End-to-end Synthesis of Dynamically Controlled Machine Learning Accelerators

Serena Curzel[§‡], Nicolas Bohm Agostini[†‡], Vito Giovanni Castellana[‡], Marco Minutoli[‡], Ankur Limaye[‡], Joseph Manzano[‡], Jeff (Jun) Zhang[*], David Brooks[*], Gu-Yeon Wei[*], Fabrizio Ferrandi[§], 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—Edge systems are required to autonomously make real-time decisions based on large quantities of input data under strict power, performance, area, and other constraints. Meeting these constraints is only possible by specializing systems through hardware accelerators purposefully built for machine learning and data analysis algorithms. However, data science evolves at a quick pace, and manual design of custom accelerators has high non-recurrent engineering costs: general solutions are needed to automatically and rapidly transition from the formulation of a new algorithm to the deployment of a dedicated hardware implementation. Our solution is the SOftware Defined Architectures (SODA) Synthesizer, an end-to-end, multi-level, modular, extensible compiler toolchain providing a direct path from machine learning tools to hardware. The SODA Synthesizer frontend is based on the multilevel intermediate representation (MLIR) framework; it ingests pre-trained machine learning models, identifies kernels suited for acceleration, performs high-level optimizations, and prepares them for hardware synthesis. In the backend, SODA leverages state-of-the-art high-level synthesis techniques to generate highly efficient accelerators, targeting both field programmable devices (FPGAs) and application-specific circuits (ASICs). In this paper, we describe how the SODA Synthesizer can also assemble the generated accelerators (based on the finite state machine with datapath model) in a custom system driven by a distributed controller, building a coarse-grained dataflow architecture that does not require a host processor to orchestrate parallel execution of multiple accelerators. We show the effectiveness of our approach by automatically generating ASIC accelerators for layers of popular deep neural networks (DNNs). Our high-level optimizations result in up to 74x speedup on isolated accelerators for individual DNN layers, and our dynamically scheduled architecture yields an additional 3x performance improvement when combining accelerators to handle streaming inputs.

1 Introduction

Next-generation edge systems will operate under conditions where exporting all the acquired data for centralized processing is inconvenient or impossible [1]. Monitoring infrastructure for highly dynamic systems (e.g., sensor networks) will need to operate in low power settings with limited bandwidth available for communication [2]. Autonomous vehicles will need to make critical decisions in real-time in a distributed setting. Experimental instruments such as the ones owned by the US Department of Energy (e.g., particle accelerators, mass spectrometers, and electron microscopes), already generate volumes of data that are impossible to store or transfer without pre-processing [3]. Such extreme conditions require highly specialized processing systems to support autonomous learning and artificial intelligence, optimized along a variety of metrics that include energy, performance, latency, size, and more. Designing and implementing domain-specific systems is challenging and expensive due to the extreme diversity and fast-paced growth of applications and algorithms, especially in the field of machine learning (ML). There is no "one-size-fits-all" solution, and developing specialized accelerators requires significant efforts by large teams of expert hardware designers.

To address these problems, we have developed the SOftware Defined Architectures (SODA) Synthesizer [4], [5], [6]: an open-source, multi-level, modular, extensible, no-human-in-the-loop hardware compiler that translates high-level ML models into domain-specific accelerators. Our tool generates highly specialized designs in a hardware description language (HDL), which can be synthesized with both commercial and open-source tools on field programmable gate arrays (FPGAs) or as application-specific integrated circuits (ASICs). The SODA Synthesizer comprises a compiler-based frontend that leverages the Multi-Level Intermediate Representation (MLIR) framework, and a compiler-based backend that integrates state-of-the-art high-level synthesis (HLS) methodologies. The SODA Synthesizer allows for the exploration of design metrics through compilation passes and parameters and enables identification of optimal trade-offs depending on the target application requirements.

HLS tools typically generate highly-specialized, power-efficient hardware designs using the finite state machine with datapath (FSMD) paradigm, which is particularly suited for extracting instruction-level parallelism. However, the FSM controller has a notable limitation: it is not scalable enough to deal with multiple, parallel, execution flows (e.g., in presence of coarse-grained parallelism). In these conditions, which are common for compute and memory-intensive ML algorithms (e.g., deep neural networks), the complexity of a centralized, statically scheduled FSM controller grows exponentially, leading to significant area and

performance overheads [7]. A system-on-chip (SoC) can use a central general-purpose microcontroller to drive multiple accelerators implementing different layers of an ML model; however, in such a system the data movement between the host microcontroller, the accelerators, and the memory quickly becomes a performance bottleneck. In this work, we have extended the SODA Synthesizer to enable automatic generation of a second type of system: a dynamically scheduled architecture where custom ML accelerators (based on the FSMD model) are composed in a data flow 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.

In a previous work [8], we implemented a solution to synthesize parallel C code, annotated with OpenMP-like directives, into a similar data flow architecture, with support for spatial parallelism, resource reuse, and memory access parallelism. That approach could identify certain degrees of parallelism by analyzing program dependencies, but it was constrained by conservative alias analysis: user-provided annotations were needed to simplify the dependency analysis and to expose dynamic parallelism. ML frameworks, instead, naturally represent models as computational graphs describing how the data flows across operators, and MLIR directly interfaces with ML frameworks, offering promising opportunities for domain-specific optimizations. By leveraging the MLIR framework, the SODA Synthesizer can take advantage of such optimization opportunities. MLIR representations capture hierarchy and parallelism of computational graphs, facilitating generation and mapping to data flow architectures. Knowing precisely how the data flows across operators and memory regions removes the need for complex alias analysis.

In summary, the contributions of this paper are:

- an automated, modular, multi-level, compiler-based design flow from high-level ML frameworks to optimized FPGA or ASIC accelerators implemented following the FSMD model;
- a search and outlining methodology to automatically extract accelerators and their dependencies from an MLIR input specification;
- a system integration methodology to assemble FSMD accelerators into a coarse-grained, dynamically scheduled data flow architecture with distributed control;
- a comparison between a standard SoC design with a centralized microcontroller and a custom system built with our distributed controller methodology.

The rest of the paper is structured as follows: we summarize related work in Section 2, the SODA Synthesizer is introduced in Section 3, and detailed in Sections 4-5. We show experimental results in Section 6 and draw conclusions for the paper in Section 7.

2 Related Work

A large number of designs (including several based on the data flow model) have been proposed as specialized ML accelerators, and existing HLS tools have been extended with data flow concepts before. In this section, we summarize relevant previous works and highlight the differences between existing approaches and our work.

2.1 Hardware acceleration for machine learning

Available commercial solutions offer acceleration of ML algorithms through specialized functional units (e.g., the Tensor Cores in NVIDIA GPUs) or entire chips based on tensor processing (e.g., Google TPU [9]). Some of them, including SambaNova, GraphCore, and Cerebras, exploit the data flow paradigm, with varying degrees of generality in their processing elements. Research and industry also proposed many FPGA-based accelerators for ML inference [10], [11], often supporting specialized numeric formats to reduce resource utilization and increase efficiency.

