This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MM.2022.3178580, IEEE Micro

Bridging Python to Silicon: The SODA Toolchain

Nicolas Bohm Agostini[†‡], Serena Curzel[§‡], Jeff (Jun) Zhang[*], Ankur Limaye[‡], Cheng Tan[‡], Vinay Amatya[‡], Marco Minutoli[‡], Vito Giovanni Castellana[‡], Joseph Manzano[‡], David Brooks[*], Gu-Yeon Wei[*], Antonino Tumeo[‡]
[‡]*Pacific Northwest National Laboratory*, Richland, WA, USA
[*]*Harvard University*, Cambridge, MA, USA
[†]*Northeastern University*, Boston, MA, USA
[§]*Politecnico di Milano*, Milan, Italy

***Abstract*—Systems performing scientific computing, data analysis, and machine learning tasks have a growing demand for application-specific accelerators that can provide high computational performance while meeting strict size and power requirements. However, the algorithms and applications that need to be accelerated are evolving at a rate that is incompatible with manual design processes based on hardware description languages. Agile hardware design tools based on compiler techniques can help by quickly producing an application specific integrated circuit (ASIC) accelerator starting from a high-level algorithmic description. We present the SODA Synthesizer, a modular and open-source hardware compiler that provides automated end-to-end synthesis from high-level software frameworks to ASIC implementation, relying on multi-level representations to progressively lower and optimize the input code. Our approach does not require the application developer to write any register-transfer level code, and it is able to reach up to 364 GFLOPS/W efficiency (32-bit precision) on typical convolutional neural network operators.**

***Index Terms*—Compiler Techniques, MLIR, High-Level Synthesis, Hardware generation, Silicon Compiler**

I. Introduction

Many applications, from environmental monitoring, to navigation and control, to scientific experiments, require efficient processing of a combination of data analysis, machine learning (ML), and scientific computing algorithms. They need systems that can effectively support each phase of the computation and adapt in real-time to changes in the environment, under a variety of energy, performance, area, and latency constraints. All these requirements combined make general-purpose processors no longer a viable solution and render application-specific accelerators a necessity.

Typically, domain experts design and validate their algorithms in high-level programming frameworks (most of which are based on Python). Both algorithmic methods and programming frameworks are evolving quickly, especially in the data science and ML areas, making it extremely difficult to design custom accelerators able to support a wide variety of solutions. At the same time, the conventional hardware design cycle has significant productivity limitations. Manually designing custom accelerators in hardware description languages (HDLs) is complex and time consuming, preventing effective exploration of alternative architectures and often requiring a new design cycle each time new algorithms or models appear. General and automated solutions are needed to quickly transition from the formulation of an algorithm to the implementation of a dedicated accelerator.

More in detail, hardware designers usually extract key computational patterns from the algorithms that need to be accelerated, identify parallelism and data reuse opportunities, and design custom functional units for specific kernels at the register-transfer level (RTL) with an HDL. A common alternative to accelerate this process is to implement the functional units in C/C++ and convert them to HDL through High-Level Synthesis (HLS) tools such as Vitis HLS from Xilinx, Catapult C from Siemens, or Stratus HLS from Cadence. In both cases, after functional verification, the HDL kernels are passed to downstream logic synthesis and physical design tools, and finally integrated into a system. This kind of design flow, with part manual coding and part automated processing, is standard practice for designing hardware. However, it still requires tremendous effort, and the quality highly depends on the designers' expertise. Moreover, the interactions between multiple Computer-Aided Design (CAD) tools at different levels of abstractions make the design process tedious and error-prone, introducing significant verification overheads and forcing manual propagation of changes across different stages of the design flow.

To address these issues, we developed the SODA (Software Defined Accelerators) Synthesizer, an open-source, modular, and extensible hardware compiler for the generation of highly specialized accelerators from algorithms designed in high-level programming frameworks. The SODA Synthesizer is composed of a compiler-based frontend, to interface with high-level programming frameworks and apply high-level optimizations, and a compiler-based backend, to generate Verilog code and interface with external tools that compile the final design (either application-specific integrated circuits - ASICs - or field programmable gate arrays - FPGAs).

We used typical linear algebra and deep neural network workloads to test the efficiency of the SODA Synthesizer, exploring its potential to generate optimized hardware designs with high performance. Figure 1, for example, shows the SODA implementations of several different layers from the LeNet convolutional neural network model, in the standard GDSII format for ASIC manufacturing. SODA users can quickly evaluate different design points until they reach the desired solution for their performance or area requirements by

