Towards Automatic and Agile AI/ML Accelerator Design with End-to-End Synthesis

(Invited Paper)

Jeff (Jun) Zhang*, Nicolas Bohm Agostini†, Shihao Song†, Cheng Tan†, Ankur Limaye†, Vinay Amatya†, Joseph Manzano†, Marco Minutoli†, Vito Giovanni Castellana†, Antonino Tumeo†, Gu-Yeon Wei*, David Brooks*

*Harvard University, †Pacific Northwest National Laboratory

*jeffzhang@g.harvard.edu, †antonino.tumeo@pnnl.gov

***Abstract*—Domain-specific designs offer greater energy efficiency and performance gain than general-purpose processors. For this reason, modern system-on-chips have a significant portion of their silicon area with custom accelerators. However, designing hardware by hand is laborious and time-consuming, given the large design space and the performance, power, and area constraints that are not realized in the software. Moreover, domain-specific algorithms (e.g., machine learning models) are evolving quickly, challenging the accelerator design further. To address these issues, this paper presents SODA Synthesizer, an automated open-source high-level ML framework to Verilog modular compiler targeting AI/ML Application-Specific Integrated Circuits (ASICs) accelerators. SODA tightly couples the Multi-Level Intermediate Representation (MLIR) compiler infrastructure [24] and open-source HLS approaches. Thus, SODA can support various ML frameworks and algorithms and can perform optimizations that combine specialized architecture templates and conventional HLS to generate the hardware modules. In addition, SODA's closed-loop design space exploration (DSE) engine allows developers to perform end-to-end design space explorations on different metrics and technology nodes.**

I. Introduction

Machine Learning (ML) has revolutionized vision, speech, language understanding, web search, medical diagnosis, and many other fields [11]. The rise of ML has been fueled by the improvements in high performance processors (e.g., AlexNet was trained on GPUs [23].); Conversely, the ever-growing computation and memory requirement of the state-of-the-art ML models also push the hardware to its limit. For example, OpenAI's GPT-3 language model has 175 billion parameters and is estimated to require 355 years to train on a single Tesla V100 [2].

From the semiconductor technology perspective, the end of Moore's Law and Dennard scaling means architects have to find new ways to exploit parallelism for hardware efficiency, i.e., domain specific designs [12]. As a result, many specialized deep learning accelerators have been designed for ML workloads over the years [13]. Modern system-on-chips (SoCs) are provisioned with a sea of custom accelerators that offer orders of magnitude power efficiency and performance gains [37].

The complexity and heterogeneity in large SoCs challenge the hardware design cycle. Fig. 1 shows a typical ML accelerator design cycle. The process starts with a high-level description (using libraries such as TensorFlow [4] and PyTorch [33]) of the ML models. Then, hardware designers analyze the ML algorithms to identify the parallelism and data reuse opportunities, determine the microarchitecture and dataflow, and manually implement the algorithms in RTL or in C/C++ with high-level synthesis tools (e.g., Xilinx's Vitis HLS). The functional verified RTL implementations are then passed through the ASIC flow for logic synthesis, physical design, and system integration. Despite being the standard practice for designing hardware, *this decoupled design flow has multiple issues.* First, writing RTL by hand requires tremendous effort, and the quality depends highly on the designer's expertise. Second, even with HLS tools, designers still have to follow specific coding styles (that require manual code transformations) and explicitly annotate the *pragmas* for the tool to understand the optimization opportunities correctly and produce good designs. Third, the whole process requires iterative interactions with multiple CAD tools at different levels of abstractions which makes the design *tedious* and *error-prone*, incurring significant verification overheads. Finally, any changes in later design stages have to be *manually* back-propagated to early stages, preventing optimization opportunities across the design flow.

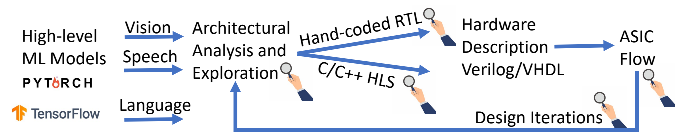

Fig. 1. A typical AI/ML accelerator design cycle that involves multiple manual efforts.

With more than 100 new ML papers being published per day [11], rapid evolution in ML models and their diverse operators further amplify the aforementioned issues. Thus, to keep pace with the AI/ML community and generate high quality accelerators for fast evaluation and deployment, there is a pressing need for an end-to-end ***automatic*** synthesis flow that enables ***agile*** hardware design.

In this paper, we describe the **Software Defined Architectures** (SODA) Synthesizer, a novel no-human-in-the-loop hardware generator that automates the creation of ML