One challenge is to design an accelerator that can support multiple classes of algorithms, rather than focusing solely on deep neural networks (DNNs) [12]. Efforts in this direction include the PuDianNao [13] and SpiNNaker [14] architectures, or the Tabla [15] and Eyeriss [16] frameworks for the generation of accelerators. DNNBuilder [17] and the related design space exploration flow DNNExplorer [18] employ configurable and composable layer-wise accelerators to implement several types of DNNs. SIGMA [19] exploits reconfigurable interconnect with specialized matrix multiplication units. Other designs with reconfigurable interconnect also support data flow models [20], [21]. Rather than proposing yet another accelerator design, we provide a methodology to design and implement new FPGA/ASIC accelerators starting from a high-level description of the input ML algorithm. By leveraging high-level and low-level (HLS) compiler-based tools, SODA provides a more general solution: in fact, it can generate hardware designs for virtually any computational pattern, as long as a lowering to MLIR is available.

One common approach to reduce design efforts of processing elements at the register-transfer level is to compile and map a high-level description of the input algorithm onto parametrized hardware modules and architecture templates. VeriGOOD-ML [22] uses the PolyMath compiler [23] to map ML models in the ONNX format to three different architecture templates designed for different types of neural networks. GEMMINI [24] offloads operations from specific layers of ONNX models to a systolic array connected to a RISC-V core, after building the systolic array itself starting from a parametrized generator in Chisel. TVM's VTA architecture [25] is a configurable FPGA co-processor for matrix multiplication; the TVM high-level framework then compiles each ML model into instructions for VTA. All these solutions can generate specialized accelerators, but they can only support ML layers and operators that have a direct mapping to the available hardware templates.

Alternative approaches translate high-level abstractions into a form that can be ingested by commercial HLS tools (typically, C/C++ code with tool-specific optimization directives). PyLog [26] defines a high-level compilation infrastructure to transform Python programs into annotated C/C++ code, which is subsequently fed to Xilinx Vivado HLS. HeteroCL [27] provides a Python-based domain-

speci c language to partition an algorithm between general-purpose processor and FPGA, and to insert hardware-speci c information in the code, which is then compiled into annotated C/C++ for different backend HLS tools. The hls4ml [28] framework translates input models selecting operators from a library of C/C++ templates optimized for Vivado HLS. ScaleHLS [29] aims at facilitating and optimizing HLS through high-level transformations implemented in MLIR, exploiting different levels of abstraction and - nally generating annotated C code for Vivado HLS. These tools provide a bridge between high-level programming frameworks and hardware generation, but they have limited exibility: they only support speci c high-level frameworks and backend HLS tools, and they generate code at a different (higher) level of abstraction after applying hardware-related optimizations, potentially losing a considerable amount of semantic information in the process. With SODA, instead, we bring together MLIR and HLS to build an integrated open-source toolchain, optimizing input ML models at appropriate levels of abstractions, without the need to generate intermediate C/C++, and offering a wide choice of FPGA and ASIC targets in the backend.

2.2 Generation of data ow accelerators

Our methodology generates a custom architecture that dynamically invokes, in a data ow fashion, highly optimized, statically scheduled accelerators based on the FSMD model. This requires a distributed controller that activates each module as soon as its inputs are available. Previous HLS research tried to decompose and distribute the classical FSM controller to reduce its complexity, restructuring it in a hierarchical way [30], but this is not very ef cient in managing concurrent execution of independent units.

The Bluespec compiler [31] implements an event-driven execution paradigm based on rules and atomic transactions that is similar to our approach. Our approach performs the synthesis of an event-driven data ow architecture starting from outlined kernels in an MLIR description, i.e., from a high-level abstract representation of the application code. The Bluespec compiler instead uses as input BSV, a language that, while higher-level than Verilog and VHDL, is closer to a behavioral HDL than to a software description. Additionally, our kernels are FSMD accelerators, rather than functional units. Dynamatic [32] proposes an HLS methodology that generates dynamically scheduled designs using the data ow paradigm, mainly focusing on supporting data ow at the instruction level, rather than at the task/function level. It does not support resource sharing, and abstracts memory by decoupling it from the accelerator through a single load/store queue, thus not taking advantage of memory-level parallelism. Dynamic scheduling leads to simpler designs when exploiting parallelism across basic block boundaries; however, FSMDs provide very high quality of results (both in performance and area) when the focus is optimizing for instruction-level parallelism inside a function or a basic block. Dynamatic has been extended to couple dynamic with static scheduling [33], supporting resource reuse and a simple memory abstraction, but again their solution does not consider coarser-grained parallelism in the input speci cations, and it only works on C inputs.

Fig. 1. The SODA Synthesizer and its interfaces towards external tools, with details on the components extended for the generation of data ow architectures.

Several other research projects are proposing domain-speci c languages and frameworks to generate accelerators that can exploit coarse-grained parallelism by combining data ow concepts with FSMDs, as we do in our methodology. For example, Spatial [34] allows to mark both data ow modules (akin to our distributed controller) and FSM modules at different levels of the code hierarchy. Xilinx Vivado/Vitis HLS tools support data ow pipelining mechanisms across functions or loops by annotating the input C speci cation with a custom pragma [35]; while the solution allows overlapping execution of functions and loops in a pipelined fashion, it only works when a similar initiation interval for all the functions/loops can be found. All these approaches still require users to describe to some extent the behavior and desired features of a circuit in their code, while the SODA Synthesizer provides a completely automated path from high-level frameworks to hardware, with no additional information required.

3 The SODA Synthesizer

Figure 1 provides an overview of the SODA Synthesizer [4], [5], [6]; components extended to support the generation of data ow architectures are highlighted in blue. The tool is composed of two main parts: a compiler-based frontend and a compiler-based hardware synthesis engine. The frontend is based on MLIR [36], a framework that allows building reusable compiler infrastructure inspired by (and contributed to) the LLVM project. The SODA Synthesizer frontend interfaces with high-level programming frameworks, partitions the input applications by identifying key computational kernels for hardware acceleration, and performs high-level optimizations that improve the subsequent generation of custom accelerators and systems. The frontend then generates an LLVM IR as output, which is the starting point for hardware generation. The SODA backend integrates

Fig. 2. Structure of the SODA-OPT high-level compilation frontend, with emphasis on the components extended for the generation of data ow architectures.

Bambu [37], a state-of-the-art open-source HLS tool, to generate the hardware accelerators. To compile code that will be executed on a host processor, instead, SODA uses standard LLVM tools. The frontend compiler and HLS backend are available at https://gitlab.pnnl.gov/sodalite/soda-opt and https://github.com/ferrandi/PandA-bambu, respectively.

3.1 Frontend compiler

SODA-OPT (Figure 2) is the high-level compilation frontend of the SODA Synthesizer. SODA-OPT performs search, outlining, optimization, and dispatching passes on the input program, preparing it for hardware synthesis targeting FPGAs or ASICs. To implement all these functionalities, SODA-OPT leverages and extends the MLIR framework. MLIR allows developers to de ne dialects, i.e., self-contained IRs that respect MLIR's meta-IR syntax. Dialects model code at different levels of abstraction, creating specialized representations that facilitate the implementation of new compiler optimizations. For example, dialects that are maintained in tree along with the MLIR framework include abstractions for linear algebra, polyhedral analysis, structured control ow, and others. We will refer to these as built-in dialects in the rest of the paper.

Built-in dialects are the entry points to the SODA Synthesizer frontend. SODA-OPT introduces new constructs speci c to hardware generation, but it also exploits existing dialects and optimizations: this enables high-level programming frameworks to leverage our toolchain just by providing a translation to built-in MLIR dialects. Several frameworks already implemented their own speci c MLIR dialects, optimization passes, and lowering methods, including TensorFlow, ONNX-MLIR, and TORCH-MLIR. One entry point to the SODA synthesizer is through the TensorFlow tf-mlir-translate and tf-opt tools, which compile ML models de ned and trained in TensorFlow into an MLIR representation.