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MM.2022.3178580, IEEE Micro

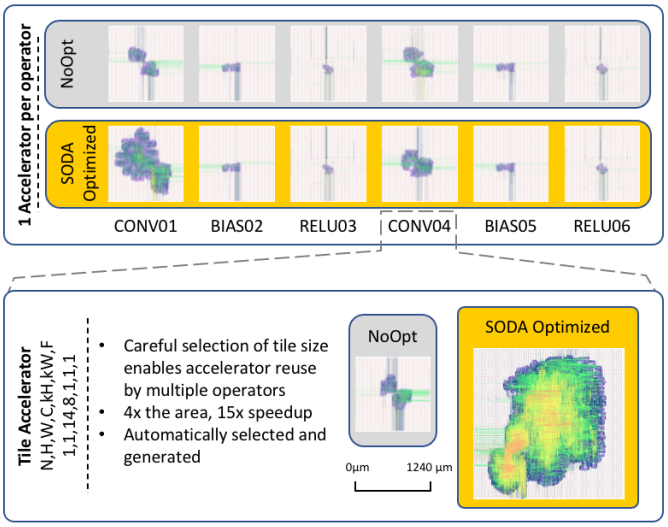


Fig. 1. ASIC implementations of LeNet layers automatically generated by the SODA Synthesizer: with a brief exploration of available compiler options it is possible to reach the desired performance-area trade-off.

selecting different command-line options. Such an exploration would require multiple expensive redesigns with traditional HDL- or HLS-based approaches, potentially never reaching the optimal result due to limited design time available and lack of integration between the different tools in the flow. SODA, instead, provides a no-human-in-the-loop end-to-end hardware compiler where no modifications to the input code are needed, and its multi-level, modular, extensible design offers new opportunities for exploring further analysis and optimization passes.

II. The SODA Framework

Figure 2a provides an overview of the SODA Synthesizer framework, which can be divided in two parts: a compiler-based frontend and a compiler-based hardware generation engine. The framework accepts input descriptions from high-level Python frameworks, translated by the frontend into a high-level intermediate representation (IR). The frontend exploits the Multi-Level Intermediate Representation (MLIR) [8] to perform hardware/software partitioning of the algorithm specifications and architecture-independent optimizations. Subsequently, it generates a low-level IR (LLVM IR) for the hardware generation engine, PandA-Bambu [4], a state-of-the-art open-source HLS tool which, differently from most commercial alternatives, can also accept LLVM IR as input. Optimizations at all levels of the SODA toolchain are implemented as compiler passes, significantly influencing the generated hardware designs in terms of performance, area, and power. An exhaustive exploration of the design space is made possible by enabling and disabling compiler passes or tuning their options.

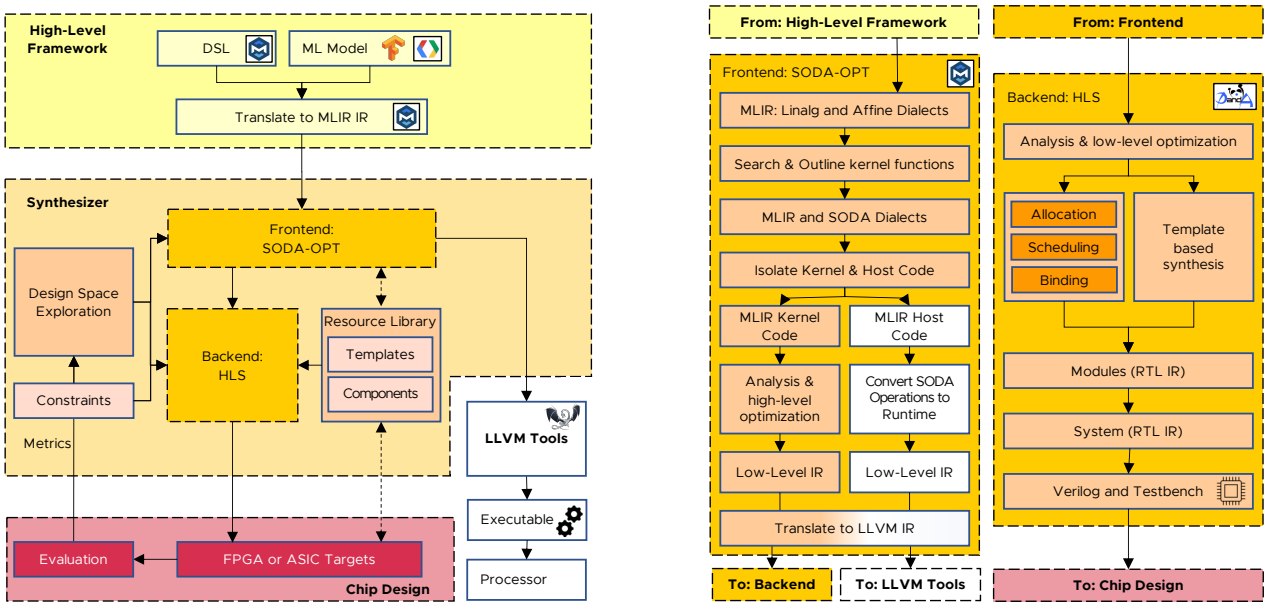
A. SODA-OPT Frontend

SODA-OPT, shown in Figure 2b, is the high-level compiler frontend of the SODA Synthesizer. Its role is to perform *search*, *outlining*, *optimization*, *dispatching*, and *acceleration* passes on the input program, preparing it for hardware synthesis targeting FPGAs or ASICs. To implement these functionalities, SODA-OPT leverages and extends the MLIR framework.

