA-OPT optimization pipeline is particularly well suited to kernels with dot product or matrix multiplication structures (providing 66.38x performance increase on *2mm* and 50.43x on *gemm*); its effect is more limited, instead, on

[1] http://web.cse.ohio-state.edu/ pouchet.2/software/polybench/



ResNet50 - 100 inputs stream

Architecture	**Computation**	**Memory**
Centralized	114,610,199,200	715,263,486
Dataflow	34,677,385,627	656,320
Speedup	**3.3**	**1089.8**

MobileNetV2 - 100 inputs stream

Architecture	**Computation**	**Memory**
Centralized	1,951,706,600	372,630,149
Dataflow	625,392,266	64,345
Speedup	**3.1**	**5,791.2**

Fig. 2: Comparison between the performance of a centralized and a dataflow architecture generated by SODA for convolutional neural network models.

kernels that contain irregular loop structures such as *syr2k*. The performance improvement is generally smaller when comparing SODA-OPT for Vitis HLS against the Vitis HLS baseline, because Vitis HLS applies loop optimizations even in absence of user directives, and the optimizations introduced by SODA-OPT can provide only a slight improvement over the default ones. The optimizations introduced by ScaleHLS greatly improve accelerator performance with respect to baseline designs synthesized by Vivado HLS; however, the annotated C++ code it produces is not portable, while the MLIR-based approach of SODA does not rely on pragma annotations and generates designs that can be synthesized with different HLS backends.

The SODA Synthesizer can generate complex multi-accelerator SoC for neural networks following either a centralized or a dataflow architecture, as presented in [9]. In a centralized architecture individual accelerators are attached to a central bus and a microcontroller drives their execution; all data is stored in and retrieved from external memory. The dataflow architecture, instead, is a system that uses a distributed controller to orchestrate the execution of accelerators accessing shared memory.

Figure 2 shows, on the right, part of the computational graph of a convolutional neural network (CNN) divided into four accelerator kernels. In the centralized architecture, every accelerator communicates with its producers and consumers through external memory, so accelerator execution and memory access are serialized. In the dataflow architecture, instead, only input arguments to the first kernel and output arguments of the last one go through external memory, while intermediate results are kept in a shared on-chip memory with as many ports as there are accelerators in the system, so that the architecture can support conflict-free concurrent accelerator execution, allowing pipelined execution of streaming inputs. The table on the left of Figure 2 reports the execution time of the two architectures in terms of clock cycles, highlighting the benefits of the dataflow architecture for streaming execution of CNN models. The high cost of communicating between accelerators and external memory is reduced when accelerators can send data to each other through shared memory, and concurrent pipelined execution provides further improvements as the overall latency for streaming inputs is mostly determined by the initiation interval, i.e., the execution of the critical path. Although the accelerators could execute in parallel on different inputs also in the centralized architecture, SODA-OPT currently does not support the generation of host code with non-blocking function calls.

4 Conclusion

In the last few years, High-Level Synthesis has become an invaluable tool to simplify the development of hardware accelerators on FPGA and ASIC, providing higher and higher quality of results to users with little expertise in low-level RTL design. State-of-the-art HLS tools still expect some hardware design knowledge from users, especially when the accelerator needs to be optimized to meet tight application requirements or when different configurations need to be evaluated looking for a specific trade-off between quality metrics.

This requirement prevents widespread adoption of HLS by domain scientists that develop data science and artificial intelligence algorithms in high-level, Python-based programming frameworks. Moreover, research that aims at improving the efficiency of the HLS process itself or the quality of generated accelerators is typically limited by the expressiveness of C/C++ code and by the annotations supported by a specific, closed-source backend tool. This paper proposed to solve the two issues by coupling established HLS tools with the modern compiler infrastructure provided by the MLIR framework, in order to improve the automated synthesis process of accelerators for high-level applications. Such an approach allows seamless integration with high-level ML frameworks, encourages the introduction of innovative optimization techniques at specific levels of abstraction, and can exploit multiple state-of-the-art HLS tools in the backend.

The proposed design flow allows to implement and apply high-level optimizations before HLS, as compiler passes supported by dedicated MLIR abstractions (dialects); such an approach can improve productivity, performance, and portability of optimizations. Loop pipelining has been used as an example of the intrinsic optimization potential in a multi-level design and optimization flow, and it has been seamlessly integrated into the SODA Synthesizer frontend. The availability of multiple levels of abstraction and domain-specific representations opens the door to new possibilities to study and implement innovative design automation methods, ranging from the exploration of techniques that can benefit HLS when applied at a high level of abstraction to the introduction of new synthesis methodologies and architectural models.

The proposed multi-level approach is modular and extensible by design, so different parts can be easily reused and adapted to the needs of different input applications, requirements, and research scenarios. A multi-level compiler-based framework can also adapt more easily to innovative input algorithms and hardware targets. For example, spiking neural networks are built of biologicallyinspired integrate-and-fire neurons, and they are usually mapped on analog neuromorphic hardware; a new MLIR dialect has been designed to support the synthesis of SNN models into neuromorphic components [10]. The dialect models concepts from the analog domain of spiking neurons through new types and operations that describe sequences of current spikes as lists of timestamps signaling their arrival.

