ICCAD Special Session Paper

Automated Generation of Integrated Digital and Spiking Neuromorphic Machine Learning Accelerators

Serena Curzel[§‡], Nicolas Bohm Agostini[†‡], Shihao Song[‡‖], Ismet Dagli[‡¶], Ankur Limaye[‡], Cheng Tan[‡], Marco Minutoli[‡], Vito Giovanni Castellana[‡], Vinay Amatya[‡], Joseph Manzano[‡], Anup Das[‖], Fabrizio Ferrandi[§], Antonino Tumeo[‡]
[‡]*Pacific Northwest National Laboratory*, Richland, WA, USA
[†]*Northeastern University*, Boston, MA, USA
[§]*Politecnico di Milano*, Milan, Italy
[¶]*Colorado School of Mines*, Golden, CO, USA
[‖]*Drexel University*, Philadelphia, PA, USA

***Abstract*—The growing numbers of application areas for artificial intelligence (AI) methods have led to an explosion in availability of domain-specific accelerators, which struggle to support every new machine learning (ML) algorithm advancement, clearly highlighting the need for a tool to quickly and automatically transition from algorithm definition to hardware implementation and explore the design space along a variety of SWaP (size, weight and Power) metrics. The software defined architectures (SODA) synthesizer implements a modular compiler-based infrastructure for the end-to-end generation of machine learning accelerators, from high-level frameworks to hardware description language. Neuromorphic computing, mimicking how the brain operates, promises to perform artificial intelligence tasks at efficiencies orders-of-magnitude higher than the current conventional tensor-processing based accelerators, as demonstrated by a variety of specialized designs leveraging Spiking Neural Networks (SNNs). Nevertheless, the mapping of an artificial neural network (ANN) to solutions supporting SNNs is still a non-trivial and very device-specific task, and completely lacks the possibility to design hybrid systems that integrate conventional and spiking neural models. In this paper, we discuss the design of such an integrated generator, leveraging the SODA Synthesizer framework and its modular structure. In particular, we present a new MLIR dialect in the SODA frontend that allows expressing spiking neural network concepts (e.g., spiking sequences, transformation, and manipulation) and we discuss how to enable the mapping of spiking neurons to the related specialized hardware (which could be generated through middle-end and backend layers of the SODA Synthesizer). We then discuss the opportunities for further integration offered by the hardware compilation infrastructure, providing a path towards the generation of complex hybrid artificial intelligence systems.**

***Index Terms*—MLIR, Artificial Neural Network Accelerators, Spiking Neural Network Accelerators**

I. Introduction

The exponential growth of interest in data analytics, machine learning, and artificial intelligence methods led to a "cambrian explosion" of domain-specific architectures [1] to accelerate new algorithms and methods as they get developed. Data scientists typically employ high-level frameworks (in Python or similar functional languages), such as TensorFlow, PyTorch, CNTK, to study, develop, and implement algorithms. On the other hand, hardware designers typically need to work at a different abstraction level by identifying key computational kernels to build specialized functional units, datapaths, memory components, and overall system design. Unfortunately, the rapid evolution of the methods and the long and complicated hardware development efforts generate fundamental productivity and time-to-market gaps. High-level compilation frameworks (some tool specific, like XLA [2] for TensorFlow, or Glow [3] for Pytorch, others more general, like TVM [4]) try to address at least the aspects of mapping "tensor-based" operations to specialized accelerators (including general purpose graphic processing units with tensor cores, tensor processing units, systolic arrays, coarse grained reconfigurable arrays, and data flow architectures). While such frameworks constitute a significant step ahead, underlying architectures typically still remain optimized for only a subset of primitives, the ones that hardware designers were able to identify and optimize upon, leaving many other opportunities for optimizations largely unexplored.

Extending High-Level Synthesis (HLS) methodologies to (semi)-automatically generate specialized accelerators in hardware description languages starting from specifications in high-level languages appears a very promising strategy. Moving in this direction can bridge the productivity gap while allowing, at the same time, a complete exploration of hardware design points along various and often contrasting metrics (not only performance, power, and area, but also real-time requirements and ability to fit in specific autonomous systems). Leveraging compiler-based frameworks, HLS can naturally implement domain-specific optimizations as compiler passes, recognize relevant code patterns, and generate efficient hardware implementations.

A number of experimental approaches in this direction [5, 6, 7] convert high-level operators from the Python-based frameworks into HLS code "templates" written in C/C++, which are then synthesized with commercial tools (Vivado HLS, Catapult C, etc), typically targeting field programmable gate arrays (FPGAs). The software defined accelerators (SODA) Synthesizer [8, 9], instead, adopts a multi-layered, modular, fully open-source, compiler-based approach with a high-level frontend and optimizer based on the multi-level intermediate representation (MLIR) framework [10] to perform hardware/software decomposition and domain-specific transformation, and a synthesizer backend to generate custom Verilog modules that could target different device technologies (FPGAs from various vendors, as well as commercial and open-source application-specific integrated circuit - ASIC). In SODA, translation across different levels of abstraction is always performed with progressive lowerings between intermediate representations (IRs). This approach allows for an integrated flow of information that helps solve design exploration challenges that arise due to semantic mismatches: in fact, directly converting Python operators to an imperative language like C inevitably leads to losing analysis and optimizations opportunities. Pre-optimized C code templates are a suboptimal solution to this problem.

