VeriGOOD-ML: An Open-Source Flow for Automated ML Hardware Synthesis

Hadi Esmaeilzadeh[1], Soroush Ghodrati[1], Jie Gu[2], Shiyu Guo[2], Andrew B. Kahng[1], Joon Kyung Kim[1], Sean Kinzer[1], Rohan Mahapatra[1], Susmita Dey Manasi[3], Edwin Mascarenhas[1], Sachin S. Sapatnekar[3], Ravi Varadarajan[1], Zhiang Wang[1], Hanyang Xu[1], Brahmendra Reddy Yatham[1], Ziqing Zeng[3]
[1]UC San Diego [2]Northwestern University [3]University of Minnesota

***Abstract*—This paper introduces VeriGOOD-ML, an automated methodology for generating Verilog with no human in the loop, starting from a high-level description of a machine learning (ML) algorithm in a standard format such as ONNX. The Verilog RTL is then translated through a back-end design flow to GDSII, driven by a design planning approach that is well tailored to the macro-intensive nature of ML platforms. VeriGOOD-ML uses three approaches to build ML hardware: the TABLA platform uses a dataflow architecture that is well suited to non-DNN ML algorithms; the GeneSys platform, with a systolic array and a SIMD array, is optimized for implementing DNNs; and the Axiline approach synthesizes small ML algorithms by hardcoding the structure of the algorithm into hardware, thus trading off flexibility for performance and power. The overall approach explores the design space of platform configurations and Pareto-optimal-PPA back-end implementations to yield designs that represent different tradeoffs at the algorithmic level between area, power, performance, and execution time. The overall methodology, from architecture to back-end design to hardware implementation, is described in this paper, and the results of VeriGOOD-ML are demonstrated on a set of ML benchmarks.**

I. Introduction

Recent advances in machine learning (ML) algorithms have seen a proliferation of new ML algorithms and architectures, as well as new work on ML accelerators. However, the design of these accelerators requires intense manual designer effort and is time-consuming. There is considerable recent interest in real-time machine learning (RTML), where data is sent to an ML accelerator chiplet through fast interfaces [1] and processed on the chiplet in real time, with applications ranging from ML tasks in autonomous vehicles (e.g., obstacle detection, collision avoidance, path planning) to next-generation wireless networks (e.g., resource sharing in virtualized radio access networks, channel estimation, channel decoding, RF fingerprinting). These applications are best supported by building an ability for rapid translation from an ML algorithm to a hardware implementation.

VeriGOOD-ML (**Veri**log **G**enerator (**O**pen-source), **O**ptimized for **D**esigns for **M**achine **L**earning) is an open-source project [2] that automatically compiles a high-level description of an ML algorithm (in a standard ML format such as ONNX) to a register-transfer level (RTL) Verilog implementation with no human in the loop. The RTL is then taken through synthesis/place-and-route, resulting in a silicon implementation. The entire design flow, from architecture design to physical implementation, is guided by models for performance, power, and area (PPA), working in conjunction with architectural simulation. This enables the designer to perform cross-layer optimizations to build high-performance design implementations that can be optimized for various objectives: size, power, performance, or solution quality (using bitwidth quantization).

The ML algorithm is specified using the Open Neural Network Exchange (ONNX) format, which is widely supported, thus maximizing interoperability across various programming environments, including Google Tensorflow, Microsoft CNTK, and Facebook PyTorch. The starting point for VeriGOOD-ML is the PolyMath compiler [3], which translates a high-level ML algorithm description (e.g., ONNX) into our intermediate representation (IR). The IR is a representation that we refer to as a simultaneous recursive dataflow graph (*sr*-DFG) that allows a hierarchical view into the structure of a design.

VeriGOOD-ML targets ML engines for both training and inference. It uses three core engines to synthesize hardware from the IR. Two of these are platform-based: **TABLA** [4], for general non-DNN ML algorithms (e.g., linear regression, logistic regression, SVM), and **GeneSys** for general DNN algorithms. TABLA uses a dataflow architecture; the core computation engines in GeneSys are a systolic array (for operations such as convolution) and a SIMD array (for operations such as ReLU and pooling). The platforms are parameterizable, and it is possible to automatically generate hardware with different numbers of processing elements, bitwidths, and on-chip memory configurations. A third approach, **Axiline**, is a hard-coded engine tailored to specific small ML algorithms: it trades off the flexibility of a platform, which can run multiple ML algorithms, for a power-efficient implementation that is tailored to a single algorithm. For TABLA and GeneSys, the platform-based architectures, PolyMath translates the *sr*-DFG into templates that implement the ML algorithm on an instruction set that is specific to the platform. The Axiline implementation is synthesized by translating the *sr*-DFG into dedicated hardware. Our silicon implementation efforts characterize the PPA of core building blocks and develop methodologies to go from RTL to physical implementations at various PPA points.