SODA-OPT implements analysis and transformation passes that parse MLIR inputs lowered from high-level frameworks, identify key operation groups, and outline them into separate MLIR modules. The operations selected for hardware acceleration undergo an optimization pipeline with progressive lowerings through different MLIR dialects (linalg ! affine ! scf ! std ! llvm), and they are nally translated into an LLVM IR purposely restructured for hardware generation. SODA-OPT can lower the remaining operations in two different ways, depending on the desired target: they can represent the orchestrating code executed by a host microcontroller, or the relationship between the accelerators in our data ow architecture. In the rst case, SODA-OPT produces another LLVM IR le including runtime calls to control the generated accelerators. In the second case, operations are transformed into a function-based representation (task graph) that allows Bambu to generate the required distributed controller logic and memory interface; accelerators and controller modules will then be assembled together to form the data ow architecture.

3.2 High-Level Synthesis backend

The SODA Synthesizer backend, Bambu, leverages state-of-the-art HLS techniques to synthesize the LLVM IR produced by the SODA-OPT frontend into an accelerator design. Bambu includes frontends based on standard open-source compilers (GCC or Clang), supporting C, C++ and, among others, LLVM IR inputs. It builds an internal IR and performs HLS steps such as resource allocation, scheduling, and binding, and nally generates the designs in a hardware description language (Verilog or VHDL).

Bambu synthesizes Register-Transfer Level (RTL) designs in Verilog following the nite state machine with datapath (FSMD) model, and we have extended it with novel methodologies that enhance modularity and generate dynamically scheduled accelerators. We enabled the reuse across an entire design of synthesized modules representing functions within a larger speci cation [38], providing opportunities for modular and hierarchical designs. We further extended Bambu to allow the integration of FSMD modules as processing elements in a coarse-grained data ow design [8], and in multithreaded parallel accelerators [39]. We initially developed these synthesis methodologies by integrating support for parallel C speci cations annotated with a set of OpenMP directives: users identify parallel sections in the input code through annotations, allowing Bambu to generate custom accelerator modules, and to combine them in a top-level, dynamically scheduled architecture. MLIR descriptions are naturally parallel and hierarchical, and the MLIR framework facilitates the implementation of the required analyses and transformations. Hence, a multi-level, extensible compiler approach as the one we implemented in SODA-OPT provides opportunities to signi cantly improve system-level design: identifying kernels that need to be accelerated, analyzing their interactions, and composing them in a system are tasks that are better solved at the MLIR level, allowing the HLS engine to focus on the generation of optimized accelerators.

The resource library provides Bambu with RTL descriptions of functional units to implement the operations present in the IR (adders, subtractors, multipliers, etc.), with different versions for different data types. It also contains the architectural templates, controller logic, and interfaces that enable the integration of synthesized modules in a top-level

design. To effectively drive synthesis algorithms, Bambu relies on a characterization process for the components in the resource library in terms of performance (e.g., latency of the critical path) and area for each target technology or device.

Bambu provides several options to connect accelerators to memories: for example, it can generate one read and one store port for a whole module, or read and store ports for each argument of the module (if they do not alias). Then it instantiates and connects multi-ported scratchpads (or BRAMs for FPGAs) to such ports. By default, Bambu connects a dual-ported scratchpad memory to each couple of load store ports, assuming a fixed latency of 1 clock cycle for reads and 2 for stores.

The SODA toolchain interfaces with both commercial and open-source logic synthesis tools. Bambu supports FPGA devices from several vendors, and we introduced the option of targeting ASICs through the OpenROAD flow, employing the OpenPDK 45 nm cell technology library. Thus, the SODA toolchain provides a completely open-source, end-to-end compiler-based hardware generation flow from high-level programming environments to silicon. We also added support for the Synopsis Design Compiler, targeting both the OpenPDK 45 nm and the Global Foundries 12/14 nm technology nodes. For each new tool and technology, we ran the Bambu characterization process, collecting all area and performance metrics needed to update the resource library and the models estimating interconnections cost.

Finally, the SODA toolchain also provides verification features to ensure 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. One of these is the open-source tool Verilator [40], which generates optimized models for the accelerators that it simulates, and drives them through C++ or SystemC top modules. The SODA frontend feeds simulation inputs to Bambu; Bambu, in turn, generates testbenches, scripts, and glue code to drive the execution of Verilator, and automatically verifies that the output values of the simulated accelerators correspond to the results from the execution of the original application with the same inputs.

4 Kernel Selection and Optimization

As mentioned at the beginning of Section 3, the SODA-OPT frontend performs search, outlining, optimization, and dispatching passes to select relevant kernels from the input model, and prepare them for hardware generation. We exploit existing and custom MLIR dialects, leveraging the possibility of working at different levels of abstraction in different stages of the compilation process. For example, high-level built-in dialects such as `linalg` and `tosa` maintain semantics from the input specification (e.g. ML operators) that simplify the identification of kernels, while lower-level abstractions such as `affine` and `scf` (also built-in) provide opportunities for code optimizations. We introduced the `soda` dialect to partition input ML models into kernels that will be translated into hardware accelerators, and logic that controls their execution. Table 1 describes the `soda` dialect operations; in the following we will detail the search and outlining processes that use them, and the subsequent optimization and dispatching phases.

TABLE 1
The SODA dialect operations

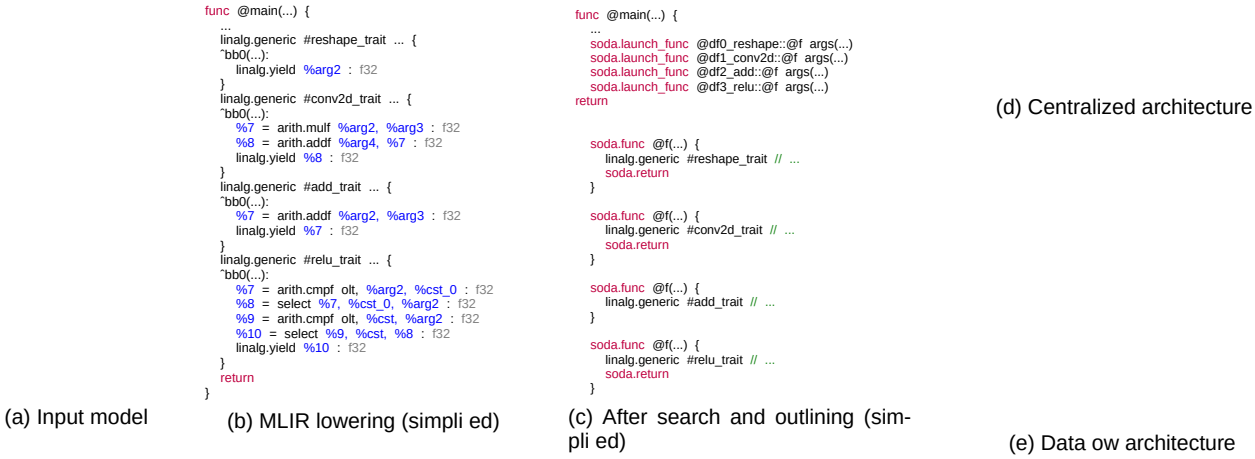
Operation	Semantic
`soda.launch`	Marks MLIR operations to be outlined and extracted into a kernel.
`soda.terminator`	Indicates the end of the operations to be outlined and extracted.
`soda.module`	Holds the list of outlined operations, it will become a unique accelerator module.
`soda.func`	Defines an outlined function with its interface.
`soda.return`	Indicates the end of an outlined function.
`soda.launch_func`	Calls the outlined function from the control logic partition of the code.