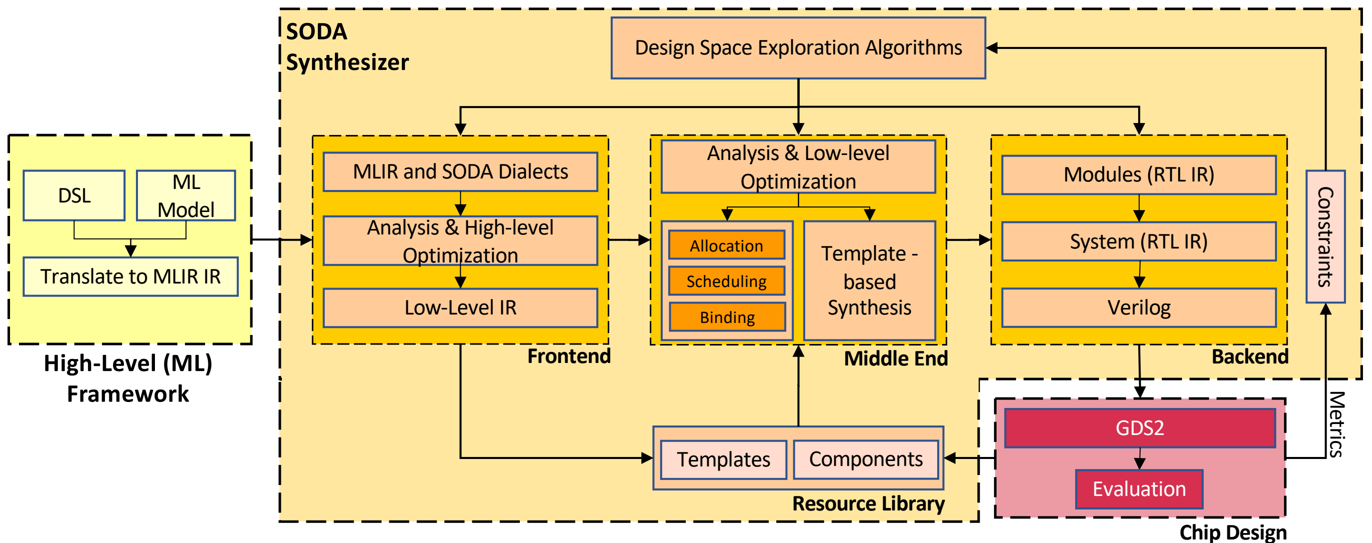

Fig. 2. Overview of the SODA Synthesizer.

accelerators from high-level ML frameworks. Our SODA Synthesizer is built with modular compiler-based approaches. At the frontend, SODA leverages the Multi-Level Intermediate Representation (MLIR) compiler infrastructure [24] to support a wide variety of ML frameworks and algorithms. It progressively lowers the intermediate representation (IR) of the deisgn into HLS compatible LLVM IR. At the middle end, SODA leverages open-source HLS tools (able to ingest LLVM intermediate representation) to perform optimizations. SODA Synthesizer combines specialized architecture templates and conventional HLS approaches to generate the hardware modules. SODA backend then generates the final RTL descriptions for different targets and technology nodes. In addition, SODA also adopts a design space exploration (DSE) engine that allows exploring trade-offs on different metrics and identifies the most promising design point.

In the remainder of this paper, we provide a general overview of SODA Synthesizer, describe its main components, present end-to-end synthesis results on full ML models and individual kernels, and discuss future research opportunities.

II. SODA Framework

Figure 2 provides an overview of the SODA Synthesizer framework. Overall, the SODA synthesizer is a modular, multi-level, interoperable, extensible, open-source compiler-based framework. The framework takes input descriptions from high-level machine learning frameworks, translated by the compiler-based frontend, to a high-level intermediate representation (IR) in MLIR [24]. The SODA frontend then exploits MLIR to perform hardware-software partitioning of the machine learning algorithm specifications and architecture independent optimizations. Subsequently, it generates the low-level IR (LLVM IR) fed to the SODA middle end, the actual hardware synthesis engine of the toolchain. SODA Synthesizer leverages conventional HLS techniques as well as optimized architectural templates that match computing patterns in typical ML models. We currently support two different synthesizer middle-ends. One is a prototype based on the actual LLVM compiler, the other is PandA-Bambu [3], a state-of-the-art high-level synthesis tool which we have extended with several techniques to support dataflow [8] and specialized high-throughput parallel templates [29]. Both middle-ends consume pre-optimized LLVM IR and generate synthesizable Verilog for Application Specific Integrated Circuits (ASICs) or Field Programmable Gate Arrays (FPGAs). For FPGA synthesis, the integrated toolchain supports devices from Xilinx, Altera, Lattice, NanoXplore, and more. For ASICs, the toolchain supports Verilog-to-GDS2 generation with both commercial (Synopsis Design Compiler) and opensource (OpenROAD) toolchains.

SODA is entirely compiler-based, so optimizations at all levels are implemented as compiler passes. This means that different optimization parameters will influence the final hardware designs generated by the SODA Synthesizer. Thus, there is a need to explore the accelerator designs space considering metrics such as performance, area, energy and aspects such as overall system cost, size, and cooling requirements that are typical for cyber-physical and autonomous systems. The SODA Synthesizer implements a pluggable set of multi-objective optimization algorithms, which can be applied both at the frontend and the middle end. Because compiler-based tools can expose a multitude of parameters, an exhaustive exploration of the design space may not be feasible within a limited time. Hence, approximate and heuristic methods are preferred. In previous works [16], [34], we have explored various bio-inspired heuristics that could be easily ported and integrated within the SODA Synthesizer. Performing efficient design space exploration obviously depends on obtaining accu-

rate results from the evaluation of the generated designs, thus requiring effective interfacing with the logic design tools to provide an accurate characterization of the final GDS2 layouts.

A. High-Level ML Framework and SODA-opt Frontend

End-to-end compiler infrastructures are being developed to map computations implemented with high-level languages to different architecture targets (CPUs, GPUs, TPUs, etc). These targets have different needs and constraints at the low-level mapping, such as custom instruction set architecture (ISAs), memory layouts, hardware interfaces, and application programming interfaces (APIs). However, as high-level computations are progressively translated into low-level instructions, different targets *can share high-level transformations and optimizations*, such as HLS approaches do.