Throughout the flow, VeriGOOD-ML optimizes the design for performance, producing a set of designs with Pareto-optimal performance/power/area (PPA) tradeoffs, and connecting these with system-level performance metrics that optimize the power and execution time for implementing an ML algorithm. In particular, a design planner, which performs floorplanning and power grid generation for the macro-intensive layout, is vital in ensuring that the back-end implementation delivers high performance. The flow includes cycle-accurate simulators for each engine, and is coupled with silicon PPA predictors that can be used to perform design-space exploration, yielding optimized ML hardware engines.

II. Compiling ONNX to Platform-Specific Instructions

In this section, we describe how the ONNX description of an ML algorithm is converted to an intermediate representation (IR), and together with information about the hardware, is used to perform end-to-end compilation using the PolyMath framework [3] for execution on TABLA, GeneSys, and Axiline.

Intermediate representation using an *sr*-DFG: To encapsulate operations at multiple levels of hierarchy, we devise a simultaneous recursive dataflow graph (*sr*-DFG), an IR that is recursively defined with respect to the *sr*-DFG nodes. The representation facilitates optimization in several ways: (1) utilizing optimizations that are

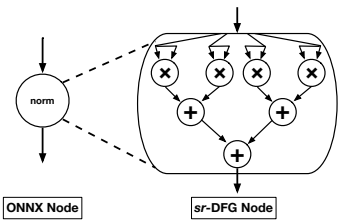


Figure 1: Translation of a "norm" ONNX node to the equivalent *sr*-DFG node that contains all the fine-grained operations that comprise a "norm" operation.

predeveloped for certain complex operations (e.g., building a binary tree for the L2 norm or optimizing the flow of data for convolution) and (2) simultaneously preserving the capability to perform fine-grained scheduling and mapping optimization.

To translate an ONNX description into an *sr*-DFG, we traverse the ONNX graph, whose nodes represent coarse-grained ML operations on multi-dimensional arrays of input data. During traversal, *sr*-DFG nodes and edges are generated using the attributes and inputs/output of each ONNX operation. The operations that comprise each coarse-grained operation (e.g., multiply-adds that constitute a norm operation, as shown in Fig. 1) are added to each *sr*-DFG node using instantiations of predefined templates. We have successfully created *sr*-DFG representations for a variety of benchmarks that cover a variety of machine learning algorithms – both non-DNN (linear regression, logistic regression, support vector machines, recommender systems, backpropagation) and DNN ML algorithms.

Modeling hardware using a HAD: We model the structure of specific accelerator platforms by introducing a reusable hardware abstraction called a hierarchical architecture description (HAD), with a corresponding architecture description language embedded in Python for targeting different types of accelerators with a unified interface. A series of compilation passes use the HAD for a specific target accelerator for mapping, memory allocation, and optimizing programs on the accelerator. Each HAD is comprised of three types of components: for functional unit calculations, for on- and off-chip communication, and for memory. In interaction with the *sr*-DFG, the HAD enables end-to-end compilation by the introduction of hardware-specific attributes to the compilation pipeline.

An architecture description language (ADL) is used to represent the HAD. Such an abstraction enables the compiler to expand its capability from optimizing for a single piece of hardware to a heterogeneous computing context with multiple disparate processors and accelerators. This ADL is built on top of Python to improve usability and versatility, easily working in tandem with various machine learning frameworks. To represent diverse types of accelerators, several primary attributes must be included in the abstraction, to provide the ability to enable (a) modeling of the hierarchy (as fine-grained as a single ALU, or as coarse-grained as an entire systolic array); (b) specification of compute, storage, and communication components; and (c) annotation of each node with attributes/metadata including, but not limited to, storage node capacity, communication bandwidth, input and output ports, latency, and computation node capabilities that describe operations supported by the architecture.

Compilation to target accelerators: Having devised an architecture description of different types of accelerator architectures, a multistage compilation process can be reused across different HADs. The stages of compilation, illustrated in Fig. 2, are:

Template mapping/scheduling: An *sr*-DFG is ordered to a sequence of operations, and each operation will be mapped to a particular component in the HAD according to the *sr*-DFG node operation and the sequence of functional unit templates that produce the equivalent operation. In addition, sequences of templates can be fused together according to user-supplied parameters.

Compilation optimization: A search for optimal compilation parameters is performed using specifications of the HAD, such as tiling sizes, loop unrolling factors, dataflow, etc. During this process, data communication instructions/operations, including off-chip communications for both read/store operations, are added according to these parameters.