4.1 Search phase

SODA-OPT automatically identifies operations that are well suited for acceleration by matching key patterns at the earliest stages of the compilation process (*search* phase). Searched patterns are mostly linear algebra operations or affine structures wrapping arithmetic operations, selected among the most common computation units in ML applications. Users can easily extend SODA-OPT by adding new patterns of interest beyond ML, as could happen when the input is a scientific computing application lowered from a domain-specific framework. Search passes wrap a `soda.launch` operation around the operations to be outlined, and inject a `soda.terminator` operation at its end. Looking at Figure 3a, representing a small portion of a CNN, a user might decide to separately accelerate each node in the computational graph (one reshape operation, one convolution, one bias add, and one ReLu activation function). When the model is lowered to the MLIR `linalg` dialect, each of them is represented by a `linalg.generic` construct (Figure 3b), which SODA-OPT can mark with launch and terminator operations. When targeting the data flow architecture, SODA-OPT individually marks for outlining all operations in the MLIR file, so that each of them will be synthesized as a data flow stage.

4.2 Outline phase

Then, at the beginning of the outlining phase, SODA-OPT extracts each region of code within marks into a separate MLIR module, inlining any functions invoked inside it. SODA-OPT adds an attribute to the module to indicate the target architecture (centralized or data flow), and to later select the corresponding backend compilation/synthesis flow. The outlining process proceeds by analyzing use-def chains of values inside each module to generate the interface of the top-level kernel functions, adding to their arguments also memory buffers allocated outside the `soda.launch` region, but referenced inside it. Constant values are instead pulled inside the kernel. The process ends with the generation of a `soda.module` containing a `soda.func` replacing each `soda.launch` block. Outlined kernels are finally substituted by `soda.launch_func` operations in the top-level code that will orchestrate their execution (Figure 3c).



(a) Input model (b) MLIR lowering (simpli ed) (c) After search and outlining (simpli ed) (d) Centralized architecture (e) Data ow architecture

Fig. 3. Intermediate steps in the SODA compilation process, from ML model to accelerator architecture.

TABLE 2
High-level optimizations in SODA-OPT and their effect on the hardware generation process.

Goal	Implications for HLS	Optimization Passes
Single basic block containing the compute intensive part of the kernel	More freedom to schedule operations	Tiling, Unrolling
Increased instruction-level parallelism	Schedule multiple independent compute operations on the same cycle, as soon as their inputs are available	Unrolling
Increased data level parallelism	Schedule multiple memory operations on the same cycle, if they access different memory units	Tiling, Unrolling, Temporary buffer allocation, Alias analysis
No unnecessary reads from kernel arguments	Reduce expensive accesses to external memory	Temporary buffer allocation, Alloca buffer promotion
Reuse read and compute results	Keep values loaded from memory and intermediate results in registers, rather than repeatedly accessing memory	Scalar replacement of aggregates
No redundant or unnecessary operations	Avoid wasting resources and cycles	Common sub-expression elimination, Dead code elimination

4.3 Optimization phase

After outlining, each kernel is optimized separately, passing through progressive lowering steps that transform its code into LLVM IR. SODA-OPT exploits several dialect-speci c optimization passes from built-in dialects, together with some custom, HLS-oriented transformations. It provides a modular optimization pipeline that restructures the kernels so that the nal low-level IR is well suited for hardware synthesis. The main available optimizations are summarized in Table 2: loop unrolling increases instruction-level parallelism, loop tiling can balance computation and data movement, alias analysis adds opportunities for data-level parallelism, and other typical compiler optimizations remove unnecessary operations (scalar replacement of aggregates, dead code elimination, common sub-expression elimination). Temporary buffer allocation and alloca buffer promotion are custom SODA-OPT optimizations that reduce expensive accesses to external memory by generating registers or memories internal to the kernel that allow to reuse values from input arguments. The optimized LLVM IR presents simpler dependency chains, few or no redundant instructions, and regular load-compute-store patterns: such characteristics improve the resource allocation and static scheduling of operations performed by the HLS engine, resulting in signi cant performance gains. The pipeline is not monolithic: developers can easily enable, disable, reuse, or modify optimizations, providing ample opportunities to customize the process for different applications and implement automated exploration strategies. In fact, optimizations signi cantly in uence the generated hardware designs in terms of performance, area, and power, and they are all implemented as compiler passes: users can thus perform an exhaustive exploration of the design space without manual interventions on the code.

4.4 Dispatch phase

Dispatching separates the kernels from the logic that orchestrates their execution: at the end of the compilation, SODA-OPT generates a separate le for each kernel that does not contain references to the rest of the code, and collects all orchestrating logic in another le. Bambu generates an FSMD accelerator for each of the IR les containing the kernels, later integrated in one of two possible system-level architectures. We currently target two types of architectures: a conventional system-on-chip where a microcontroller drives one or more accelerators connected through a bus (centralized architecture, Figure 3d), and a single accelerator where kernels are connected together in a

Fig. 4. Execution Manager (EM) components, interfacing with Resource Manager (RM)

dynamically scheduled data flow architecture (Figure 3e). In the first case, the orchestrating logic extracted by SODA-OPT will contain function calls for the outlined kernels, which will be substituted by driver calls to the corresponding accelerators in the compiled host program. Instead, when targeting the data flow architecture, SODA-OPT generates a task graph representing interactions between the kernels, containing information that will be used to assemble the accelerators and the distributed controller. In particular, the task graph includes the name of each kernel with the direction (input/output) of its arguments, and the sizes of exchanged data structures retrieved by leveraging the memref dialect.

5 Dataflow Architecture Generation

The process to generate the dynamically scheduled data flow architecture starts by instantiating Distributed Controller (DC) components, that will activate FSMD accelerators at runtime. The DC starts the execution of each FSMD (synthesized from the kernels outlined by SODA-OPT) according to data dependencies described in the task graph (also generated by SODA-OPT). The generation process then continues by instantiating a Hierarchical Memory Interface (HMI) that manages concurrent memory access to a shared memory from multiple accelerators. The designs of the DC and of the HMI derive from the ones presented in [8], but they are now integrated in the SODA Synthesizer where the generation process can take advantage of the outlining, analysis, and transformations performed by SODA-OPT.

5.1 Distributed controller

The DC employs dedicated hardware components to check, at runtime, when to start the execution of the FSMD accelerators, allowing concurrent execution of multiple modules even when their latency depends on the inputs, or they simultaneously access a shared memory. In this way, the DC allows pipelined execution of kernels, which is essential to run ML inference on streaming inputs with low latency.