To implement this approach, SODA leverages the MLIR compiler infrastructure framework on its frontend. The MLIR framework facilitates the creation of domain-specific compilers by providing code generators, translators, optimizers, and the infrastructure to define subsets of operations that expose well-defined language abstractions [24]. MLIR is designed to allow progressive lowering between existing and newly designed operations, promoting the reuse of abstraction levels and compiler passes that are the design philosophy of SODA framework.

With MLIR, complicated analysis passes that reconstruct a program's high-level information (e.g., task-level parallelism, loop analysis, memory access patterns) can be avoided, as this information is readily available at this abstraction. This approach contrasts with reconstructing the information from a control flow graph (i.e., LLVM IR CFG) or manually encoding the information with annotations (i.e., OpenMP, HLS pragmas). Moreover, this information can be easily carried over to drive lower-level transformations, which alleviates the need to translate between multiple frameworks and toolchains, making the end-to-end compilation more effectively.

In MLIR, a group of operations modeling an abstraction is called a *dialect*. Dialects are self-contained IRs and follow the language rules of MLIR's meta-IR, enabling the framework to have multiple dialects co-existing in the same MLIR environment. To support the underlying kernels from many different high level ML frameworks, MLIR contains a linear algebra dialect called `linalg`. This dialect implements a set of common ML operations such as *convolutions*, *matrix multiplications*, *dot products*, etc. Operations of higher-level dialects can be lowered to `linalg` operations and leverage all the subsequent transformations supported by `linalg` or other lower-level abstractions.

Figure 3 shows when a Deep Neural Network (DNN) model gets translated into MLIR, undergoes a sequence of progressive lowering steps, and generates the LLVM IR. During these steps, we observe the available MLIR concepts, and highlight specific dialects that our proposed `soda` dialect will interact with (i.e. for `soda` dialect to outline).

We present *SODA-opt* frontend, a high-level compiler tool that fully leverages and extends the MLIR framework. *SODA-opt* enables system-level design with mechanisms to search, outline, optimize, and accelerate application code targeting custom accelerator synthesis. To achieve these, *SODA-opt* implements a set of compiler libraries and passes that perform the code outlining and optimization, and a set of runtime libraries to offload the computations from the host to selected accelerator targets. Specifically, in *SODA-opt*, we introduce the `soda` dialect. Akin to the `gpu` dialect [14], `soda` defines operations to *outline* MLIR code region(s) that will become our kernel(s) for HLS, and operations to separate the outlined kernels. The process of finding code regions, outlining these regions into a module and individual kernels, optimizing each kernel for different HLS backends, and setting up the host calls to drive the generated accelerators is fully automated as the transformation and optimization passes introduced by *SODA-opt*.

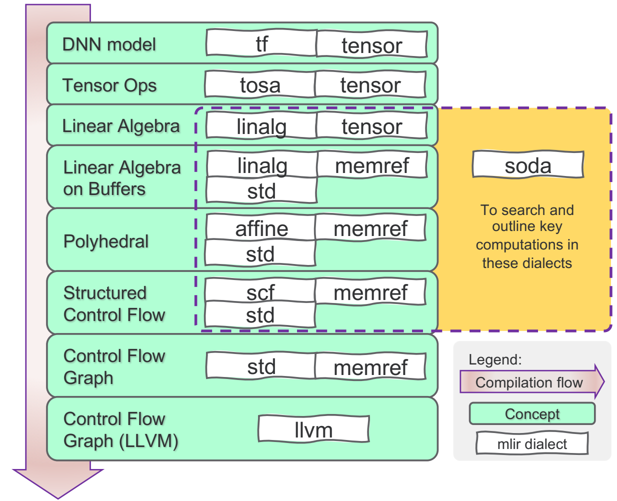

Fig. 3. Progressive lowering, MLIR and SODA dialects.

During the kernel optimization, *SODA-opt* implements target-aware transformation pipelines based on existing optimizations in MLIR and custom optimizations specific for accelerator HLS targets. These optimizations focus on exposing instruction- and data-level parallelism, automatic loop transformations (e.g., loop unrolling, loop reordering), and other steps such as buffer hoisting, accumulation on temporary variables, etc. Paired with *SODA-opt* flexible outlining pass, it enables (1) the possibility that different code regions/kernels can be fused to generate a single accelerator; (2) multiple accelerator design points for the same model inputs. *SODA-opt*'s compiler based optimization passes eliminate HLS designers to manually perform code transformations and explicitly tweak *pragmas* annotations. After outlining the kernels, *SODA-opt* applies optimizations to lower the code through the MLIR concepts/abstractions as shown in Figure 3. The final output of *SODA-opt* is an LLVM IR file to be used by SODA Synthesizer middle end.