MLIR is a framework that allows building reusable, extensible, and modular compiler infrastructure by defining *dialects*, i.e., self-contained IRs that respect MLIR's meta-IR syntax. Dialects allow modeling code at different levels of abstraction, enabling the use of specialized representations to facilitate specific compiler optimizations. We refer to dialects that are maintained in tree, along with the MLIR framework, as *built-in* dialects. These include abstractions for linear algebra, polyhedral analysis, structured control flow, and others. Several high-level programming frameworks for various domains such as machine learning (TensorFlow, ONNX-MLIR, TORCH-MLIR), scientific computing (NPCOMP), and general-purpose languages (e.g., the FLANG frontend for Fortran) started leveraging MLIR to implement their own specific dialects, optimizations passes, and lowering methods to translate their programs into built-in MLIR dialects. Built-in dialects are entry points to the SODA Synthesizer, enabling high-level frameworks to leverage our toolchain.

SODA-OPT introduces the `soda` dialect to partition input applications into an orchestrating host program and custom hardware accelerators. SODA-OPT analysis and transformation passes ingest MLIR inputs from high-level frameworks, identify key code regions, and outline them into separate MLIR modules. Code regions that are selected for hardware acceleration undergo an optimization pipeline with progressive lowerings through different MLIR dialects (`linalg` → `affine` → `scf` → `cf` → `llvm`), until they are finally translated into an LLVM IR purposely restructured for hardware synthesis. Instead, the host module is lowered into an LLVM IR file that includes runtime calls to control the generated custom accelerators.

Table I summarizes the high-level optimization passes in the SODA-OPT pipeline, and their benefits for the hardware synthesis process. Traditional HLS design flows expect manual code modifications that restructure the original algorithm (to create internal buffers or apply profitable tiling strategies) or tool-specific pragma annotations (to guide unrolling or provide alias information). Instead, SODA-OPT exploits dedicated and context-specific MLIR dialects to apply *systematic* high-level transformations. These can expose instruction- and data-level parallelism, perform loop transformations, and apply various other steps such as buffer hoisting or accumulation on temporary variables. SODA-OPT leverages the `linalg` dialect to identify operations and separate hardware and software partitions, then it optimizes loops through the `affine` dialect, and finally performs CSE, DCE, and SRoA optimizations through the `cf`, `arith`, and `memref` dialects. The optimization pipeline is not monolithic: developers can easily enable, disable, reuse, or tune SODA-OPT passes, providing ample opportunities to enhance them for specific domains and implement automated exploration strategies.



(a) (b) (c)

Fig. 2. The SODA Synthesizer is an open-source, multi-level, modular, extensible, no-human-in-the-loop hardware compiler composed of a high-level compilation framework and a lower-level hardware generator exploiting advanced HLS techniques.

TABLE I
SUMMARY OF HIGH-LEVEL OPTIMIZATIONS IN SODA-OPT. PART OF THEM ARE EXISTING MLIR PASSES, WHILE OTHERS ARE CUSTOM, HLS-ORIENTED IMPLEMENTATIONS.

Optimization	Benefit for HLS	Passes
Single basic block containing the compute intensive part of the kernel	More freedom to schedule operations	Tiling, Unrolling
Increased instruction-level parallelism	Schedule independent arithmetic operations on the same cycle when their inputs are available	Unrolling
Increased data level parallelism	Schedule operations into different memory units on the same cycle	Tiling, Unrolling, Temporary Buffer Allocation
Avoid unnecessary reads from kernel arguments	Reduce expensive accesses to external memory	Temporary Buffer Allocation, Alloca Buffer Promotion
Reuse read results, aggregate on scalars	Save scalar values loaded from memory and intermediate results in registers rather than performing repeated memory accesses	Scalar Replacement of Aggregates (SRoA)
Early alias analysis	Schedule memory operations independently on regions that don't alias	Early Alias Analysis (noalias), Outlining pass
Remove redundant or unnecessary operations	Avoid wasting resources	Common Sub-expression Elimination (CSE), Dead Code Elimination (DCE)

The SODA Synthesizer multi-level approach aims at exploiting different abstractions for different transformations. In the current implementation, there are optimization techniques that can be applied both in the frontend and in the backend: this is often the case for basic compiler passes such as DCE, which are available both in SODA-OPT and in Bambu. Should the two levels interfere with each other in a disruptive way, we would currently intervene and control backend passes on a case-by-case basis.