The DC generation flow instantiates a dedicated component named Execution Manager (EM, Figure 4) for each FSMD accelerator. The EM collects token signals and triggers the execution of FSMD accelerators once the activating conditions for an operation are verified. Specifically, EMs are composed of three parts: the Activation Manager (AM), the Operation Manager (OM), and the Status Manager (SM). The AM is responsible for collecting token signals denoting completion of producer operations and verifying whether they correspond to an activating condition. Activating conditions for each FSMD accelerator are derived from the data dependencies between operators in the task graph provided by SODA-OPT: once all necessary tokens are received, the AM notifies the OM to start execution. In case the associated module is shared among multiple operations, the OM checks for resource availability by interacting with a dedicated arbiter, the Resource Manager (RM). When execution of and FSMD accelerator starts, the SM sends required control signals to the accelerator, and prevents the RM from accepting new requests until the operation is completed, i.e., when a completion signal (FU-done in Figure 4) is received from the module itself. Each FSMD accelerator produces the completion signal which notifies that the output of the operation is ready for its consumers.

The FU-done signal is received by all EMs bound to the same shared module; however, it is ignored if the SM does not indicate the operation is running. This procedure allows each EM to discriminate between the end of their associated operation and the end of other operations mapped on the same module. Finally, the EM emits OP-done token signals to notify the end of the execution to EMs associated with consumer operations. All EM components are based on combinational logic, so they do not add delay cycles to the execution time of the FSMD modules.

In statically scheduled designs, operations that execute concurrently are not allowed to share the same hardware module, thus avoiding resource conflicts. Instead, in our data flow design, RMs dynamically resolve resource conflicts at runtime. During the synthesis process, the module binding phase maps operations to resources: in our case, operations are neural network layers (or other coarse-grained linear algebra algorithms), and resources are the statically scheduled FSMD accelerators. Module binding aims at heuristically reducing the number of resource conflicts stalling the execution. Binding is implemented with an heuristic algorithm that solves the clique covering problem on a Weighted Compatibility Graph (WCG) [41] where nodes represent operations, and edges represent compatibility relations (i.e., if two operations are connected, they can share a hardware resource). While clique covering is an NP-complete algorithm, we use a well-known heuristic on relatively small graphs: nodes in our approach correspond to layers in a neural network, or in any case to large portions of the input application. Typically the size of the graphs is at most in the hundreds of nodes, allowing the module binding phase to complete in few minutes on the largest cases.

After binding, the distributed controller generation process defines tie-breaking rules for the RMs, determining how to resolve structural conflicts that may occur at runtime if operations concurrently request the same module. When operations require the same module at different times, there is no competition and RMs simply process the requests following the order of arrival. Our implementation defines the tie-breaking rules based on the topological order of the operations in the input task graph. A different method may lead to a different execution order, but the execution output would remain the same because the system is built to respect dependencies between operations.

The high regularity of the DC architecture facilitates

Fig. 5. Data ow accelerator schematic for the model of Figure 3.

the automated synthesis process. The synthesis process allocates one RM for each shared module, according to the results of module binding. Then, it traverses the call graph, instantiating an EM for each operation, with custom AMs synthesized according to the operation dependencies. Figure 5 shows a schematic of the overall architecture design for the ML model of Figure 3. After SODA-OPT optimization and dispatching, the task graph contains four calls to the different kernel functions. Bambu synthesizes the four kernel functions using the standard FSMD approach, and the necessary DC components from the task graph describing the dependencies between functions. FSMD modules will then be assembled with their EMs, RMs, and with the memory interface (Section 5.2).

5.2 Hierarchical memory interface

Fig. 6. Hierarchical Memory Interface architecture.

After generating the datapath and the DC, the accelerators need to connect to the memory. Our dynamically scheduled design leverages a specialized memory architecture (hierarchical memory interface, or HMI) to manage concurrent access to shared memory from independent accelerators. The HMI is a multi-ported memory controller that dynamically assigns concurrent memory requests from different resources to multiple external memory channels, computing destination addresses at runtime with no additional delay. If the destination addresses of different memory operations collide, the HMI serializes the memory accesses. The HMI extends the design of the custom Memory Interface Controller (MIC) described in [42]. It is composed of several replicated memory interfaces (MIs), interconnected in a chain. Each MI performs only one memory operation at a time, but all MIs can operate in parallel. The concept of hierarchy appears in the way signals are propagated across the architecture. Figure 6 shows the schematic representation of the HMI for two modules x and y. Additional modules would be chained in the same way. Each MI provides the following ports:

1) sel_store: write access request;
2) sel_load: read access request;
3) addr: memory address;
4) w_data: data to write;
5) r_data: loaded data;
6) ready: completion of the memory access.

The top-level module is the only one that directly interfaces with the memory. The propagation scheme requires that only one module at a time sets sel_store and sel_load signals, which identify memory access requests. Statically scheduled designs ensure this behavior by pre-determining the operations order and executing only one operation at a time. However, this can degenerate in sequential execution of modules that could instead execute simultaneously for part of their computation. We avoid this issue by integrating additional control logic in the HMI that exploits the presence of the RM and SM blocks from the data ow architecture. An RM intercepts memory access requests (req). If it accepts a request, it noti es a dedicated SM component, associated with the MI of each module. For example, in Figure 6, SM_x is associated with the MI of module x, while SM_y is associated with the MI of y. If the top module encapsulating x and y also needed to access memory, a third SM and a third MI would simply be added to the arbitration scheme.

Load and store ports from communicating FSMD accelerators are connected to the HMI which, in turn, connects them to a multi-ported shared memory. Our data ow design can either connect to high-performance multi-banked scratchpad memories or to external multi-ported DRAM controllers (e.g., Xilinx AXI DRAM controllers for FPGAs). SODA-OPT analysis passes compute the amount of data exchanged between kernels, and consequently determine the required size of the shared memory, accounting for double buffering and concurrent execution of the accelerators.

6 Experimental Results

In this section, we present results of our end-to-end hardware generation ow. We rst synthesize isolated ML operators from representative DNN models, and then we evaluate the difference between the two available architectures (centralized and data ow) composed of multiple kernels.

In all experiments we maintained the following setup: we target ASIC devices at the 45 nm technology node through the OpenRoad ow, with an operating frequency of 500MHz. We use Bambu with its Clang12 frontend and O2

(a) ResNet50 block

(b) MobileNetV2 block

Fig. 7. DNN Operators used during the experimental campaign (compute intensive subset of representative DNNs).

(a) ResNet50 layers

(b) MobileNetV2 layers

Fig. 8. Speedup, area overhead, and power overhead obtained through SODA-OPT high-level optimizations.

optimization level. Each synthesized kernel has two ports connecting it to a shared memory with 2 cycles read latency and 1 cycle write latency. Models are synthesized using 32-bit oating point units. Our ow is also able to generate several solutions for memory interfacing, including instantiating dedicated load/store ports for each input/output argument to an operator, or a parametric number of load-/store ports. However, for this analysis, we only employ two ports per accelerator, because we then combine them in larger designs with multiple intercommunicating accelerators. Hence, in the complete architectures, memory parallelism is exploited by having different accelerators operating concurrently, limiting growth of the HMI complexity.

6.1 Effectiveness of the Optimization Pipeline

We automatically outline and synthesize individual operators from the ResNet50 and MobileNetV2 DNN models (Figure 7), in two different con gurations. In the baseline con guration, we outline, lower to LLVM IR, and synthesize each kernel without applying optimization passes. In the optimized con guration, we add the SODA-OPT high-level optimization pipeline, with the goal of reducing execution time. As discussed in Section 4, SODA-OPT automatically optimizes IRs, for example, to present increased instruction-level parallelism and reduced number of redundant instructions. The transformations allow Bambu to compute ef cient schedules and to best leverage the available hardware resources during HLS, as shown by the increase in performance.