B. SODA Synthesizer Middle End

As highlighted in the overview, the SODA toolchain is currently able to leverage two different middle ends. The

first one is PandA-Bambu [3], an open-source HLS tool supporting several advanced features. Bambu normally interfaces with frontend compilers such as GCC or CLANG, ingesting their single source assignment (SSA) form IR (GIMPLE or LLVM IR), building its own IR to perform the HLS steps, and generating the hardware designs in hardware description languages (HDL) such as Verilog and VHDL. While Bambu, in combination with more conventional compiler frontends, is optimized to support a wide array of C and C++ constructs, we extend it to support the MLIR-based *SODA-opt* frontend in the context of the SODA framework. After *SODA-opt*'s architecture independent and the hardware synthesis specific high-level optimizations, Bambu takes in the optimized LLVM IR and applies its low-level *synthesis-specific* optimizations (including datapath-bitwidth simplifications, loop optimizations, advanced resource allocation, scheduling, and binding algorithms) to generate efficient hardware designs. By leveraging the natural parallel and hierarchical description provided by MLIR, it is also possible to simplify Bambu's parallel accelerator generation methodologies. Rather than requiring high-level code annotations to instantiate parallel hardware templates, MLIR facilitates identifying the parallel set of operations and abstract definition of the design hierarchy. Bambu also includes a suite of tools that enable automatic testbench generation and automatic results validation, which certify the functional correctness of synthesized modules.

The second middle end of the SODA toolchain is a new synthesis tool entirely based on the LLVM compiler. Other HLS tools leverage LLVM only to convert to its own IR (e.g., Bambu) or to exploit the IR of the typical general purpose languages compiler (e.g., LegUp). In contrast, the SODA Synthesizer middle end implements the hardware synthesis algorithms and the generation of the hardware description as an LLVM target. While this requires rewriting several of the low-level compiler passes to remove constraints on functional resources present when targeting a specific processor design with an instruction set, it allows reusing all the algorithms implemented in the compiler itself. Instead of directly emitting HDL, the LLMV-based SODA Synthesizer emits another register-transfer level IR. In its current version, our synthesizer emits FIRRTL (Flexible Intermediate Representation for Register Transfer-Level) because tools are readily available to automatically convert FIRRTL into Verilog designs and testbench files (e.g., the Scala compiler). However, a potential future target could be the circuit-level IRs currently explored in the CIRCT (Circuit IRs and Compiler Tools), an LLVM incubator project leveraging the MLIR framework. Emitting a circuit-level IR enables supporting different device technology by implementing the retargeting as compiler transformations (e.g., supporting block RAMs in FPGAs in place of multi-ported scratchpad memories in ASICs), and provides natural modularity and composability, as the actual instantiations of the interfaces can also be implemented as a compiler pass.

Figure 5 shows how the LLVM-based middle end converts the LLVM IR of Figure 4 to FIRRTL.

```
; Function Attrs: norecurse nounwind readnone ssp uwtable
define i32 @f(i32, i32) local_unnamed_addr #0 {
  %3 = icmp sgt i32 %0, %1
  %4 = select i1 %3, i32 %0, i32 %1
  ret i32 %4
}
```

Fig. 4. LLVM IR of a simple if-then-else branch.

```
; Generated with the SODA compiler
; f                    ;  -- Begin function f
circuit f:
  module f:            ;  @f
    input clk : Clock
    input rst : UInt<1>
    input start : UInt<1>
    output done : UInt<1>
    output return_value : UInt<32>
    input arg_1 : UInt<32>
    input arg_2 : UInt<32>

    reg  state : UInt<64>, clk with :
                 (reset => (rst, UInt<64>(2)))
    reg  r1 : UInt<32>, clk with :
                 (reset => (rst, UInt<32>(0)))
    reg  r2 : UInt<32>, clk with :
                 (reset => (rst, UInt<32>(0)))
    reg  rb1 : UInt<1>, clk with :
                 (reset => (rst, UInt<1>(0)))

    done <= UInt<1>(0)
    state <= UInt<64>(2)
    return_value <= UInt<32>(0)

    when eq(start, UInt<1>(1)):
      state <= UInt<64>(0)
      r1 <= arg_1
      r2 <= arg_2

    when eq(state, UInt<64>(0)):
      rb1 <= gt(r1, r2)
      r1 <= mux(rb1, r1, r2)
      return_value <= r1
      done <= UInt<1>(1)
      state <= UInt<64>(2)  ;  -- End function
```

Fig. 5. Example of FIRRTL generated by the LLVM-based synthesizer of the SODA toolchain.

C. SODA Resource Library, Backend, and Verification

The resource library is a key component for any hardware synthesis toolchain. For example, a conventional HLS tool requires as a minimum the RTL description of the functional units that implements the operations identified in the IR (adders, subtractors, multipliers, etc) for the various datatypes, which are then combined into the design. To effectively drive the synthesis algorithms, these functional units also need to be characterized, at least in terms of performance (e.g., latency of the critical path) and area on the target technology of choice.

For example, aggressive resource sharing can result in additional steering logic and deeper multiplexers chains, which can lead to violation of latency constraints and, consequently, to the need of lowering clock frequency. In other cases, we can further optimize the circuit design, applying techniques like chaining of functional units (which removes registers between them), when the overall latency of a sequence of cascading functional units (i.e., directly operating on the output of the previous ones) stays within the clock cycle boundary. Hence, area and performance estimates, and related models that describe area and latency of the interconnection across

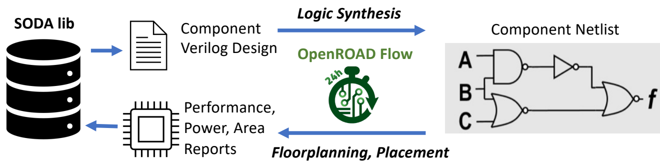

Fig. 6. The SODA Synthesizer characterization flow.