While the focus of this paper is the generation of hardware accelerators, SODA-OPT can be extended to apply optimizations also on the host code generation path: for example, to enable parallel execution of different accelerators, better use the CPU cache hierarchy, and automatically re-use accelerators when possible.

B. SODA Synthesizer Backend

The SODA Synthesizer backend (Bambu), shown in Figure 2c, leverages state-of-the-art HLS techniques to generate accelerator designs starting from the low-level LLVM IR produced by the SODA frontend. Bambu has several frontends based on standard compilers (GCC or CLANG), it builds an internal IR to perform HLS steps (including bitwidth analysis, loop optimizations, resource allocation, scheduling, and binding algorithms), and finally generates the designs in a hardware description language (Verilog or VHDL). Alongside synthesizable HDL, it can also automatically produce testbenches for verification. Bambu enables the SODA Synthesizer to target FPGAs (from Xilinx, Altera, Lattice, NanoXplore) and ASICs. For ASICs, SODA supports Verilog-to-GDSII generation with both commercial (Synopsis Design Compiler) and open-source (OpenROAD flow) logic synthesis tools.

Bambu is optimized to support a wide set of C and C++

constructs, but it can also ingest LLVM IR through its internal Clang frontend; through SODA-OPT, we connect Bambu with MLIR code. The LLVM IR generated after SODA-OPT performed high-level optimizations is explicitly restructured for HLS, resulting in more efficient accelerators when compared to an input obtained through direct MLIR to LLVM IR translation (as will be shown in the experimental evaluation).

Bambu generates designs at the register transfer level (RTL) following the finite state machine with datapath (FSMD) model; the generated accelerators can subsequently be integrated in larger system-level designs, with or without microcontrollers driving the execution. Bambu also exposes modular synthesis methodologies [10]: differently from other HLS tools, it can generate modules representing functions that may be reused or replicated across an entire design and composed in a complex multi-accelerator system before generating the RTL code.

We have extended Bambu with new HLS methodologies that can integrate FSMD modules as processing elements in coarse-grained dataflow designs [1], and in high-throughput, dynamically scheduled, multithreaded parallel templates [9]. MLIR descriptions are naturally parallel and hierarchical, so it will be possible to instantiate such architectural templates from SODA-OPT. Rather than requiring manual annotations on the input code, we can define the design hierarchy at a higher level of abstraction by exploiting MLIR abstractions, which allow to automatically identify independent operations (`linalg`) and create task-parallel regions (`affine`) in the input code. Each region can subsequently be optimized through the SODA-OPT pipeline described in Section II-A.

C. SODA Resource Library and Verification

The resource library is a crucial component for any hardware synthesis toolchain: it contains RTL descriptions of functional units implementing the operations present in the IR (adders, subtractors, multipliers, etc.), with different versions for different data types. The HLS tool then combines functional units together to build the design. To effectively drive the synthesis algorithms, these functional units also need a *characterization* in terms of performance (e.g., latency of the critical path) and area for each target technology or device. Area and performance estimates, together with related models that describe the area and latency of the interconnections among resources, directly affect many optimization passes and synthesis algorithms: for example, they help decide whether functional units can be chained together by removing intermediate registers, if their combined latency does not exceed the required clock period.

The SODA backend can interface with commercial and open-source logic synthesis tools. We introduced support for the OpenROAD flow and the FreePDK (formerly Nangate) 45 nm cell technology library, providing a completely open-source, end-to-end compiler-based hardware generation flow from high-level programming environments to silicon. We have also added support for the Design Compiler flow, targeting either the FreePDK 45 nm or the Global Foundries 12/14 nm technology node. To achieve this, we have extended the characterization process of the functional units in Bambu: we performed logic synthesis of functional units with FreePDK, collecting all the relevant area and performance metrics to build the resource library and model estimates.

The characterization is also relevant for the implementation of floating point units. While Bambu can integrate hand designed functional units and external intellectual property (IP) libraries (e.g., for FPGAs we select FloPoCo[1]), for the ASIC target in SODA we choose to generate floating point units starting from the standard C soft float library (`math.h`); this allows to easily support different data types (FP32 and FP64), and full IEEE754 compliance if required. The characterization improves the quality of the generated floating point units: for example, the FP32 multiplier has an overall latency of 4 cycles at 200 MHz and 5 cycles at 500 MHz.

Finally, a key component in an end-to-end agile and automated design flow is verification, which assures that the generated designs are functionally correct. Bambu includes a suite of tools that enable automatic testbench generation and validation of results, supporting external open-source and commercial simulators; in the SODA toolchain, we choose to leverage Verilator[2]. We provide Bambu with a set of input values for the synthesized kernel (for example, input arguments of a function) in an XML file. Then, Bambu generates Verilog testbenches and scripts to drive the execution of Verilator. After HLS, Bambu launches the simulation and verifies that the output values from the Verilog kernel correspond to the golden results from the execution of the input code.