Research on new domain-specific accelerators for artificial intelligence is also leading to the exploration of neuromorphic computing, i.e., models of computation that mimic how biological brains operate and promise to perform machine learning tasks with orders of magnitude higher efficiency than conventional "tensor-based" digital approaches. Spiking Neural Networks (SNN), in particular, incorporate

the concept of time in their operating model, transmitting information (firing) when a "membrane potential" reaches a specific value. FPGAs have been used in many cases to evaluate and implement SNNs; a number of specialized devices that can run SNN models have also been designed, both exploring the digital electronics domain (e.g., IBM's TrueNorth [11], Intel's Lohihi [12]) and the analog electronics domain (e.g., Georgia Tech's Field Programmable Analog Array [13]). FPGA implementations have also started to leverage HLS techniques to quickly generate digital SNN accelerators, after performing the necessary high-level transformations [14]. There is a significant trend towards providing more structured solutions to perform mapping and design space exploration for SNN accelerators. However, their hardware and software interfaces, their integration in complex systems with "conventional" accelerators, and the possibility of exploring and implementing hybrid analog/digital systems remain largely unexplored.

This paper discusses how the SODA's end-to-end, multi-layered, compiler-based framework can be adapted to support SNN models. We identify the roadmap for integration, first introducing our current approach to extend SODA's MLIR-based frontend to support SNN mapping leveraging the NeuroXplorer toolchain [15]. Then we sketch subsequent steps that could enable the generation of a complete hybrid digital and spiking neural network system, potentially allowing such systems to be used for both inference (on the spiking part) and training (on the conventional "digital" part). We finally provide a perspective on how interoperable hardware compilers could further allow the integration of hybrid digital and analog components in complex heterogeneous systems, exploring new ideas in the area of HW/SW codesign for artificial intelligence.

II. Background

This section provides the background information needed to discuss our approach for designing a generation infrastructure for integrated conventional and spiking neural network accelerators. We first introduce the SODA Synthesizer and its MLIR-based frontend, providing information on why and how the MLIR framework is employed. We then introduce SNNs and neuromorphic accelerators, focusing particularly on how the NeuroXplorer framework can efficiently map SNNs to neuromorphic hardware.

A. SODA Synthesizer

The SODA synthesizer answers to the request for fast and reliable design automation for domain-specific hardware accelerators, with a particular focus on the acceleration of machine learning applications. Machine learning models, algorithms, and approaches are evolving very quickly, with new application-specific methodologies rapidly emerging in response to new problems. With the end of Dennard's scaling, general purpose architectures are experiencing diminishing improvements, thus domain-specific accelerators are the only possible approach to keep increasing efficiency (performance per watt) with current silicon manufacturing technologies. However, designing hardware by hand is complex and time-consuming and hardly keeps up with quick algorithmic evolution. Additionally, different target applications have different performance, area and power consumption requirements, and a hardware designer may want to explore a variety of trade-offs among such metrics. A quick transition from an algorithm formulation to a correspondent accelerator implementation is thus highly desirable to explore possible designs with minimal human interaction.

With SODA, we address the abstraction gap between high-level algorithmic design and low-level hardware implementation by introducing a modular, multi-level, and extensible open-source compiler-based framework. SODA is composed of a compiler-based frontend, a compiler-based middle-end, and a compiler-based backend, which work together to generate synthesizable Verilog for a variety of targets.

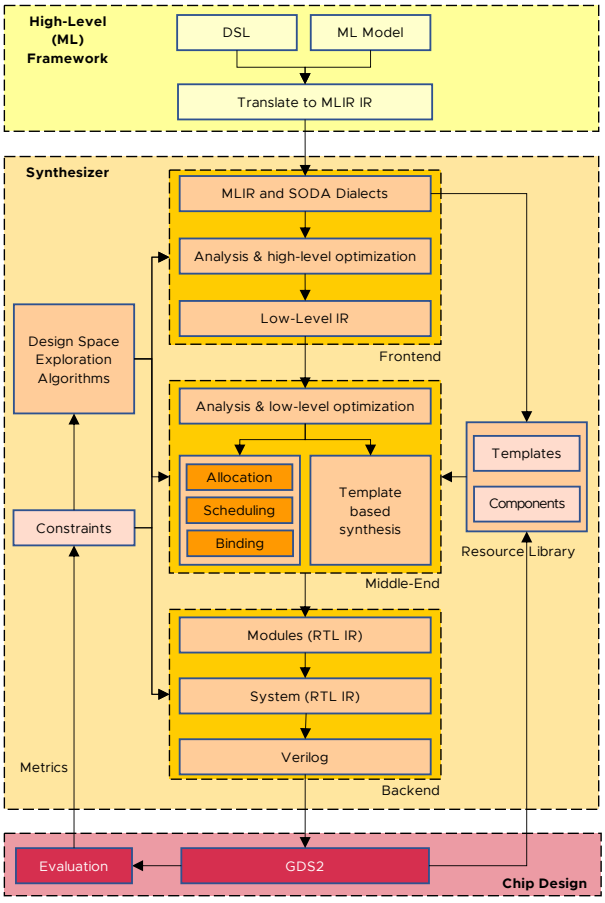


Fig. 1. SODA - A framework for generation of custom accelerators

SODA leverages a multi-level compiler infrastructure to perform optimizations all along the compilation flow, selecting the correct level of abstraction and promoting separation of concerns to increase modularity. The compilation process includes high-level transformations and optimizations, a hardware synthesis middle-end which supports various conventional and *templated* high-level synthesis methodologies, and a backend to generate hardware description language files (Verilog, in our case). Figure 1 showcases the various steps involved in the generation of GDSII layouts for ASICs, which is one of the possible synthesis flows enabled by SODA.

The multi-layered, compiler-based approach of the SODA infrastructure is further supported by the adoption of the Multi-Level Intermediate Representation (MLIR) [10] in the SODA-OPT frontend. MLIR provides a framework to implement reusable compiler infrastructures, leveraging a meta-IR, inspired by the LLVM IR, that allows defining specialized IRs (*dialects*) at different levels of abstractions, which tackle specific types of transformations and optimizations. Such an approach is particularly suited to support domain-specific compilation pipelines for specialized architectures and has been proven valuable, among the other things, within the TensorFlow 2.0 runtime. MLIR has been integrated in the overall LLVM infrastructure and provides several dialects of general interest (e.g., `linalg`, `scf`) together with many others pertaining to specific high-level frameworks, optimizations,

or architectures. By supporting MLIR and its dialects we make our framework available to any application designed with tools that provide a translation into MLIR: this includes TensorFlow and ONNX for machine learning and deep learning algorithms, but can be of interest to many other domain-specific frameworks or languages.