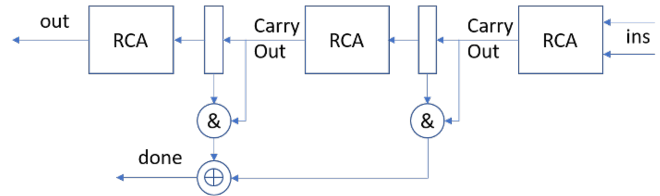

Fig. 7. Multi-speculative Adder. We can see the flip-flops between each couple of Ripple-Carry Adders (RCAs) are used for the predictions.

resources, directly affect many of the optimization passes and synthesis algorithms, such as scheduling and binding.

One of the distinctive features of the SODA backend is the ability to interface with both commercial and open-source ASIC tools. In particular, we introduced the support for the OpenROAD logic synthesis project and the OpenPDK (formerly Nangate) 45nm cell technology library. We are also exploring the use of Design Compiler, with both OpenPDK and the Global Foundries 12nm technology node. This includes, for Bambu, both the scripts and the characterization process of the functional units, exploiting Bambu's flexible and extendible resource library. For the LLVM-based SODA Synthesizer middle end, we have been designing an approach that, starting from a library of resources in Verilog (SODA Lib), enables us to generate a custom TableGen file containing all the relevant metrics by iteratively synthesizing the resources in OpenROAD. The TableGen file is then used at compilation of the actual synthesis engine. The LLVM-based synthesizer exploits modularity and support for importing external components in a FIRRTL description to then integrate the specialized functional units (rather than using high-level behavioral FIRRTL descriptions).

Figure 6 overviews our characterization flow with the OpenROAD. The flow does not perform all the steps of a typical VLSI flow (e.g. detailed routing) since the objective is to estimate relative latency/area/power metrics useful to drive the hardware generation algorithms rather than obtaining absolute synthesis results of a full design.

Our exploration of novel functional units has mostly focused to integer operations for now, the premier choice for low-power machine learning inference accelerators. Specifically, we have designed speculative functional units with and without error correction. Differently from the usual concept of speculation in computer architecture (related to branches in control flow), these units speculate on the result of the arithmetic operations (for example, not considering the carry bits in additions), thus potentially allowing to reach higher degrees of parallelism and higher frequencies (or reduced power), at the price of compromised accuracy. These designs also allow combining error correction for the most significant bits, providing additional design tradeoffs. However, these designs require datapath controllers able to deal with variable latency (when an error is detected and the computation is executed). Figure 7 shows the high-level schematic of a speculative adder with error control.

A final key component needed to implement an end-to-end agile and automated design flow is the support for efficient verification, to assure that the generated designs are correct. The SODA toolchain provides this component by leveraging Verilator [39], in both the cases of using Bambu or the LLVM-based Synthesizer middle end. When provided with the input values to synthesized function/kernels (for example, input arguments of a function) in a XML file, Bambu is able to generate Verilog testbenches and scripts to drive and control execution of Verilator. Bambu then executes the "Verilated" models (launches the simulation) and verifies that the output values correspond to the golden results from the execution of the original function code. For the SODA LLVM-based Synthesizer, instead, we have designed and started implementing a C++ modeling tool to connect and drive simulation of "Verilated" hardware accelerators modules with a memory model and the rest of the system.

III. SODA End-to-End Synthesis Results

In this section, we evaluate SODA Synthesizer's effectiveness, flexibility and generality.

A. Effectiveness of SODA

To illustrate SODA's potential for optimization and achievable performance of generated kernels, we outline and optimize a tile of a matrix-multiplication (matmul) algorithm and report post-synthesis results provided by Design Compiler targeting GF 12nm technology node. Table I shows results for the 8x8 by 8x8 8-bit integer (int8) Matmul accelerators generated by SODA. The SODA generated designs A and B vary in target frequency and the available memory channels for synthesis. The latency, reported in cycles, accounts for load, compute, and store operations. For this example, we asked SODA to perform extensive design space exploration (DSE) with all available optimizations turned on. Both designs (A and B) are generated automatically with SODA and achieves

TABLE I
Highly optimized 8x8 by 8x8 Matmul accelerators for int8 data-type generated with SODA. Area and Power results are reported by Design Compiler using GF12nm technology node.

Design ID	A	B
Frequency $[MHz]$	1,000	800
Number of Mem Channels	4	8
Latency $[cycles]$	31	14
Area $[mm^2]$	0.028	0.026
Power $[mW]$	5.28	7.27
Energy Efficiency $[TOPS/W]$	**13**	**17**

TABLE II

RUNTIME RESULTS FOR THE LENET MODEL (FIRST 3 LAYERS) WITH DIFFERENT OUTLINING STRATEGIES.

<table>
<tr><th rowspan="2">Layer type</th><th rowspan="2">Layer params</th><th colspan="3">Outlining Strategy</th></tr>
<tr><th>Individual Ops</th><th>Fused as in TF kernels</th><th>Fused in coarser granularity</th></tr>
<tr><td>Conv2D</td><td>5x5, 6, padded</td><td>2,423,374</td><td rowspan="2">2,388,122</td><td rowspan="3">2,443,073</td></tr>
<tr><td>Activation</td><td>Relu</td><td>39,230</td></tr>
<tr><td>AveragePooling2D</td><td>2x2, 2x2 stride</td><td>88,244</td><td>88,224</td></tr>
<tr><td>Total</td><td></td><td>2,551,848</td><td>2,476,366</td><td>2,443,073</td></tr>
</table>

promising energy efficiency — more than 13 TOPS/W for accelerating the Matmul kernel.