III. Experimental Evaluation

In this section, we present results of our end-to-end hardware generation flow. We first demonstrate the effectiveness of the SODA-OPT high-level optimization pipeline on a set of representative linear algebra benchmarks, and then evaluate the entire toolchain by generating custom ASIC accelerators for classic deep neural network models.

The SODA synthesizer enables the generation of custom accelerators for any algorithm that can be described in MLIR. The linear algebra and ML kernels that we considered in this evaluation could also be executed on traditional templated accelerators (i.e. dot-product, matrix-vector, matrix-matrix engines), and our HLS-based approach could instead be used to generate accelerators for less common computational patterns. Nevertheless, we employ these kernels to demonstrate the effectiveness of our high-level optimization flow because they are broadly used in high-level scientific computing frameworks.

In all following experiments, execution times are obtained through simulation using randomly generated test vectors. Area and power results are obtained after OpenROAD place-and-route. Baseline designs (noopt) are synthesized from MLIR code *without high-level optimizations*. All designs

[1]https://gitlab.inria.fr/flopoco/flopoco

[2]https://www.veripool.org/wiki/verilator

(baseline or optimized) are synthesized with Bambu -O2 optimizations.

A. Linear Algebra Kernels

Table II demonstrates the impact of the SODA-OPT optimization pipeline, applied to feed an optimized and restructured low-level IR to the HLS tool for RTL generation. In these experiments, we generate ASIC accelerators for 14 linear algebra kernels from PolyBench[3] translated from C to MLIR `affine`, representing common computations performed within scientific computing and machine learning high-level programming frameworks. We simulate each kernel in isolation, without system-level considerations, to focus on the effects of the optimization pipeline. *Kernel Size* refers to the size of all the dimensions of input and output tensors.

We compare the performance of accelerators generated by simply lowering the benchmarks to LLVM IR (*No High Level Opts.*) against the performance of accelerators generated after performing the SODA-OPT optimizations listed in Table I (*SODA-OPT Pipeline*). In particular, we apply full unrolling on the three innermost nested loops, apply all buffer-related transformations, mark function arguments as not aliasing, apply CSE, DCE and SRoA. Providing an optimized and restructured LLVM IR to Bambu results in more performant designs: accelerators generated from the optimized IRs exhibit an average speedup of 18x, with peaks of 86x, over the baseline. Three kernels exhibit only a small performance improvement (*syr2k* and *syrk* improve between 2x and 3x, while *trmm* does not improve). The reason is that these kernels include inner-loop bounds which depend on the induction variables of the outer loops, and the Scalar Replacement of Aggregates (SRoA) pass could not perform scalar replacement. This can be solved in the future by adding an additional optimization pass to simplify index calculations when the loop bounds are known.

B. Neural Network Models

We used the SODA Synthesizer to automatically generate accelerators for relevant operators of the LeNet, MobilneNetV2, ResNet-18, and ResNet-50 convolutional neural network models. These models were trained with TensorFlow in 32-bit floating point precision, converted into protobuf files, and translated into built-in MLIR abstractions (`tosa` and `linalg`). No modifications to the original high-level models were required. By default, SODA-OPT selects and partitions the input model to create one accelerator for each DNN layer. For the sake of conciseness, and because the same computation patterns are repeated multiple times in the network, we selected a subset of layers for our experiments. We outlined them into isolated kernels, applied selected high-level optimizations or the complete SODA-OPT pipeline, and generated Verilog targeting ASIC technologies. We report execution time, area, power, and efficiency (expressed as FLOPS per Watt) for each experiment. Although the total end-to-end synthesis time from high-level description to GDSII varies depending on the specific kernel, all designs required less than 3 hours of processing on a node with two AMD EPYC 7282 16-Core CPUs and 256 GB of DDR4 3.2 GHz memory.

a) LeNet: - In the top part of Table III, we present runtime, area and power metrics of LeNet accelerators that cover 98% of its execution time (45 nm technology). Each line in the table corresponds to a single accelerator. We previously showed the final floorplans of these accelerators in the top part of Figure 1. We applied a subset of the available MLIR optimizations at the linear algebra and affine abstractions, observing speedups up to 6.2x and an efficiency between 2.68 and 41.75 GFLOPS/W.

b) MobileNetV2: - Table III also shows results for relevant MobileNetV2 Depth-Wise convolution (DWC2D) layers, representing 35% of MobileNetV2 inference time. The simplest optimization (leveraging high-level abstractions to propagate alias information automatically) already results in speedups of around 2x and designs reaching an efficiency over 1 GFLOPS/W. All the selected MobileNet layers have the same structure (varying only tensor dimensions and loop bounds), and thus benefit in the same way from the applied optimization, i.e., allowing Bambu to schedule memory operations on different arguments in parallel because the input arguments do not alias.