By enabling the high-level optimizations in SODA-OPT, we observe an average speedup of 7.2x in the execution time (clock cycles) over the baseline for ResNet50 layers (Figure 8a). For operations from the MobileNetV2 model, we see an average speedup of 23.5x over the baselines, with peaks of 52-74x in the convolutional layers (Figure 8b). Table 3 shows the post oorplanning characteristics of the optimized accelerators generated by the SODA Synthesizer. We computed the ef ciency (GFLOPS/W) by counting the total number of oating point arithmetic operations performed during the whole execution of an accelerator, divided by execution time and power consumption reported by OpenROAD after oorplanning. All the accelerators provide ef ciency well over the GFLOP/W, with power consumption ranging from 20 to 440 mW.

In all cases, after enabling SODA-OPT we observe a trade-off between performance and area/power consumption, with power and area overheads that linearly increase with the obtained speedup. This is expected, as the SODA-OPT default optimization pipeline generates bigger designs by allocating more resources in parallel to reduce the execution time (especially through loop unrolling). With the selected benchmarks, we can see that simple operators, such as ReLU, achieve an ef ciency up to hundreds of GFLOPS/W, while more complex operators, such as convolutions, reach 10 GFLOPS/W. In fact, an increase in the amount of allocated computational resources increases power consumption: for this reason, smaller kernels (e.g., ReLU) have lower power overhead and higher ef ciency.

TABLE 3
Execution Delay, Area, Power, and Ef ciency of the DNN accelerators synthesized with high-level optimizations.

B (bottom), T (top) and C (centered) refer to the branches in Figure 7a.

ResNet50				
Kernels	Cycles	Area(um ^{2})	Power(W)	GFLOPS/W
B_00_conv2d	2,554,953,728	175,874	0.069	10.3
B_01_fbn	25,619,335	662,899	0.042	19.1
B_02_relu	3,353,684	70,949	0.032	141.3
B_03_conv2d	2,860,150,784	517,396	0.237	6.2
B_04_fbn	6,602,595	639,438	0.042	19.7
B_05_relu	870,770	48,977	0.021	185.6
B_06_conv2d	1,277,263,872	173,603	0.069	10.8
B_07_fbn	26,395,323	638,947	0.044	19.2
C_00_add	5,724,970	78,439	0.0378	17.8
C_01_relu	3,480,074	49,253	0.0217	183.0
T_00_conv2d	2,552,758,272	174,580	0.0627	11.6
T_01_fbn	26,395,323	638,929	0.042	19.5

MobileNetV2				
Kernels	Cycles	Area(umˆ2)	Power(W)	GFLOPS/W
00_conv2d	6,058,752	1,281,674	0.440	7.2
01_add	707,350	83,958	0.049	8.7
02_relu	648,214	42,050	0.023	82.2
03_dwconv2d	3,622,080	407,501	0.204	7.3
04_fbn	3,468,402	758,623	0.055	7.5
05_relu	648,214	42,050	0.023	82.2
06_conv2d	4,246,144	724,024	0.383	11.8
07_add	117,910	81,636	0.041	62.1

6.2 Qualitative comparison with other ML accelerators

Table 4 shows characteristics and quality metrics of popular architectures used for training and inference of DNNs, compared to one of the highly specialized accelerators generated by the SODA Synthesizer. To perform this comparison, we used SODA to generate dense matrix multiplication (MatMul) accelerators with 8 memory ports and different number formats. For programmable devices, peak rates may or may not be achieved depending on how optimized is the input code; instead, our approach implements fully specialized accelerators for speci c operations (e.g., matrix multiplication, matrix-vector multiplication, or ML operators and models). Therefore, the ef ciency results reported in Table 4 are derived from a theoretical peak throughput, except for the accelerator generated by SODA, where we calculated the actual throughput and ef ciency based on execution time. SODA generates FSMD accelerators where the number of functional units depends on the amount of exposed parallel arithmetic operations in the kernel (which is controlled by high-level optimizations), while other devices based on systolic arrays contain thousands of processing units that may or may not be fully utilized depending on the operation. Considering these differences, and the signi cantly older technology node, our ef ciency results at FP32 are comparable to the other FP32 accelerators, which are also considerably larger devices and might not meet edge requirements.

The designs in the bottom half of the table support ML-speci c oating point number formats (e.g., b oat16 or tensor oat32) or integer/ xed point formats. FPGA-based custom accelerators typically focus on integer/ xed point formats, as implementing oating point units on ne-grained recon gurable devices is inef cient. For example, DNNBuilder, targeting the Kintex UltraScale FPGA, leverages a xed-point 16-bit format. Custom number formats increase ef ciency with limited loss of accuracy; the SODA toolchain, thanks to its modularity, can easily be extended to parse quantized models and generate functional units with specialized number formats. The SODA-generated xed-point 16-bit MatMul reaches an ef ciency >150 GOPS/W.

6.3 Comparison between data ow architecture and centralized architecture

We used the blocks of DNN layers in Figure 7 to compare the performance of the two different ways in which we can connect synthesized accelerators: the centralized architecture, and the data ow architecture. The centralized architecture is a system like the one depicted in Figure 3d, where individual accelerators are attached to a central bus, a microcontroller drives their execution, and the data they exchange is stored in and retrieved from an external memory. The data ow architecture, instead, is a system that uses our distributed controller to orchestrate the execution of accelerators accessing a shared memory, similar to the one of Figure 3e. For all experiments, each individual operator in the DNN graph is outlined, processed by SODA-OPT with the full optimization pipeline, synthesized by Bambu, and simulated with Verilator.

While the simulation already accounts for shared memory accesses, we estimate the cost of communication between accelerators and external memory taking into consideration the type and size of the inputs and outputs for each kernel. We consider a memory bandwidth of 6400MB/s, typical of DDR3 RAM modules using 45 nm technology cells, and calculate transfer times as seen by the accelerator, in terms of clock cycles at 500MHz. In the centralized architecture, accelerators communicate with each other through the external memory. In the data ow architecture, only the graph inputs and outputs go through the external memory, while intermediate results are kept within a shared internal scratchpad memory. We assume this shared scratchpad memory to have as many ports as independent accelerators, so that, using the HMI memory interface described in Section 5, the architecture can support con ict-free concurrent accelerator execution, allowing for ef cient pipelined execution of streaming workloads. We assume a latency of 2 cycles for read and 1 cycle for write operations. This assumption is reasonable, as there exist high-performance scratchpad designs with up to 16 independent banks, enough to support the benchmarks in our experiments.

To model the overall latency of the centralized architecture, we simply add the execution time of each accelerator with the time it takes to transfer data to/from external memory before and after its execution. We compute the streaming latency by multiplying the result by the number of inputs in the stream. In fact, although the synthesized accelerators can execute in parallel on different inputs, the application host code derived from the original MLIR representation of the DNN model only invokes them sequentially.

The model to estimate the performance of the data ow architecture, instead, takes into account the support for

TABLE 4
Comparison between DNN accelerators, extended from [43]. Giga Operation per Seconds per Watt (GOPS/W) was calculated on the respective GOPS, Power, and Clock values.