SODA-OPT starts by parsing input models produced by machine learning high-level frameworks: it identifies dataflow segments that are amenable to hardware acceleration, outlines the relevant regions, and performs high-level optimizations. Regions of code which are not selected for acceleration are equipped with interfacing code and runtime calls that will connect the host microcontroller to the generated accelerator. Eventually, the frontend feeds a pre-optimized lower-level Intermediate Representation to the SODA middle-end, which is ready to perform the first target-specific transformations. SODA-OPT exploits its own `soda` dialect, which provides the necessary abstraction to search and outline code patterns that need to be lowered to other dialects, and finally implemented on hardware. Other custom dialects can be implemented in the frontend for specific purposes, as will be the case for the `snn` dialect described in Section III-B.

The SODA middle-end and backend components perform target-specific optimizations on the code outlined by the frontend. Frontend and middle-ends communicate by exchanging compiler IRs, in particular LLVM IR, which is a natural target of MLIR lowering pipelines. For example, one of the SODA synthesizer middle-ends is the open-source HLS tool Bambu [16], which supports the generation of classic finite state machine with datapath (FSMD) architectures but also offers advanced parallel multithreaded accelerator templates with complex memory subsystems [17]. Bambu supports FPGA devices from a variety of vendors, with specialized functional units implemented for each specific target, and we also recently extended it to interface with the OpenROAD flow providing a fully open-source ASIC backend.

Another experimental SODA backend leverages an LLVM-based synthesizer able to generate hardware descriptions in a circuit-level IR, currently FIRRTL [18]. We argue that further decoupling the hardware generation process from the actual code generation provides the opportunity to perform new types of optimizations at the system level (i.e., between hardware modules), and better specialize the generated HDL code for the target device without need for hand tuning. Similar principles and concepts are currently being explored within the CIRCT (Circuit IR Compilers and Tools [19]) LLVM incubator project, which is implemented within the MLIR framework and thus represents a possible target for future integration with the SODA Synthesizer.

B. Spiking Neural Networks and Neuromorphic Accelerators

Spiking Neural Networks (SNNs) enable powerful computations due to their spatio-temporal information encoding capabilities [20]. In an SNN, spikes (i.e., current) injected from pre-synaptic neurons raise the membrane voltage of a post-synaptic neuron (see the middle sub-figure of Figure 2). When the membrane voltage crosses a threshold (V_{th} in the figure), the post-synaptic neuron emits spikes that propagate to other neurons (see the right sub-figure of Figure 2). SNNs implement some variants of Integrate and Fire (I&F) neurons with a spike duration ranging from 1 μs to several ms [21] (see the left sub-figure of Figure 2).

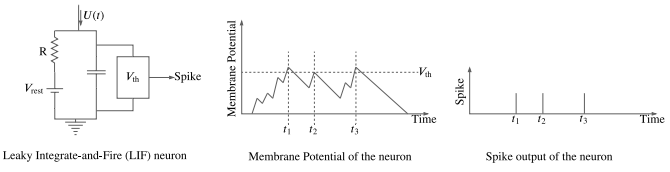
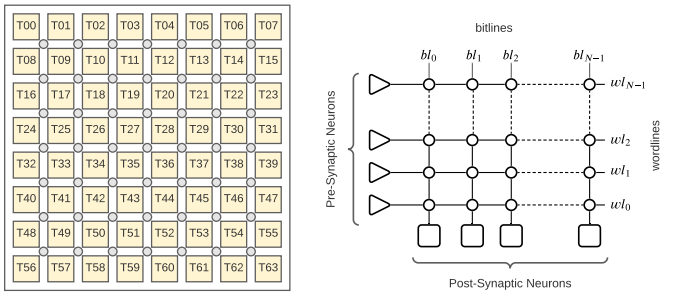


Fig. 2. A leaky integrate-and-fire (LIF) neuron with current input $U(t)$ (left). The membrane potential over time of the neuron (middle). The spike output of the neuron representing its firing time (right).

SNNs can implement different machine learning approaches. Examples include deep learning [22], liquid state machine [23], and reinforcement learning [24]. We focus on inference, referring to feeding live data points to a trained SNN in order to generate the corresponding output.

SNNs are executed on many-core neuromorphic accelerators such as TrueNorth [11], Loihi [12], and SpiNNaker [25]. Figure 3(a) shows the tile-based architecture of such an accelerator, where each tile consists of 1) a neuromorphic processing core that integrates neuron circuitry and synaptic storage and 2) a network interface to communicate spike packets over a shared interconnect such as Network-on-Chip (NoC) [26] and Segmented Bus [27]. A common practice is to design the processing core as a crossbar (see Figure 3(b)), where the synaptic cells are organized in a two-dimensional grid, with neuron circuitry placed along bitlines and wordlines [28].



(a) A many-core neuromorphic accelerator.

(b) Analog crossbar-based neuromorphic processing core.

Fig. 3. Tile-based neuromorphic accelerator and crossbar architecture.

To map SNNs to a many-core neuromorphic accelerator, a system software such as NEUTRAMS [29], SpiNeMap [30], Corelet [31], and NeuroXplorer [15] is used. These software frameworks consist of 1) a compiler to partition an SNN into clusters, such that each cluster can fit onto a core of the accelerator, and 2) a run-time manager to allocate clusters to cores, improving energy [32], reliability [33], endurance [34], or inference lifetime [35].