c) Reusable Accelerators: - Optimizing entire convolution operations in LeNet and MobileNetV2 does not allow performance increases higher than 2.1x. Instead, applying an appropriate tiling strategy to balance the size of the design considering both operations and memory parallelism allows to significantly improve performance. We tile a convolution operation and outline the tile, so that the generated accelerator is invoked multiple times to run a convolutional layer. We also ensure that the same tile can be reused across different layers in deeper networks (35, 14, and 46 convolutional layers in MobileNetV2, ResNet-18, and ResNet-50, respectively). Table IV shows results for the generated accelerators with and without applying the SODA-OPT optimization pipeline, which provides up to 15.2x speedup with respect to the unoptimized baseline and efficiency between 103 and 364 GFLOPS/W in the 12/14 nm technology. If we compare the results of the *tile approach* with what can be achieved by outlining a *full convolution*, we obtain, for example, that executing the fastest version of the LeNet CONV_04 layer is 14.89x slower than executing 44800 times the optimized LeNet tile in Table IV (assuming 2 cycles latency for load and 1 cycle for store operations from a private scratchpad memory with two ports).

Overall, our experimental evaluation demonstrates the effectiveness of an end-to-end modular silicon compiler. The SODA Synthesizer allows generating, optimizing, and exploring hardware designs without requiring to write any RTL code. Optimizations implemented at different abstraction levels across the modular compiler-based toolchain allow iterative improvements of the generated accelerators, with high quality results in terms of performance and efficiency.

[3]https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/ and https://github.com/wsmoses/Polygeist-Script

TABLE II
EXECUTION TIME (IN CLOCK CYCLES) FOR POLYBENCH KERNELS WITH ASIC TARGET - FREEPDK 45NM @ 500MHZ. SPEEDUP SHOWN IN PARENTHESIS.

Opt. Strategy	No MLIR Opts.				SODA-OPT Pipeline			
Kernel Size	**2**	**4**	**8**	**16**	**2**	**4**	**8**	**16**
symm	421	2,928	21,400	163,368	31 (13.6x)	34 (86.1x)	325 (65.8x)	2,600 (62.8x)
three_mm	388	3,087	25,010	211,298	47 (8.3x)	82 (37.6x)	656 (38.1x)	5,248 (40.3x)
two_mm	315	2,475	20,258	167,490	52 (6.1x)	86 (28.8x)	688 (29.4x)	5,504 (30.4x)
gemm	186	1,446	11,922	95,376	31 (6.0x)	56 (25.8x)	448 (26.6x)	3,584 (26.6x)
doitgen	277	4,282	67,666	999,698	29 (9.6x)	258 (16.6x)	2,064 (32.8x)	16,512 (60.5x)
bicg	129	518	2,058	8,482	26 (5.0x)	43 (12.0x)	85 (24.2x)	340 (24.9x)
mvt	130	514	2,051	8,195	26 (5.0x)	45 (11.4x)	89 (23.0x)	356 (23.0x)
gemver	283	1,118	4,393	17,617	77 (3.7x)	106 (10.5x)	424 (10.4x)	1,696 (10.4x)
gesummv	162	578	2,178	8,722	39 (4.2x)	56 (10.3x)	105 (20.7x)	420 (20.8x)
atax	132	523	2,067	8,227	44 (3.0x)	73 (7.2x)	292 (7.1x)	1,168 (7.0x)
syr2k	186	1,310	9,018	68,986	38 (4.9x)	567 (2.3x)	3,033 (3.0x)	24,264 (2.8x)
syrk	142	990	6,714	49,250	31 (4.6x)	453 (2.2x)	2,581 (2.6x)	20,648 (2.4x)
trmm	46	532	4,402	34,018	24 (1.9x)	532 (1.0x)	4,402 (1.0x)	34,018 (1.0x)

TABLE III
LENET AND MOBILENETV2 RESULTS - FREEPDK 45NM @500MHZ. GRAY ROWS SHOW BASELINE KERNELS.