Floating-point accelerators						
Platform	Technology	Precision	Power [W]	Clock [MHz]	GFLOPS/W	Notes
V100 GPU	12 nm	FP32	300	1246	52:33	Theoretical peak
A100 GPU	7 nm	FP32	400	1410	48:75	Theoretical peak
TPU v3	16 nm	FP32	450	940	8:89	Theoretical peak
SODA MatMul	45 nm	FP32	0:42	500	17:46	Derived from execution time
Accelerators with different numerical formats						
Platform	Technology	Precision	Power [W]	Clock [MHz]	GOPS/W	Notes
A100 GPU Tensor Core	7 nm	TF32	400	1410	780	Theoretical peak
TPU v3	12 nm	FP16	450	940	273:33	Theoretical peak
TPU v4	7 nm	BF16	175	N/A	1432:29	Theoretical peak
SIGMA Sparse	28 nm	FP16	22:3	500	480	Average across GEMMs
(KU115) DNNBuilder	20 nm	Fixed16	22:9	235	90:2	Batch execution of VGG
SODA MatMul	45 nm	Fixed16	0:05	500	162:25	Derived from execution time

TABLE 5
Number of cycles to execute the DNN accelerators using the centralized (baseline) or the data ow architecture.

ResNet50						
	Single Input			Streaming (100 inputs)		
Arch.	Runtime	Memory	Total	Runtime	Memory	Total
Centralized	1,146,101,992	7,152,635	1,153,254,627	114,610,199,200	715,263,486	115,325,462,686
Data ow	806,742,427	656,320	807,398,747	34,677,385,627	656,320	34,678,041,947
Speedup	1.4	10.9	1.4	3.3	1089.8	3.3
MobileNetV2						
	Single Input			Streaming (100 inputs)		
Arch	Runtime	Memory	Total	Runtime	Memory	Total
Centralized	19,517,066	3,726,301	23,243,367	1,951,706,600	372,630,149	2,324,336,749
Data ow	19,517,066	64,345	19,581,411	625,392,266	64,345	625,456,611
Speedup	1.0	57.9	1.2	3.1	5,791.2	3.7

concurrent and pipelined execution provided by the distributed controller. We rst identify the longest path in a directed acyclic graph where vertices correspond to kernel or memory latencies and edges replicate the edges in the application data ow graph; the sum of latencies along the critical path corresponds to the overall execution time for a single input. In this way, we account for fork-join patterns in the application data ow graph, where multiple branches can be executed in parallel and the overall latency is determined by the slowest branch latency. In streaming execution, the data ow architecture latency becomes the latency of a single input execution plus N - 1 times the initiation interval, where N is the number of elements in the input stream and the initiation interval is the latency of the slowest kernel or memory transfer.

Table 5 provides the execution latency in clock cycles for the two blocks of layers in Figure 7, and uses the results from the centralized architecture as a baseline to assess the performance improvement provided by the data ow architecture. For the Resnet50 block, using our data ow architecture, accelerators implementing layers in the upper branch of Figure 7a can execute in parallel with accelerators implementing layers in the lower branch. Compared to the centralized architecture baseline, this results in a speedup of 1.4x during single input execution, and a speedup of 3.5x when streaming a batch of 100 inputs. For MobileNetV2, although Figure 7b does not have parallel branches, we still observe signi cant savings due to the reduced accesses to external memory; in fact, the centralized system spends 57.9x and 5,791.2x more cycles to transfer data between accelerators and external memory, with a single input and when streaming a batch of inputs, respectively.

6.4 Discussion

Our experiments show how the SODA Synthesizer high-level optimizations and data ow methodology provide signi cant performance gains, with reasonable area and power overheads, while requiring minimal user interaction. Outlining each layer for acceleration, as we did in the experiments, can lead to imbalanced execution times and utilization (e.g., a ReLU node remaining idle for most of the time waiting until the convolution node has nished). In the future, the outlining strategy can be improved to better exploit optimization at the level of the computational graph. Oper-

ators (e.g., convolution, bias, ReLU) can be fused together or further partitioned into smaller primitives, aiming to generate accelerators with similar computational intensity and a higher utilization of resources. For kernels that are memory-bound, the HMI design can be further extended to support FSMD accelerators with multiple ports and better manage buffers between nodes, to increase memory parallelism in the data ow architecture. In summary, our data ow generation methodology enables the synthesis of input DNN models in a system of specialized accelerators, generates ef cient accelerators through high-level optimizations, and does not require any manual code modi cation to identify the accelerators in the input speci cation.

7 Conclusion

This paper presents the SODA Synthesizer, an open-source, multi-level, no-human-in-the-loop hardware compiler able to transform speci cations from high-level software frameworks (Machine Learning in particular) into ef cient FPGA/ASIC accelerators. Its frontend, SODA-OPT, leverages the MLIR framework to identify kernels for acceleration, to generate orchestrating code, and to implement a set of high-level optimizations that restructure the kernels to enhance the hardware generation backend, i.e., state-of-the-art HLS tool Bambu. SODA also implements a methodology to assemble highly optimized FSMD accelerators in a coarse-grained, dynamically scheduled data ow design, which provides better performance compared to a centralized architecture with a microcontroller driving the execution of accelerators, especially in the case of streaming inputs. We show that our-high level optimization pipeline effectively yields better HLS results (up to 74x speedup compared to an unoptimized baseline), and that the data ow architecture can provide a further 3x speedup thanks to reduced accesses to external memory, concurrent execution, and pipelining.

Future works involve further extending the methodology for the generation of the data ow system of accelerators, both at the frontend level and at the backend. At the front-end, SODA can exploit semantic information of the computational graph to better balance the custom accelerators generated for each operator or layer. At the backend, there are opportunities to further improve interfacing between accelerators and memory.

Our compiler-based toolchain is modular by construction, and it will easily allow further development to introduce new optimization techniques, automate design space exploration, and explore different architectural models.

Acknowledgments

This research was partially supported by the Software De ned Accelerators for Data Analytics (SO(DA) ^{2}) project in the Data Model Convergence Initiative under the PNNL's Laboratory Directed Research and Development (LDRD) program and the HERMES project, which received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N° 101004203.

References

[1] T. Tuttle, “Keynote: Accelerating the IoT,” 2017, DAC 2017: 54th ACM/IEEE Design Automation Conference.

[2] P. Beckman, C. Catlett, M. Ahmed, M. Alawad, L. Bai, P. Balaprakash, K. Barker, R. Berry, A. Bhuyan, G. Brebner, K. Burkes, A. Butko, F. Cappello, R. Chard, S. Collis, J. Cree, D. Dasgupta, A. Evdokimov, J. M. Fields, P. Fuhr, C. Harper, Y. Jin, R. Kettimuthu, M. Kiran, R. Kozma, P. A. Kumar, Y. Kumar, L. Luo, L. Mashayekhy, I. Monga, B. Nickless, T. Pappas, E. Peterson, T. Pfeffer, S. Rakheja, V. R. Tribaldos, S. Rooke, S. Roy, T. Saadawi, A. Sandy, R. Sankaran, N. Schwarz, S. Somnath, M. Stan, C. Stuart, R. Sullivan, A. Sumant, G. Tchilinguirian, N. Tran, A. Veeramany, A. Wang, B. Wang, A. Wiedlea, S. Wielandt, T. Windus, Y. Wu, X. Yang, Z. Yao, R. Yu, Y. Zeng, and Y. Zhang, “5G Enabled Energy Innovation: Advanced Wireless Networks for Science (Workshop Report),” 3 2020. [Online]. Available: https://www.osti.gov/biblio/1606538

[3] E. W. Bethel and M. G. eds, “Report of the DOE Workshop on Management, Analysis, and Visualization of Experimental and Observational data – The Convergence of Data and Computing,” May 2016. [Online]. Available: https://www.osti.gov/biblio/1525145

[4] A. Tumeo, M. Minutoli, V. G. Castellana, J. Manzano, V. Amatya, D. Brooks, and G.-Y. Wei, “Invited: Software de ned accelerators from learning tools environment,” in DAC 2020: 57th ACM/IEEE Design Automation Conference, 2020, pp. 1–6.