To enable efficient compilation of SNNs, dataflow-based techniques are commonly used [36]. An SNN is represented as a dataflow graph, where nodes represent neurons and edges represent synaptic connections (See Figure 4a). Formally,

Definition 1: (SNN GRAPH) *An SNN* $\mathbf{G_{SNN}} = (\mathbf{N}, \mathbf{S})$ *is a directed graph consisting of a finite set* $\mathbf{N}$ *of nodes, representing neurons and a finite set* $\mathbf{S}$ *of edges, representing synapses.*

A dataflow compiler partitions an SNN into clusters, where each cluster consists of a subset of neurons and synapses of the SNN. Partitioning algorithm incorporates the resource constraints of a core, allowing the clusters to be mapped to cores of the many-core accelerator. In many recent works [30, 36, 37], partitioning is performed using a variant of the Kernighan–Lin graph partitioning heuristic with the objective of minimizing the spike communication between the clusters (see for instance Figure 4b). This leads to reduced latency of the shared interconnect where the inter-cluster communication links are mapped. Formally,

Definition 2: (CLUSTERED SNN GRAPH) *A clustered SNN graph* $\mathbf{G_{CSNN}} = (\mathbf{C}, \mathbf{E})$ *is a directed graph consisting of a finite set* $\mathbf{C}$ *of clusters and a finite set* $\mathbf{E}$ *of edges between these clusters.*

A clustered SNN graph can be represented as a Synchronous Dataflow Graph (SDFG), where each cluster is represented as an *actor* and the inter-cluster communication channels are represented as *edges*. Each actor is associated with a set of input and output ports to which the incoming and outgoing edges are connected, respectively. Actor communicates by exchanging *tokens* on edges, which represent spikes. A spike is encoded as an address event representation (AER) data packet with a payload containing the address of clusters, where the destination neurons are mapped. In representing a clustered SNN graph as an SDFG, each communication channel between a pair of neurons that are mapped to two different clusters is represented as an edge between the clusters.

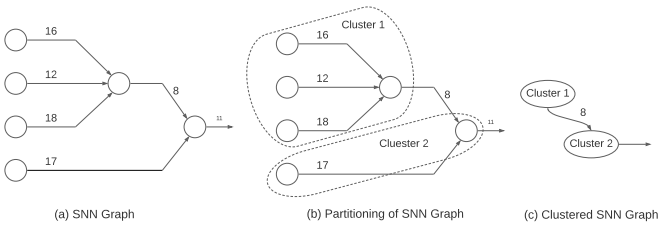


Fig. 4. (a) Representing an SNN graph. (b) Partitioning the SNN graph into two clusters. (c) Representing a clustered SNN graph.

Therefore, multiple incoming/outgoing edges may exist between any two pair of clusters. Additionally, each actor has port association based on its neurons and their connectivity. An actor is called *ready*, when it has sufficient number of tokens on the input ports of any of its neurons. Once a neuron in an actor fires, it generates tokens on all the output ports associated with the neuron. Therefore, in the SDFG representation of a clustered SNN graph, tokens are generated on a subset of the output ports, rather than on all output ports as in the original SDFG formalism.

Figure 5 illustrates the NeuroXplorer framework [15], which estimates the performance of an SNN using dataflow analysis techniques. The framework can work with both Artificial Neural Networks (ANNs) and biology-inspired Spiking Neural Networks (SNNs). NeuroXplorer interfaces with ANN workloads that are specified in high-level frameworks such as Tensorflow and PyTorch. To analyze an ANN workload for an event-driven neuromorphic hardware, the workload is first converted to an SNN using the SNN Conversion unit, and later the SNN is simulated using the SNN Simulation unit.

Alternatively, an SNN workload can be specified in PyNN [38] or PyCARL [39], which are Python interfaces to SNN simulators such as CARLsim [40], Brian [41], NEST [42], and Neuron [43]. These simulators model neural functions at various levels of detail and therefore have different requirements for computational resources. An SNN model can also be specified directly using these simulators.

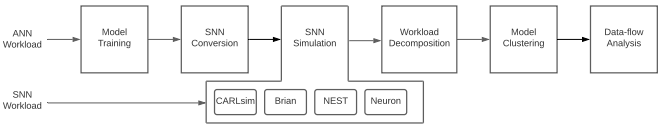


Fig. 5. A design flow to estimate performance of an SNN using dataflow analysis techniques.

The next step in the NeuroXplorer framework is the workload decomposition unit, where the simulated SNN workload is decomposed into fan-in-of-two (FIT) neural units to allow mapping them onto the processing cores of a neuromorphic hardware [44]. The decomposed SNN workload is then clustered using the Model Clustering unit, which uses a variant of the Kernighan–Lin graph partitioning heuristic to minimize inter-cluster spike communication [45]. Finally, the Dataflow Analysis unit converts the clustered SNN graph into SDFG representation and uses Max Plus Algebra to analyze performance, e.g., throughput for a given mapping of clusters to cores of the hardware [46]. NeuroXplorer can also be used to perform mapping explorations beyond load balancing.

Figure 6 illustrates the mapping of actors generated from the LeNet SNN model (circles) to the tiles of a neuromorphic hardware (rectangles) using the NeuroXplorer framework.

III. SNN INTEGRATION IN SODA

This section discusses how the SODA Synthesizer can be extended and adapted to generate accelerators for SNNs, learning and integrating features from the NeuroXplorer framework described in Section II-B. We aim at exploiting MLIR dialects and optimizations to translate a pre-trained ANN into an SNN, optimize it, and deploy it on specialized hardware. Section III-A describes the transformations involved in the SODA SNN deployment flow, while Section III-B dives into the details of the proposed SNN dialect.