SODA can effectively identify other code snippets that perform Matmul operations and leverage the same optimization passes from this exploration to reuse the synthesized accelerator and achieve similar performance. SODA also warps all transformation passes into a pipeline, exposing it with a single compilation flag to the SODA end user. For Matmul operations with bigger matrix sizes, SODA Synthesizer is able to tile the matrices and and optimizes the tiled code in the same manner. After each synthesis, the user satisfied designs will be cached in the SODA resource libraries, reducing the synthesis time for the same kernel.

B. Benefits from SODA-opt Outlining

To examine *SODA-opt*'s outlining technique, we explore the opportunity of fusing different combinations of operations in the ML models for a single accelerator synthesis and evaluate its performance impact on the generated hardware execution time (i.e., total cycle counts). We performed SODA synthesis of a full LeNet CNN model (Trained on MNIST dataset with 28x28 image size) from TensorFlow, and generated a self-contained Verilog descriptions to perform inference simulation. The synthesized accelerators assume floating-point datatype, a single memory channel and different outlining strategies. Table II summarizes the synthesis and simulation results for the first three layers of the LeNet model. Compared to the design that generates individual accelerators for each layer operation, we observe a 5% performance (cycle counts) improvements by outlining and fusing operations from those layers to a single accelerator execution.

We expect greater benefits from *SODA-opt* outlining when the input models are diverse and thus exhibit more fusion opportunities.

C. Flexibility and Generality of SODA

Finally, to demonstrate the generality of the SODA Synthesizer, we performed a set of synthesis for various commonly seen ML models as summarized in Table III. These ML models span classical ML algorithms (SVM), CNNs, graph and region-based CNNs, and reinforcement learning based models, showing the input flexibility of SODA frontend.

Note that some of these models are too big to be fully synthesized with a single accelerator design. However, with

TABLE III

SUMMARY OF SYNTHESIS RESULTS OF DIFFERENT ML MODELS USING OPENROAD FLOW, TARGETING OPENPDK 45NM TECHNOLOGY NODE.

ML Model	SVM	MiniGo	GCN	ResNet-50	Mask R-CNN
Frequency $[MHz]$	200	200	200	200	200
Area $[mm^2]$	0.036	0.060	0.062	0.305	0.509
Cells	25k	44k	49k	220k	369k

SODA-opt's outlining capability we are able to build accelerator modules starting with small operations (i.e., convolutions and matrix multiplications) in the models. The generated accelerator modules are then integrated to yield the final design, and the area and cell counts after logic synthesis are reported in Table III.

IV. FUTURE RESEARCH OPPORTUNITIES

SODA is still in its active development phase, and we plan to open-source SODA's entire synthesis flow. SODA's end-to-end synthesis infrastructure will facilitate both the AI/ML and hardware design communities and enable the following future research directions.

A. SODA with Neuromorphic Computing

Spiking Neural Networks (SNNs) [26] are designed for machine learning on energy-constrained devices. Neuromorphic hardware such as TrueNorth [5], Loihi [10], and DynapSE [31] implement biological neurons and synapses, making them efficient in executing SNNs. Specialized operators and novel mapping techniques [1], [41], [42] introduced by SNNs create an opportunity for SODA to support the synthesis of SNNs onto neuromorphic hardware. The current implementation of SODA's frontend supports the translation of regular neural network models (NNs, CNNs, DNNs), preparing them for the middle end synthesizer. We plan to design a novel SNN-oriented MLIR dialect along with its lowering techniques in support of SNNs. The new dialect will cover major SNN operators, and the lowering techniques will consider neuromorphic hardware configurations.

B. AI-Assisted SODA

SODA's end-to-end approach enables greater optimization opportunities that form a decision-making sequence along with the synthesis flow. All possible decision choices contribute to a huge design space. Exhaustive search over the

whole space incurs significant compilation overheads. ML techniques have been successfully applied and show superior performance across the stack: system design [15], compilation techniques [6], architectural design space exploration [25], and electronic design automation (EDA) [30]. We anticipate the SODA Synthesizer to be more *intelligent*, i.e., by enhancing SODA with learning and generalization capabilities. Specifically, at the frontend, ML techniques can facilitate the exploration of progressive code lowering strategies in *SODA-opt*; The AI-assisted DSE engine can provide accurate performance predictions for different hardware platforms at earlier stages of the synthesis flow, eliminating the long feedback path from the backend. In addition, ML-based DSE can sample the design space more efficiently, pruning poor design choices. Overall AI-assisted SODA will reduce solution search time and *yield* the optimal designs.

C. Closed-loop Model Hardware Co-design

It is possible to improve ML models' runtime with a series of optimizations such as pruning, quantization, and neural architecture search (NAS) [28], enhancing their inference efficiency while maintaining model's accuracy. Several works have attempted to make such optimizations more effective on real hardware. For example, recent work [17] proposes a model compression and software compilation co-design framework that achieves real-time DNN inference on off-the-shelf mobile devices. [21] proposes a novel hardware and software co-exploration engine for efficient search of both the neural architecture and hardware design space. Furthermore, many hardware templates and improvements designed explicitly for DNN compression and weight sparsity are available [13], [7], [47]. SODA's resource library can incorporate those templates, and SODA frontend can use model (weights) statistics to perform sparsity-aware HLS. Moreover, SODA Synthesizer can be extended and repurposed to co-design ML models and hardware with end-to-end optimizations.