Model	Kernel	MLIR Opts.	Cycles	Area(um^2)	Power(W)	Runtime (s)	GFLOPS	GFLOPS/W	Speedup
LeNet	CONV_01	noopt	10,262,618	29,073	0.01380	20.53E-03	0.061	4.43	Baseline
LeNet	CONV_01	noalias	4,627,982	124,255	0.05060	9.26E-03	0.135	2.68	2.22
LeNet	BIAS_02	noopt	251,694	10,395	0.00434	503.39E-06	0.049	11.48	Baseline
LeNet	BIAS_02	noalias+unroll	40,826	60,048	0.03410	81.65E-06	0.307	9.01	6.17
LeNet	RELU_03	noopt	151,342	7,385	0.00399	302.68E-06	0.165	41.55	Baseline
LeNet	RELU_03	noalias+unroll	38,446	35,695	0.01700	76.89E-06	0.652	38.39	3.94
LeNet	CONV_04	noopt	85,380,948	36,814	0.01770	170.76E-03	0.058	3.32	Baseline
LeNet	CONV_04	noalias	83,380,180	37,556	0.01800	166.76E-03	0.060	3.34	1.02
LeNet	BIAS_05	noopt	62,932	10,409	0.00453	125.86E-06	0.049	11.00	Baseline
LeNet	BIAS_05	noalias+unroll	10,222	60,007	0.03650	20.44E-06	0.306	8.41	6.16
LeNet	RELU_06	noopt	37,844	7,464	0.00397	75.69E-06	0.165	41.75	Baseline
LeNet	RELU_06	noalias+unroll	9,620	35,950	0.01760	19.24E-06	0.651	37.04	3.93
MobileNetV2	DWC2D_01	noopt	87,319,010	39,106	0.01800	174.64E-03	0.062	3.45	Baseline
MobileNetV2	DWC2D_01	noalias	43,966,946	62,676	0.02570	87.93E-03	0.123	4.80	1.99
MobileNetV2	DWC2D_02	noopt	65,482,874	38,108	0.01790	130.97E-03	0.015	0.87	Baseline
MobileNetV2	DWC2D_02	noalias	32,968,826	61,767	0.02500	65.94E-03	0.030	1.23	1.99
MobileNetV2	DWC2D_05	noopt	32,740,654	38,142	0.01720	65.48E-03	0.062	3.61	Baseline
MobileNetV2	DWC2D_05	noalias	16,483,630	61,684	0.02510	32.97E-03	0.123	4.91	1.99

TABLE IV
CONV2D KERNEL RESULTS @500MHZ. WHITE ROWS SHOW BASELINE KERNELS.

Operation and Kernel Information			Runtime Information				Synthesis Results for FreePDK 45nm			Synthesis Results for GF 12nm		
Target Model	**Dimensions**	**MLIR Opt.**	**Cycles**	**Runtime (s)**	**GFLOPS**	**Avg. Speedup**	**Area(um^2)**	**Power(W)**	**GFLOPS/W**	**Area(um^2)**	**Power(W)**	**GFLOPS/W**
LeNet	1,1,14,8,1,1,1	No Opt.	1,809	3.62E-06	0.061	Baseline	38,375	0.01760	3.52	8,470	0.00017	364.19
LeNet	1,1,14,8,1,1,1	SODA-OPT	125	250.00E-09	0.896	15.25	673,558	0.27400	3.27	169,635	0.00689	130.04
MobileNetV2	1,7,7,4,1,1,1	No Opt.	3,194	6.39E-06	0.061	Baseline	26,811	0.01050	5.84	6,257	0.00059	103.31
MobileNetV2	1,7,7,4,1,1,1	SODA-OPT	225	450.00E-09	0.871	14.20	752,356	0.38200	2.28	190,354	0.00523	166.56
ResNet18,50	1,1,1,64,1,1,1	No Opt.	963	1.93E-06	0.066	Baseline	15,994	0.00533	12.47	3,794	0.00038	176.75
ResNet18,50	1,1,1,64,1,1,1	SODA-OPT	99	198.00E-09	0.646	9.73	413,867	0.20200	3.20	105,053	0.00455	142.08

IV. RELATED WORK

Several works have explored generation of custom hardware accelerators starting from high-level programming frameworks, focusing in particular on Python and machine learning. They typically resort to one of two approaches: either (1) compile and map functions to parametrized modules or architectures, or (2) convert code to imperative languages (C/C++) for HLS, often heavily annotated to work with specific commercial tools.

Approach (1) consists of solutions like VeriGOOD-ML [3], which maps ML models described in the ONNX format to three substantially different architecture templates for different types of neural networks through the PolyMath compiler. GEMMINI [5] provides a parametrized systolic array generator in Chisel that connects to a RISC-V core; the GEMMINI toolchain then offloads operations from specific layers of ONNX models to the systolic array. TVM's VTA architecture [11] is a specialized co-processor for matrix multiplication, generated through HLS for FPGA; the TVM high-level framework can compile machine learning models into a stream of instructions for VTA. Additional ongoing work on TVM proposes to compile specific deep neural network operators into ASIC leveraging parametrized RTL templates. All these solutions aim at automatically generating ASIC designs, but they remain limited as they only support layers and kernels that have a direct mapping to one of the provided hardware templates. The SODA Synthesizer, instead, leverages high-level and lower-level (HLS) compiler-based tools. Hence, it provides a more general framework able to generate ASIC designs for virtually any computational pattern, as long as a

lowering to MLIR is available. Such automatically generated accelerators lead to less flexible designs with respect to dedicated parametrized templates, but they can provide higher performance and efficiency. To the best of our knowledge, our design flow is the first one to provide a completely automated path from generic high-level code to fully custom ASIC accelerators.