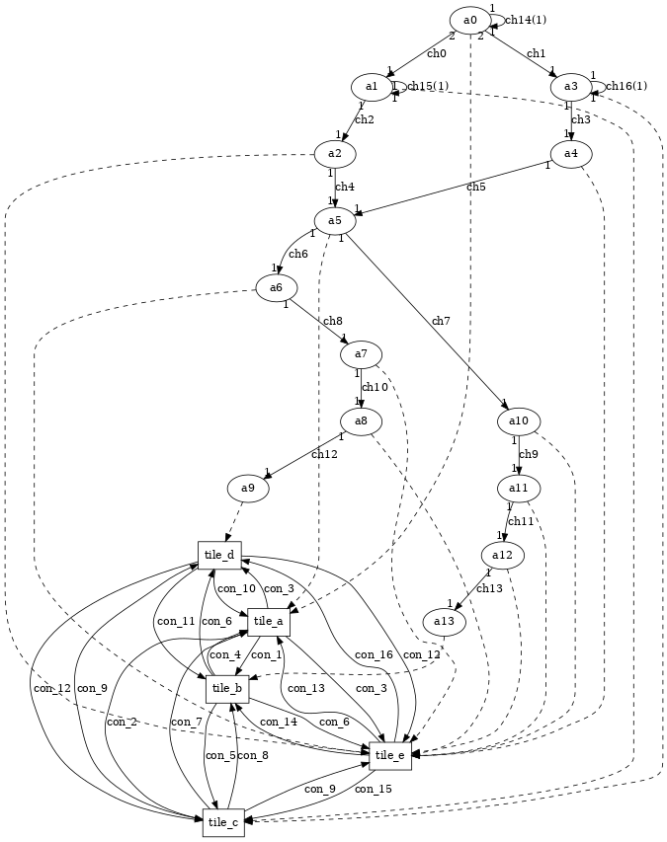


Fig. 6. LeNet platform mapping.

A. SNN deployment flow

Figure 7 divides the steps needed to generate an SNN accelerator in SODA into three groups, which correspond to three different development phases. In the first phase, the SODA frontend is modified to allow pre-trained models to SNN conversion and SNN emulation. The second and third phase produce two alternative deployment flows: one exploits the NeuroXplorer tools to program an existing neuromorphic chip, while the other synthesizes a custom hardware accelerator.

As in the original SODA flow, we expect the input to be a neural network model trained in a high-level software framework and translated to MLIR IR containing, for example, operations in the `tf` or `tosa` dialects. All the lowering and optimization passes that are already present in the SODA frontend can be applied, if beneficial, before the conversion to the `snn` dialect. The SNN conversion is the most relevant aspect of this first development phase, as it can impact the accuracy of the network: for this reason, we develop a specific dialect to represent neurons, spikes, and other SNN characteristics. This dedicated dialect also exposes further opportunities for optimizations that are specific to SNN models, and would not be accessible to the frontend with the direct conversion of the NeuroXplorer design flow. Operations and types of the `snn` dialect are then lowered and translated to LLVM IR for software execution, in order to emulate the behavior of the network on a test dataset, verify whether the initial accuracy was maintained within an acceptable error range, and collect data about spike times for all neurons in the network.

The second development phase aims at integrating the NeuroXplorer tools in the design flow, providing a path to map the input network to a neuromorphic crossbar-based accelerator. The original first steps that were depicted in Figure 5 are substituted by the SNN conversion and software emulation within SODA; the missing link before decomposition,

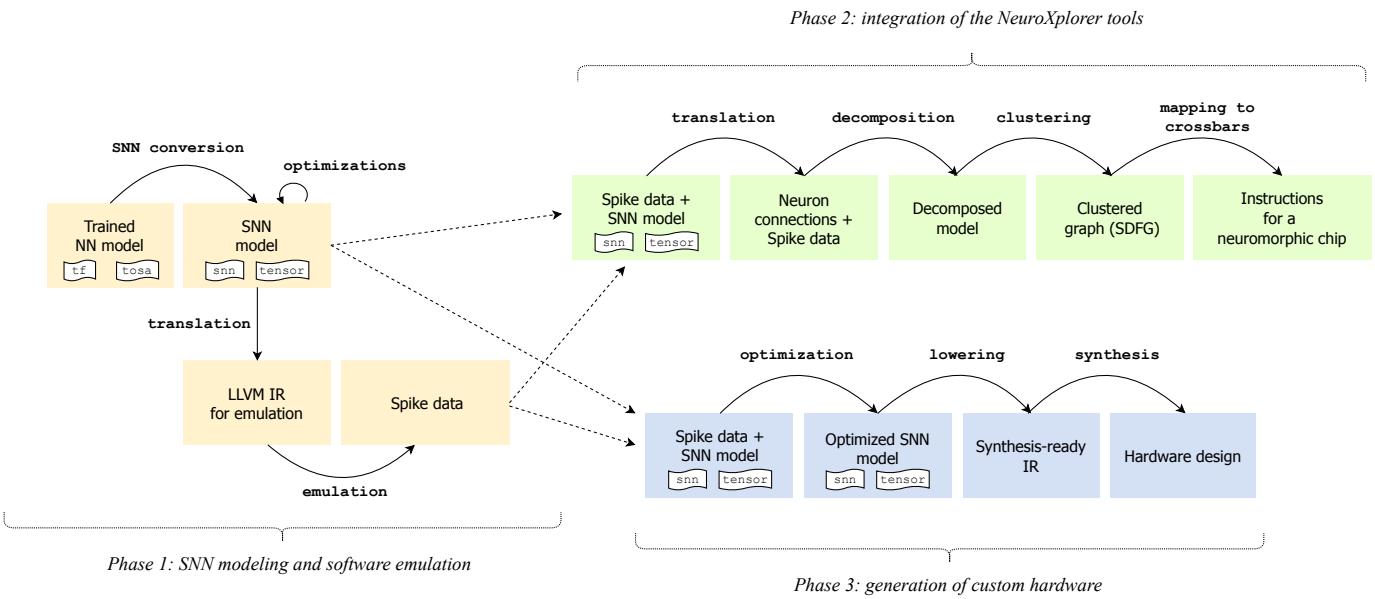


Fig. 7. Evolution of the SNN flow in SODA.

clustering, and mapping is a translation from the MLIR SNN model into NeuroXplorer's internal representation of spiking neurons connectivity and their individual spiking behaviour.