D. Beyond ML and Full SoC Integration

SODA can support full system integration and DSE, i.e., from a discrete accelerator chiplet to the synthesis of accelerators with SoC integration. Analysis of die photos [19] from Apple's A6, A7, and A8 SoCs shows that more than half of chip real estate is dedicated to specialized IP blocks (neither CPUs nor GPUs). We observe a similar trend with other manufacturers such as NVDIA's A102 chips and Qualcomm's Snapdragon chips. Moreover, those IP blocks implement heterogeneous functionalities beyond ML acceleration: camera/image signal processing, video encoding/decoding, cryptography, and encryption acceleration. SODA has the potential to explore this ample design space to understand the tradeoffs across the entire system. SODA's frontend will be updated to leverage MLIR *dialects* that expands its operator coverage. SODA's resource library can leverage components (e.g., AXI interconnects, networks-on-chip, on-chip memories, etc.) from an open-source agile SoC development flow, such as CHIPKIT [44], or ESP [27].

V. Related Work

SODA's modular, multi-level, compiler-based synthesis framework ties closely to two lines of research:

ML compiler infrastructure - Many ML (domain-specific) compiler infrastructures have been developed to minimize the manual effort during the deployment. In general, an ML compiler parses the computational graphs from the high-level frameworks, generating a specific IR, performs optimization passes, and maps it to *existing* hardware backends (CPUs, GPUs, FPGAs, and ASICs). HPVM [22] implements LLVM IR extension and a virtual instruction set (ISA), aiming to enable effective code generation and optimization for heterogeneous systems. Google's XLA [20] is a domain-specific compiler specifically designed to accelerate linear algebra for TensorFlow models. By also leveraging MLIR, XLA generates LLVM IR for CPU/GPU targets and provides dialects for TPU compilation. Glow [36] implements its own IR to support PyTorch models. TVM [9] targets multiple deep learning frameworks and can reuse Halide's [35] existing scheduling primitives and features with a powerful ML-based autotuner. MLIR [24] is an highly extendable and flexible multi-level IR compiler infrastructure as discussed in Section II-A.

High-level synthesis - Inspired by the state-of-the-art software compilation techniques, HLS is an increasingly popular hardware design approach that takes untimed C/C++ as input, extracts IRs with a set of optimizations, performs scheduling/allocation/binding, and *automatically* generates synthesizable RTL that targets FPGAs or ASICs. Most HLS tools [32] repurpose the existing IR from LLVM or GCC and thus expect designers to properly annotate the C code, ensuring the quality of the generated RTL. For this reason, several works [38], [18], [45] have attempted to design better IRs specific for HLS that can improve the ease of use.

Ours - SODA project [43], is an open-source effort to chain together the best practices from MLIR's compiler infrastructure and HLS based approaches. SODA aims to support diverse ML (domain-specific) models. Moreover, SODA provides a uniformed synthesis framework that can expose end-to-end optimization opportunities and minimize the human efforts for generating hardware designs for AI/ML. Recent works [40], [46] also utilize MLIR for HLS but focus primarily on higher-level IR optimization and not end-to-end results.

VI. Conclusion

This paper presents an overview of SODA, an end-to-end synthesizer that enables automatic and agile hardware design for general ML acceleration. We plan to open source the entire SODA infrastructure to facilitate future research across ML, compiler, architecture and hardware design.

VII. Acknowledgment

This research is partially supported by the Defense Advanced Research Projects Agency's (DARPA) Real-Time Machine Learning Program (RTML) and Pacific Northwest National Laboratory's Data-Model Convergence Initiative.

REFERENCES

[1] N2D2: Neural Network Design and Deployment. https://github.com/CEA-LIST/N2D2.

[2] OpenAI's GPT-3 Language Model: A Technical Overview. https://lambdalabs.com/blog/demystifying-gpt-3/#2. Accessed: 2021-05-30.

[3] PandA: on Open Source Framework for Hardware-Software Codesign. https://panda.dei.polimi.it.

[4] Martín Abadi, Paul Barham, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[5] Filipp Akopyan, Jun Sawada, et al. Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(10):1537–1557, 2015.

[6] Jason Ansel, Shoaib Kamil, et al. Opentuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*, Edmonton, Canada, Aug 2014.

[7] Nicolas Bohm Agostini, Shi Dong, et al. Design space exploration of accelerators and end-to-end DNN evaluation with TFLITE-SOC. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 10–19, 2020.

[8] Vito Giovanni Castellana, Antonino Tumeo, and Fabrizio Ferrandi. High-level synthesis of parallel specifications coupling static and dynamic controllers. In *Proceedings of the 35th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, page to appear, 2021.

[9] Tianqi Chen, Thierry Moreau, et al. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[10] Mike Davies, Narayan Srinivasa, et al. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, 2018.

[11] Jeff Dean. Deep learning for solving important problems. In *The World Wide Web Conference (WWW 19)*, page 1. ACM, 2019.

[12] Jeff Dean, David Patterson, and Cliff Young. A new golden age in computer architecture: Empowering the machine-learning revolution. *IEEE Micro*, 38(2):21–29, 2018.