Solutions that implement approach (2) include PyLog [6], which defines a high-level compilation infrastructure for Python programs and generates annotated C/C++ code that is then fed to Xilinx Vivado HLS for generation of the accelerators. HeteroCL [7] partitions code between general-purpose processor and FPGA, providing a library of functions to insert hardware-specific information in the source code, which is then used to generate annotated C/C++ for HLS tools. ScaleHLS [12] is a tool that facilitates HLS through high-level optimizations implemented in MLIR, potentially allowing to synthesize accelerators starting from high-level programming environments that lower to an MLIR representation; however, it also resorts to writing back annotated C code for Vivado HLS. While all these tools bridge high-level programming frameworks with hardware generators, they have limited flexibility, as they define compilation pipelines that only support specific high-level frameworks and backend HLS tools. Moreover, after applying hardware-related optimizations, they all generate code at a different (higher) level of abstraction, potentially losing a considerable amount of semantic information in the process.

Finally, the CIRCT (Circuit IRs Compiler and Tools) incubator project [2] uses MLIR to build a set of interoperable tools for hardware design. The project focuses on creating relevant circuit-level IR abstractions for RTL generation. Once matured, CIRCT dialects could be merged into the MLIR framework, potentially becoming a building block for hardware compilers.

V. Conclusions

This paper presents the SODA Synthesizer, a modular, multilevel, end-to-end compiler-based design automation tool that enables the generation of custom accelerator designs starting from high-level software programming frameworks. The framework is composed of inter-operating open-source technologies: SODA-OPT (https://gitlab.pnnl.gov/sodalite/soda-opt), an extensible high-level frontend and optimizer based on the MLIR framework, and PandA-Bambu HLS (https://panda.dei.polimi.it), a lower-level hardware generator. The toolchain can interface with the OpenROAD Flow (https://theopenroadproject.org) to provide a fully open-source path to ASIC generation.

We have shown the effectiveness of compiler-based optimizations on linear algebra kernels and deep neural network models, discussed the impact of the optimizations on the final ASIC designs, and demonstrated how our toolchain allows generating efficient hardware designs without requiring developers to write any RTL code. The SODA toolchain dramatically shortens the hardware design cycle from algorithmic formulation to hardware implementation, considers system-level implications, and enables rapid design space exploration and agile hardware development.

References

[1] V. G. Castellana, A. Tumeo, and F. Ferrandi. High-level synthesis of parallel specifications coupling static and dynamic controllers. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 192–202, 2021.

[2] CIRCT Developers. "CIRCT" / Circuit IR Compilers and Tools, 2020.

[3] H. Esmaeilzadeh, S. Ghodrati, J. Gu, S. Guo, A. B. Kahng, J. K. Kim, S. Kinzer, R. Mahapatra, S. D. Manasi, E. S. Mascarenhas, S. Sapatnekar, R. Varadarajan, Z. Wang, H. Xu, B. R. Yatham, and Z. Zeng. VeriGOOD-ML: An open-source flow for automated ml hardware synthesis. In *2021 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–7, 2021.

[4] F. Ferrandi, V. G. Castellana, S. Curzel, P. Fezzardi, M. Fiorito, M. Lattuada, M. Minutoli, C. Pilato, and A. Tumeo. Bambu: an open-source research framework for the high-level synthesis of complex applications. In *DAC 2021: 58th ACM/IEEE Design Automation Conference*, pages 1327–1330, 2021.

[5] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, et al. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *Proceedings of the 58th Annual Design Automation Conference (DAC)*, 2021.

[6] S. Huang, K. Wu, H. Jeong, C. Wang, D. Chen, and W. Hwu. Pylog: An algorithm-centric python-based fpga programming and synthesis flow. *IEEE Transactions on Computers*, 70(12):2015–2028, 2021.

[7] Y. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang. Heterocl: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 242–251, 2019.

[8] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. MLIR: A Compiler Infrastructure for the End of Moore's Law. *ArXiv*, 0(0):1–21, 2020.

[9] M. Minutoli, V. Castellana, N. Saporetti, S. Devecchi, M. Lattuada, P. Fezzardi, A. Tumeo, and F. Ferrandi. Svelto: High-level synthesis of multi-threaded accelerators for graph analytics. *IEEE Transactions on Computers*, (01):1–14, feb 2021.

[10] M. Minutoli, V. G. Castellana, A. Tumeo, and F. Ferrandi. Function proxies for improved resource sharing in high level synthesis. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 100–100, 2015.

[11] T. Moreau, T. Chen, L. Vega, J. Roesch, E. Yan, L. Zheng, J. Fromm, Z. Jiang, L. Ceze, C. Guestrin, et al. A hardware–software blueprint for flexible deep learning specialization. *IEEE Micro*, 39(5):8–16, 2019.

[12] H. Ye, C. Hao, J. Cheng, H. Jeong, J. Huang, S. Neuendorffer, and D. Chen. Scalehls: Scalable high-level synthesis through mlir. *arXiv preprint arXiv:2107.11673*, pages 1–13, 2021.