Instead of relying on existing neuromorphic architectures, the third development phase aims at generating custom hardware designs, possibly combining analog and digital elements. Here, the SNN model generated by the frontend is further optimized in the SODA middle-end and backend, where information about the target hardware can be exploited alongside the spike data gathered during software emulation. The final form of the SNN model after the middle-end transformations is again an IR, in a synthesis-ready abstraction suited to the backend tools that will generate the accelerator design in a hardware description language. In the existing SODA HLS flow, the synthesis IR is an LLVM IR that is fed to Bambu HLS to generate Verilog code for ASIC/FPGA. For a neuromorphic architecture the synthesis IR may need to describe both analog and digital blocks, and the interfaces between them; the synthesizer would then combine pre-existing IPs and custom logic to generate the design and integrate it in a full system with other accelerators generated by SODA flows.

B. SNN dialect

The proposed `snn` dialect implements an intermediate representation that captures intrinsics of SNN models. In its design we developed *types* to represent the data that flows during SNN execution and *operations* representing the transformations applied to the data. Figure 8 shows the key concepts that we strive to capture in the dialect.

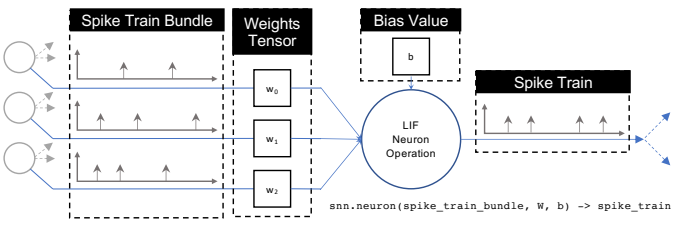


Fig. 8. The snn.neuron operation.

Types - The MLIR code below represents the train of timestamps on which spikes occur and N-dimension bundles of spike trains. These are the key types in the `snn` dialect.

```
!spkt_t = snn.spike_train<?xf32>
!spk1db_t = snn.spike_bundle<?x snn.spike_train<?xf32>>
!spk2db_t = snn.spike_bundle<?x?x snn.spike_train<?xf32>>
```

Spike train conversions - We use the following operations to generate a spike train from a list of times or extract the list of spike times from a spike train, converting our types into or out of the SNN domain.

```
%spike_train = snn.encode_spike_train(%list_of_times :
    tensor<?xf32>) : !spkt_t
%list_of_times = snn.decode_spike_train(%spike_train :
    !spkt_t) : tensor<?xf32>
```

Spike bundle conversions - The following MLIR operations are used to perform bulk conversions of inputs to *bundles of spike trains*. Each input inside the tensor is converted into a unique spike train.

```
%spk_bundle = snn.tensor_to_spike_train_bundle(data :
    tensor<?x?xf32>) : !spk2db_t
%data = snn.spike_bundle_to_tensor(%sb : !spk2db_t ) :
    tensor<?x?xf32>
```

Spike train slicing and bundling - We propose the following operations to *group several spike trains in a bundle* or *select a spike train from a bundle*. The `snn` dialect implements additional operations in this category to provide finer grained control of spike train bundles that ease dataflow analysis, however these operations are omitted here for simplicity.

```
%spk_train = snn.spike_bundle_select (%spk_bundle[%i, %j] :
     !spk2db_t) : !spkt_t
%spk_bundle2 = snn.spike_bundle_collect (%spk : !spkt_t ,
    %spk : !spkt_t , ...) : !spk1db_t
```

Neuron operations - Currently modeling the behaviour of LIF neurons, the following operations represent the analog transformation shown in Figure 2 and Figure 8. The output spike train generated by this operation takes in consideration the input spike trains multiplied by the weights (modeling LIF neuron resistances R), the neuron Bias (that can increase the neuron's resting potential), and the LIF neuron intrinsic attributes of Vth, $Vrest$, k, discussed in Section III-C.

```
%id0 = snn.neuron (%in : !spkt_t) : !spkt_t // Input neuron
%id1 = snn.neuron (%in_spikes : !spk1db_t , W :
    tensor<?xf32> , bias : f32) : !spkt_t
```

Ordering operations - These operations create a tensor with the direct (or reverse) order of which spike trains spiked first. The output tensor has the same size as the 1-D bundle size. These operations are

used in evaluation of which category is most important on a category classification problem.

```
%d_order = snn.get_order(%out_spikes : !spk1db_t) :
     tensor<?xindex>
%r_order = snn.get_rev_order(%out_spikes : !spk1db_t) :
     tensor<?xindex>
```

C. SNN example

Figure 9 shows an example of 2-input 2-output fully connected SNN model described using the `snn` dialect. It demonstrates how a SNN model can interface with inputs in the digital domain represented with tensors by using *spike train conversions*, *spike bundle conversions*, and *ordering operations*, as well as how spike trains are manipulated in the SNN domain to represent our fully connected model.

```
module attributes {soda.snn.container_module, snn.lif.k=1,
                   snn.lif.vth=30, snn.lif.vrest=-20}{
  !spkt_t = snn.spike_train<?xf32>
  !spk1db_t = snn.spike_bundle<?x snn.spike_train<?xf32>>

  func @dense_2(%data: tensor<2xf32> ,
                %W : tensor<2x2xf32> ,
                %B : tensor<2xf32>) -> tensor<2xindex>){
    // Use tensor.extract_slice and tensor.extract ops
    //   to collect %w_1_0, %w_1_1, %b_1_0, %b_1_1
    //   from %W and %B input arguments.

    // Input Layer
    %in = snn.tensor_to_spike_train_bundle(%data :
     tensor<2xf32>) : !spk2db_t
    %in_0 = snn.spike_bundle_select (%in[0] : !spk1db_t) :
     !spkt_t
    %in_1 = snn.spike_bundle_select (%in[1] : !spk1db_t) :
     !spkt_t
    %v_0_0 = snn.neuron (%in_0 : !spkt_t)
    %v_0_1 = snn.neuron (%in_1 : !spkt_t)
    %v_0 = snn.spike_bundle_collect (%v_0_0 : !spkt_t ,
     %v_0_1 : !spkt_t) : !spk1db_t

    // Hidden/Output Layer
    %v_1_0 = snn.neuron (%v_0 : !spk1db_t , %w_1_0 :
     tensor<2xf32> , %b_1_0 : f32 )
    %v_1_1 = snn.neuron (%v_0 : !spk1db_t , %w_1_1 :
     tensor<2xf32> , %b_1_1 : f32 )
    %v_1 = snn.spike_bundle_collect (%v_1_0 : !spkt_t ,
     %v_1_1 : !spkt_t) : !spk1db_t

    // Output transformation
    %order = snn.get_order(%out_spikes : !spk1db_t) :
     tensor<?xindex>
    %out = tensor.cast %order : tensor<?xindex> to
     tensor<2xindex>
    return %out
} }
```