[5] M. Minutoli, V. G. Castellana, C. Tan, J. Manzano, V. Amatya, A. Tumeo, D. Brooks, and G.-Y. Wei, “Soda: a new synthesis infrastructure for agile hardware design of machine learning accelerators,” in ICCAD 2020: IEEE/ACM International Conference On Computer Aided Design, 2020, pp. 1–7.

[6] J. J. Zhang, N. Bohm Agostini, S. Song, C. Tan, A. Limaye, V. Amatya, J. Manzano, M. Minutoli, V. G. Castellana, A. Tumeo, G.-Y. Wei, and D. Brooks, “Towards automatic and agile ai/ml accelerator design with end-to-end synthesis,” in ASAP 2021: IEEE 32nd International Conference on Application-speci c Systems, Architectures and Processors, 2021, pp. 218–225.

[7] V. G. Castellana and F. Ferrandi, “Abstract: Speeding-up memory intensive applications through adaptive hardware accelerators,” in Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, ser. SCC '12, 2012, p. 1415–1416.

[8] V. G. Castellana, A. Tumeo, and F. Ferrandi, “High-level synthesis of parallel speci cations coupling static and dynamic controllers,” in 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2021, pp. 192–202.

[9] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, “In-datacenter performance analysis of a tensor processing unit,” in Proceedings of the 44th Annual International Symposium on Computer Architecture, ser. ISCA '17, 2017, p. 1–12.

[10] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, “A Survey of FPGA-Based Neural Network Inference Accelerators,” ACM Trans. Recon gurable Technol. Syst., vol. 12, no. 1, mar 2019. [Online]. Available: https://doi.org/10.1145/3289185

[11] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi et al., “A con gurable cloud-scale dnn processor for real-time ai,” in 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA) IEEE, 2018, pp. 1–14.

[12] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, “Ef cient processing of deep neural networks: A tutorial and survey,” Proceedings of the IEEE, vol. 105, no. 12, pp. 2295–2329, 2017.

[13] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, “Pudiannao: A polyvalent machine learning accelerator,” in Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages

and Operating Systems, ser. ASPLOS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 369–381. [Online]. Available: https://doi.org/10.1145/2694344.2694358

[14] M. Khan, D. Lester, L. Plana, A. Rast, X. Jin, E. Painkras, and S. Furber, "Spinnaker: Mapping neural networks onto a massively-parallel chip multiprocessor," in 2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence), 2008, pp. 2849–2856.

[15] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh, "Tabla: A unified template-based framework for accelerating statistical machine learning," in 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2016, pp. 14–26.

[16] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," in IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers, 2016, pp. 262–263.

[17] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W. Hwu, and D. Chen, "DNNBuilder: an Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs," in ICCAD, 2018, pp. 1–8.

[18] X. Zhang, H. Ye, J. Wang, Y. Lin, J. Xiong, W. Hwu, and D. Chen, "DNNExplorer: A Framework for Modeling and Exploring a Novel Paradigm of FPGA-based DNN Accelerator," in ICCAD, 2020, pp. 1–9.

[19] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA) 2020, pp. 58–70.

[20] H. Kwon, A. Samajdar, and T. Krishna, MAERI: Enabling Flexible Data-flow Mapping over DNN Accelerators via Reconfigurable Interconnects, 2018, p. 461–475.

[21] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel patterns," in 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), 2017, pp. 389–402.

[22] 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), 2021, pp. 1–7.

[23] S. Kinzer, J. K. Kim, S. Ghodrati, B. Yatham, A. Althoff, D. Mahajan, S. Lerner, and H. Esmaeilzadeh, "A computational stack for cross-domain acceleration," in 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA) IEEE, 2021, pp. 54–70.

[24] 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.

[25] 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, vol. 39, no. 5, pp. 8–16, 2019.

[26] 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, vol. 70, no. 12, pp. 2015–2028, 2021.

[27] 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, 2019, pp. 242–251.

[28] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, and Z. Wu, "Fast inference of deep neural networks in FPGAs for particle physics," Journal of Instrumentation, vol. 13, no. 07, pp. P07 027–P07 027, jul 2018. [Online]. Available: https://doi.org/10.1088/1748-0221/13/07/p07027

[29] 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, pp. 1–13, 2021.

[30] A. Girault, B. Lee, and E. Lee, "Hierarchical Finite State Machines with Multiple Concurrency Models," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 18, no. 6, pp. 742–760, Jun 1999.

[31] R. S. Nikhil, "Bluespec: A general-purpose approach to high-level synthesis based on parallel atomic transactions," in High-Level Synthesis: From Algorithm to Digital Circuit, P. Coussy and A. Morawiec, Eds. Springer Netherlands, 2008, pp. 129–146.

[32] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically scheduled high-level synthesis," in FPGA '18: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2018, p. 127–136.

[33] J. Cheng, L. Josipović, G. A. Constantinides, P. Ienne, and J. Wickerson, "Combining Dynamic & Static Scheduling in High-Level Synthesis," in FPGA '20: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2020, p. 288–298.

[34] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, and K. Olukotun, "Spatial: A language and compiler for application accelerators," in PLDI 2018: the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2018, p. 296–311.

[35] Xilinx, "Exploiting task level parallelism: Data flow optimization," in Vitis HLS User Guide, 2021, pp. 283–296.

[36] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling compiler infrastructure for domain specific computation," in 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2021, pp. 2–14.

[37] 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, 2021, pp. 1327–1330.

[38] 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, 2015, pp. 100–100.

[39] 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, no. 01, pp. 1–14, feb 2021.

[40] W. Snyder, "Verilator," 2003, online accessed on 22-11-2020. [Online]. Available: https://www.veripool.org/wiki/verilator

[41] C.-J. Tseng and D. P. Siewiorek, "Automated synthesis of data paths in digital systems," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 5, no. 3, pp. 379–395, 1986.

[42] V. G. Castellana, A. Tumeo, and F. Ferrandi, "An Adaptive Memory Interface Controller for Improving Bandwidth Utilization of Hybrid and Reconfigurable Systems," in DATE 2014: Design, Automation and Test in Europe, 2014, pp. 1–4.

[43] N. B. Agostini, S. Dong, E. Karimi, M. T. Lapuerta, J. Cano, J. L. Abellán, and D. Kaeli, "Design space exploration of accelerators and end-to-end dnn evaluation with tightly-coupled soc," in 2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD) IEEE, 2020, pp. 10–19.

Serena Curzel received the B.S. degree in Electronics and Telecommunication Engineering from Universita degli studi di Trento, Italy, in 2016 and the M.S. degree in Electronics Engineering from Politecnico di Milano, Italy, in 2019, where she is currently pursuing the Ph.D. degree in Information Technology. Her main research interests are acceleration of domain-specific applications (including deep neural networks) and High-Level Synthesis. Since 2021 she is a PhD intern at Pacific Northwest National Laboratory, where she is collaborating with the HPC group to develop new HLS techniques and compiler optimizations for machine learning hardware.