[13] Lei Deng, Guoqi Li, et al. Model compression and hardware acceleration for neural networks: A comprehensive survey. *Proceedings of the IEEE*, 108(4):485–532, 2020.

[14] MLIR Developers. The 'gpu' dialect. https://mlir.llvm.org/docs/Dialects/GPU/. Accessed: 2021-05-27.

[15] Fahimeh Farahnakian, Pasi Liljeberg, and Juha Plosila. Energy-efficient virtual machines consolidation in cloud data centers using reinforcement learning. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 500–507. IEEE.

[16] Fabrizio Ferrandi, Pier Luca Lanzi, et al. Ant colony heuristic for mapping and scheduling task and communications on heterogeneous embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(6):911–924, June 2010.

[17] Hui Guan, Shaoshan Liu, et al. Cocopie: Enabling real-time ai on off-the-shelf mobile devices via compression-compilation co-design. *Commun. ACM*, 64(6):62–68, May 2021.

[18] Sumit Gupta, Rajesh Gupta, et al. *SPARK: a parallelizing approach to the high-level synthesis of digital circuits*. Springer Science & Business Media, 2007.

[19] Harvard Architecture, Circuits, and Compilers Group. Die Photo Analysis of Apple SoCs. http://vlsiarch.eecs.harvard.edu/accelerators/die-photo-analysis. Accessed: 2021-05-27.

[20] Google Inc. XLA is a compiler that optimizes tensorflow computations, 2019.

[21] Weiwen Jiang, Lei Yang, et al. Hardware/software co-exploration of neural architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(12):4805–4815, 2020.

[22] Maria Kotsifakou, Prakalp Srivastava, et al. HPVM: Heterogeneous parallel virtual machine. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 68–80, 2018.

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[24] C. Lattner, M. Amini, et al. MLIR: Scaling compiler infrastructure for domain specific computation. In *CGO*, pages 2–14, Seoul, Korea (South), 2021. IEEE.

[25] Ting-Ru Lin, Drew Penney, et al. A deep reinforcement learning framework for architectural exploration: A routerless NoC case study. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 99–110. IEEE, 2020.

[26] Wolfgang Maass. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10(9):1659–1671, 1997.

[27] Paolo Mantovani, Davide Giri, et al. Agile SoC development with open ESP. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2020.

[28] Gaurav Menghani. Efficient deep learning: A survey on making deep learning models smaller, faster, and better, 2021.

[29] Marco Minutoli, Vito Giovanni Castellana, et al. Svelto: High-level synthesis of multi-threaded accelerators for graph analytics. *IEEE Transactions on Computers*, pages 1–1, 2021.

[30] Azalia Mirhoseini, Anna Goldie, et al. Chip placement with deep reinforcement learning. *arXiv preprint arXiv:2004.10746*, 2020.

[31] Saber Moradi, Ning Qiao, et al. A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (DYNAPs). *IEEE Transactions on Biomedical Circuits and Systems*, 12(1):106–122, Feb 2018.

[32] Razvan Nane, Vlad-Mihai Sima, et al. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2015.

[33] Adam Paszke, Sam Gross, et al. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*, 2019.

[34] Christian Pilato, Antonino Tumeo, et al. Improving evolutionary exploration to area-time optimization of FPGA designs. *Journal of Systems Architecture*, 54(11):1046–1057, November 2008.

[35] Jonathan Ragan-Kelley, Andrew Adams, et al. Halide: Decoupling algorithms from schedules for high-performance image processing. *Communications of the ACM*, 61(1):106–115, 2017.

[36] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

[37] Yakun Sophia Shao, Brandon Reagen, et al. The aladdin approach to accelerator design and modeling. *IEEE Micro*, 35(3):58–70, 2015.

[38] Amirali Sharifian, Reza Hojabr, et al. μIR-an intermediate representation for transforming and optimizing the microarchitecture of application accelerators. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 940–953, 2019.

[39] W Snyder. Verilator. https://www.veripool.org/wiki/verilator, 2003. Online accessed on 05-11-2021.

[40] Stephanie Soldavini and Christian Pilato. Compiler infrastructure for specializing domain-specific memory templates. *arXiv preprint arXiv:2104.01448*, 2021.

[41] Shihao Song, Anup Das, and Nagarajan Kandasamy. Improving dependability of neuromorphic computing with non-volatile memory. In *2020 16th European Dependable Computing Conference (EDCC)*, pages 17–24, 2020.

[42] Twisha Titirsha, Shihao Song, et al. Endurance-aware mapping of spiking neural networks to neuromorphic hardware. *IEEE Transactions on Parallel and Distributed Systems*, pages 1–1, 2021.

[43] Antonino Tumeo, Marco Minutoli, et al. Software defined accelerators from learning tools environment. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[44] P. Whatmough, M. Donato, et al. Chipkit: An agile, reusable open-source framework for rapid test chip development. *IEEE Micro*, pages 1–1, 2020.

[45] Qiang Wu, Yunfeng Wang, et al. A hierarchical CDFG as intermediate representation for hardware/software codesign. In *IEEE 2002 International Conference on Communications, Circuits and Systems and West Sino Expositions*, volume 2, pages 1429–1432. IEEE, 2002.

[46] Hanchen Ye, Cong Hao, et al. Scalehls: Achieving scalable high-level synthesis through mlir. 2021.

[47] Jeff Zhang, Parul Raj, et al. CompAct: On-chip compression of activations for low power systolic array based CNN acceleration. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–24, 2019.