Fig. 9. Two-Neuron fully connected model.

In additional to the types and operations previously described, Figure 9 also presents important `module` attributes associated with LIF neurons characteristics shared by all neurons in a given target chip. The `snn.lif.vth` and `snn.lif.vrest` attributes represent the firing threshold and resting potential in mV of a LIF neuron; and `snn.lif.k` represents the constant that multiplies the neuron's discharge rate dv/dt. These values must be know for optimizations such as neuron decomposition [36] or to perform SNN emulation.

The example described in Figure 9 provides abstractions in two domains, a purely digital domain captured by tensors and their values, and the SNN domain, abstracted by types and operations in our proposed dialect. Although analog models ultimately govern some SNN operations (such as the LIF neuron operation), digital logic circuits can mimic their behavior. Henceforth, the `snn` dialect can be lowered, translated, and mapped to purely digital resource elements. In this way, phase three of our integration may start from adapting the existing HLS-based flow to support the digital versions of SNN operations, and later enable an analog resource library with dedicated middle-end and backend for mixed-circuit design.

IV. Conclusion

This paper discusses the opportunities to integrate conventional artificial neural networks techniques and spiking neural networks elements by providing a modern, modular, multi-level, end-to-end compiler-based infrastructure that automatically generates domain-specific accelerators. Specifically, we presented our initial approach for the integration of the NeuroXplorer framework with the SODA Synthesizer. Leveraging the MLIR framework, the SODA frontend allows defining and interfacing with a new specialized IR (the SNN dialect) that deals with the complexities related to SNN representation and mapping. The SODA synthesizer middle-ends provide support to generate Verilog code representing spiking neurons in digital logic that can be mapped onto FPGAs or ASICs. Down the road, the adoption of interoperable hardware compiler-based infrastructures can further provide opportunities to define and integrate, in a single complex heterogeneous system, ML accelerators composed of both digital and analog components. The SODA synthesizer will be a crucial element providing compiler methods to map the software onto the hardware, and to solve challenges connected to the integration of modules that leverage intrinsically different technologies.

References

[1] J. Hennessy and D. Patterson, "A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 27–29.

[2] C. Leary and T. Wang, "XLA: TensorFlow, compiled," 2017. [Online]. Available: https://www.tensorflow.org/xla

[3] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein, J. Montgomery, B. Maher, S. Nadathur, J. Olesen, J. Park, A. Rakhov, M. Smelyanskiy, and M. Wang, "Glow: Graph lowering compiler techniques for neural networks," 2019.

[4] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," in *OSDI*. USA: USENIX Association, 2018, p. 579–594.

[5] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis *et al.*, "Fast inference of deep neural networks in FPGAs for particle physics," *Journal of Instrumentation*, vol. 13, no. 07, p. P07027, 2018.

[6] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu *et al.*, "FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 3, pp. 1–23, 2018.

[7] 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*. San Diego, CA, USA: IEEE, 2020, pp. 1–9.

[8] 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 *International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–7.

[9] 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 *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2021, pp. 218–225.

[10] 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 *CGO*. Seoul, Korea (South): IEEE, 2021, pp. 2–14.

[11] M. V. Debole, B. Taba, A. Amir, F. Akopyan, A. Andreopoulos, W. P. Risk, J. Kusnitz, C. O. Otero, T. K. Nayak, R. Appuswamy, P. J. Carlson, A. S. Cassidy, P. Datta, S. K. Esser, G. J. Garreau, K. L. Holland, S. Lekuch, M. Mastro, J. Mckinstry, C. Di Nolfo, J. Sawada, B. Paulovicks, K. Schleupen, B. G. Shaw, J. L. Klamo, M. D. Flickner, J. V. Arthur, and D. S. Modha, "TrueNorth: Accelerating from zero to 64 million neurons in 10 years," *Computer*, 2019.

[12] M. Davies, N. Srinivasa, T. H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C. K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y. H. Weng, A. Wild, Y. Yang, and H. Wang, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, 2018.

[13] J. Hasler, "Large-Scale Field-Programmable Analog Arrays," *Proceedings of the IEEE*, vol. 108, no. 8, pp. 1283–1302, 2020.

[14] S. Panchapakesan, Z. Fang, and J. Li, "SyncNN: Evaluating and Accelerating Spiking Neural Networks on FPGAs," in *International Conference on Field-Programmable Logic and Applications (FPL)*, 2021.

[15] A. Balaji, S. Song, T. Titirsha, A. Das, J. Krichmar, N. Dutt, J. Shackleford, N. Kandasamy, and F. Catthoor, "NeuroXplorer 1.0: An extensible framework for architectural exploration with spiking neural networks," in *ICONS*, 2021.

[16] 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*. IEEE, 2021.

[17] 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.