Experimental results showed strengths and weaknesses of the approach, indicating possible next steps to improve the QoR of generated accelerators and the applicability of the proposed tools and techniques. Code for the tools developed in this research has been released in open-source to foster collaboration[2]; parts of them can be easily reused or integrated with future research efforts.

References

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., et al.: TensorFlow: A System for Large-Scale Machine Learning. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI). pp. 265–283 (2016)
2. Agostini, N.B., Curzel, S., Limaye, A., Amatya, V., Minutoli, M., Castellana, V.G., Manzano, J., et al.: The SODA Approach: Leveraging High-Level Synthesis for Hardware/Software Co-Design and Hardware Specialization. In: Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC). pp. 1359–1362 (2022)
3. AMD-Xilinx: Vitis HLS LLVM 2021.2 (2021), https://github.com/Xilinx/HLS
4. Blott, M., Preußer, T.B., Fraser, N.J., Gambardella, G., O'brien, K., Umuroglu, Y., et al.: FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks. ACM Transactions on Reconfigurable Technology and Systems **11**(3), 1–23 (2018)
5. Bohm Agostini, N., Curzel, S., Amatya, V., Tan, C., Minutoli, M., Castellana, V.G., Manzano, J., et al.: An MLIR-based Compiler Flow for System-Level Design and Hardware Acceleration. In: IEEE/ACM International Conference On Computer Aided Design (ICCAD). pp. 1–9 (2022)
6. Bohm Agostini, N., Curzel, S., Zhang, J.J., Limaye, A., Tan, C., Amatya, V., Minutoli, M., et al.: Bridging Python to Silicon: The SODA Toolchain. IEEE Micro **42**(5), 78–88 (2022)
7. Cong, J., Lau, J., Liu, G., Neuendorffer, S., Pan, P., Vissers, K., Zhang, Z.: FPGA HLS Today: Successes, Challenges, and Opportunities. ACM Transactions on Reconfigurable Technology and Systems **15**(4), 1–42 (2022)
8. 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 **30**(4), 473–491 (2011). https://doi.org/10.1109/TCAD.2011.2110592
9. Curzel, S., Agostini, N.B., Castellana, V.G., Minutoli, M., Limaye, A., Manzano, J., Zhang, J., Brooks, D., Wei, G.Y., Ferrandi, F., et al.: End-to-end synthesis of dynamically controlled machine learning accelerators. IEEE Transactions on Computers **71**(12), 3074–3087 (2022)

[2] https://github.com/ferrandi/PandA-bambu
https://gitlab.pnnl.gov/sodalite/soda-opt

10. Curzel, S., Agostini, N.B., Song, S., Dagli, I., Limaye, A., Tan, C., Minutoli, M., Castellana, V.G., Amatya, V., Manzano, J., Das, A., Ferrandi, F., Tumeo, A.: Automated generation of integrated digital and spiking neuromorphic machine learning accelerators. In: 2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD). pp. 1–7 (2021). https://doi.org/10.1109/ICCAD51958.2021.9643474
11. Curzel, S., Bohm Agostini, N., Tumeo, A., Ferrandi, F.: Hardware Acceleration of Complex Machine Learning Models through Modern High-Level Synthesis. In: Proceedings of the 19th ACM International Conference on Computing Frontiers. pp. 209–210 (2022)
12. Curzel, S., Jovic, S., Fiorito, M., Tumeo, A., Ferrandi, F.: Higher-Level Synthesis: experimenting with MLIR polyhedral representations for accelerator design. In: 12th International Workshop on Polyhedral Compilation Techniques (IMPACT). pp. 1–10 (2022)
13. Curzel, S., Jovic, S., Fiorito, M., Tumeo, A., Ferrandi, F.: Mlir loop optimizations for high-level synthesis: A case study. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques. p. 544–545. PACT '22, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3559009.3569688, https://doi.org/10.1145/3559009.3569688
14. Duarte, J., Han, S., Harris, P., Jindariani, S., Kreinar, E., Kreis, B., Ngadiuba, J., et al.: Fast inference of deep neural networks in FPGAs for particle physics. Journal of Instrumentation **13**(07), P07027 (2018)
15. Ferrandi, F., Castellana, V.G., Curzel, S., Fezzardi, P., Fiorito, M., Lattuada, M., Minutoli, M., Pilato, C., et al.: Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications. In: Proceedings of the 58th ACM/IEEE Design Automation Conference (DAC). pp. 1327–1330 (2021)
16. Hennessy, J., Patterson, D.: A new golden age for computer architecture: domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development. In: 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA). pp. 27–29 (2018). https://doi.org/10.1109/ISCA.2018.00011
17. Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., et al.: MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In: IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp. 2–14 (2021)
18. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., et al.: PyTorch: An Imperative Style, High-Performance Deep Learning Library. In: Proceedings of the 33rd International Conference on Neural Information Processing Systems (NeurIPS) (2019)
19. Semiconductor Industry Association: International Technology Roadmap for Semiconductors 1999 Edition (1999)
20. Truong, L., Hanrahan, P.: A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity. In: 3rd Summit on Advances in Programming Languages (SNAPL). vol. 136, pp. 7:1–7:21 (2019)
21. Ye, H., Hao, C., Cheng, J., Jeong, H., Huang, J., Neuendorffer, S., Chen, D.: ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation. In: 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA). pp. 741–755. IEEE (2022)