[18] P. S. Li, A. M. Izraelevitz, and J. Bachrach, "Specification for the FIRRTL Language," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-9, Feb 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-9.html

[19] C. Developers, ""CIRCT" / Circuit IR Compilers and Tools," 2020. [Online]. Available: https://github.com/llvm/circt

[20] W. Maass, "Networks of spiking neurons: The third generation of neural network models," *Neural Networks*, 1997.

[21] S. Fusi and M. Mattia, "Collective behavior of networks with linear (VLSI) integrate-and-fire neurons," *Neural Computation*, 1999.

[22] A. Tavanaei, M. Ghodrati, S. R. Kheradpisheh, T. Masquelier, and A. Maida, "Deep learning in spiking neural networks," *Neural Networks*, 2019.

[23] A. Das, P. Pradhapan, W. Groenendaal, P. Adiraju, R. Rajan, F. Catthoor, S. Schaafsma, J. Krichmar, N. Dutt, and C. Van Hoof, "Unsupervised heart-rate estimation in wearables with Liquid states and a probabilistic readout," *Neural Networks*, 2018.

[24] M. S. Shim and P. Li, "Biologically inspired reinforcement learning for mobile robot collision avoidance," in *IJCNN*, 2017.

[25] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana, "The SpiNNaker project," *Proceedings of the IEEE*, 2014.

[26] X. Liu, W. Wen, X. Qian, H. Li, and Y. Chen, "Neu-NoC: A high-efficient interconnection network for accelerated neuromorphic systems," in *ASP-DAC*, 2018.

[27] A. Balaji, Y. Wu, A. Das, F. Catthoor, and S. Schaafsma, "Exploration of segmented bus as scalable global interconnect for neuromorphic computing," in *GLSVLSI*, 2019.

[28] C. Liu, B. Yan, C. Yang, L. Song, Z. Li, B. Liu, Y. Chen, H. Li, Q. Wu, and H. Jiang, "A spiking neuromorphic design with resistive crossbar," in *DAC*, 2015.

[29] Y. Ji, Y. Zhang, S. Li, P. Chi, C. Jiang, P. Qu, Y. Xie, and W. Chen, "NEUTRAMS: Neural network transformation and co-design under neuromorphic hardware constraints," in *MICRO*, 2016.

[30] A. Balaji, A. Das, Y. Wu, K. Huynh, F. G. Dell'anna, G. Indiveri, J. L. Krichmar, N. D. Dutt, S. Schaafsma, and F. Catthoor, "Mapping spiking neural networks to neuromorphic hardware," *TVLSI*, 2020.

[31] A. Amir, P. Datta, W. P. Risk, A. S. Cassidy, J. A. Kusnitz, S. K. Esser, A. Andreopoulos, T. M. Wong, M. Flickner, R. Alvarez-Icaza *et al.*, "Cognitive computing programming paradigm: a corelet language for composing networks of neurosynaptic cores," in *IJCNN*, 2013.

[32] T. Titirsha, S. Song, A. Balaji, and A. Das, "On the role of system software in energy management of neuromorphic computing," in *CF*, 2021.

[33] S. Song, J. Hanamshet, A. Balaji, A. Das, J. Krichmar, N. Dutt, N. Kandasamy, and F. Catthoor, "Dynamic reliability management in neuromorphic computing," *JETC*, 2021.

[34] T. Titirsha, S. Song, A. Das, J. Krichmar, N. Dutt, N. Kandasamy, and F. Catthoor, "Endurance-aware mapping of spiking neural networks to neuromorphic hardware," *TPDS*, 2021.

[35] S. Song, T. Titirsha, and A. Das, "Improving inference lifetime of neuromorphic systems via intelligent synapse mapping," in *ASAP*, 2021.

[36] S. Song, L. V. Mirtinti, A. Das, and N. Kandasamy, "A design flow for mapping spiking neural networks to many-core neuromorphic hardware," in *ICCAD*, 2021.

[37] C.-K. Lin, A. Wild, G. N. Chinya, T.-H. Lin, M. Davies, and H. Wang, "Mapping spiking neural networks onto a manycore neuromorphic architecture," in *PLDI*, 2018.

[38] A. P. Davison, D. Brüderle, J. M. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger, "PyNN: a common interface for neuronal network simulators," *Frontiers in Neuroinformatics*, 2009.

[39] A. Balaji, P. Adiraju, H. J. Kashyap, A. Das, J. L. Krichmar, N. D. Dutt, and F. Catthoor, "PyCARL: A PyNN interface for hardware-software co-simulation of spiking neural network," in *IJCNN*, 2020.

[40] T. Chou, H. Kashyap, J. Xing, S. Listopad, E. Rounds, M. Beyeler, N. Dutt, and J. Krichmar, "CARLsim 4: An open source library for large scale, biologically detailed spiking neural network simulation using heterogeneous clusters," in *IJCNN*, 2018.

[41] D. F. Goodman and R. Brette, "The brian simulator," *Frontiers in Neuroscience*, 2009.

[42] J. M. Eppler, M. Helias, E. Muller, M. Diesmann, and M.-O. Gewaltig, "PyNEST: a convenient interface to the NEST simulator," *Frontiers in Neuroinformatics*, 2009.

[43] M. L. Hines and N. T. Carnevale, "The NEURON simulation environment," *Neural Computation*, 1997.

[44] A. Balaji, S. Song, A. Das, J. Krichmar, N. Dutt, J. Shackleford, N. Kandasamy, and F. Catthoor, "Enabling resource-aware mapping of spiking neural networks via spatial decomposition," *ESL*, 2020.

[45] A. Das, Y. Wu, K. Huynh, F. Dell'Anna, F. Catthoor, and S. Schaafsma, "Mapping of local and global synapses on spiking neuromorphic hardware," in *DATE*, 2018.

[46] S. Song, A. Balaji, A. Das, N. Kandasamy, and J. Shackleford, "Compiling spiking neural networks to neuromorphic hardware," in *LCTES*, 2020.