Code generation for the target HAD: This step is based on the instantiated functional unit templates from the two previous steps. The compiler combines code templates with the *sr*-DFG node attributes and HAD attributes. These templates represent instruction templates for target accelerators. The *sr*-DFG is converted to this abstraction for every type of accelerator, with the only difference being the underlying instruction template used for binary generation. There are four primary types of templates: (i) Functional templates that represent instructions for implementing functions that act on data; (ii) Memory templates for instructions that move data from one memory location to another (e.g., load from/store to off-chip or on-chip memory); (iii) Iteration templates that repeat operations over a number of loop levels; and (iv) Control templates for instructions that determine program flow. These templates are combined to form operations that match the semantics of execution for a given *sr*-DFG node.

The overall compilation flow is depicted in Fig. 3, which demonstrates the different stages as well as the ability to apply architectural attributes to the compiler passes. In combination with the code templates associated with templates, additional compiler passes were implemented to optimize and transform the program, e.g., datatype transformations, layout transformations, and padding tensors for GeneSys to map data onto the systolic array and SIMD array.

III. Target Hardware Substrates

In this section, we overview three target substrates for VeriGOOD-ML: TABLA for non-DNN ML algorithms, GeneSys for DNNs, and Axiline for ultraefficient hard-coded implementations of small ML algorithms.

A. The TABLA Platform for Non-DNN ML Algorithms

Overview of the TABLA architecture: The overall TABLA architecture [4] for training and inference for non-DNN ML algorithms is shown in Fig. 4 and consists of multiple levels of hierarchy. An array of *processing units* (PUs) constitutes the first level. The PUs are connected through two different busing mechanisms – the "neighbor bus" and the "global bus." All PUs are connected to the global bus, and the communication between all the PUs imposes a high pressure on the global bus. The neighbor bus aims to minimize this pressure by enabling the adjacent PUs to send their data through it. Moreover, connecting all PUs to the global bus can result in a race between the PUs. To ensure proper data transfer between PUs, a bus arbitration module is implemented.

At the next level of hierarchy, each PU comprises of a set of *processing engines* (PEs). Similar to buses for inter-PU communication, there are two buses for inter-PE communication. The *bus arbiter* consists of a single leader controller per PU and one follower controller for each PE. The leader controller determines which PE has control of the bus in a given cycle, and the follower controller has a write buffer and a set of read buffers (one for each PE/PU), organized

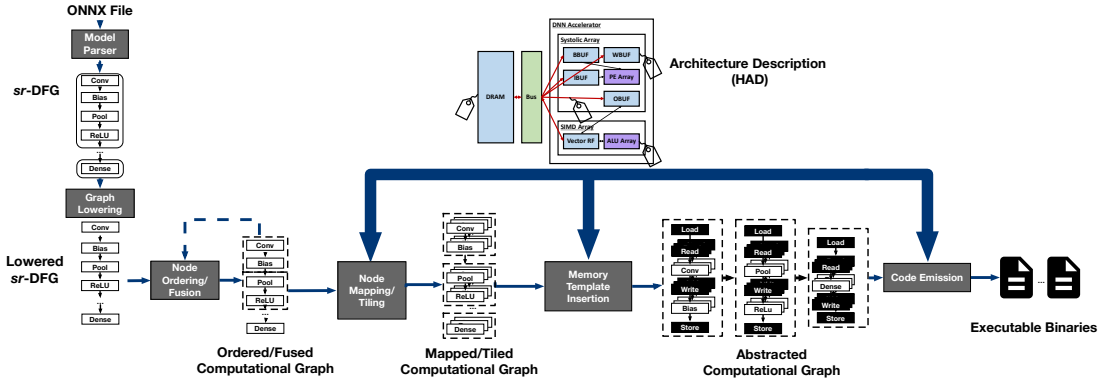


Figure 2: ONNX-to-hardware mapping flow through the *sr*-DFG and HAD representations.

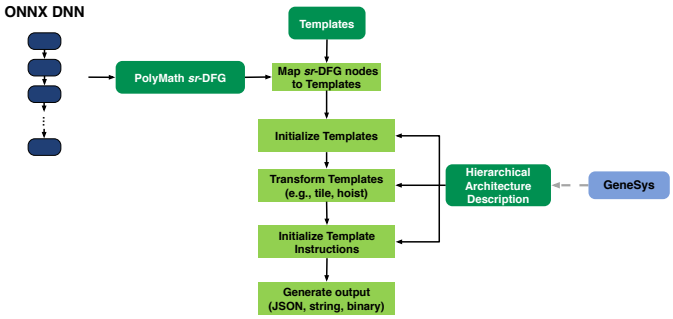


Figure 3: Compilation flow combining the HAD and templates to apply multiple stages of transformation and optimization.

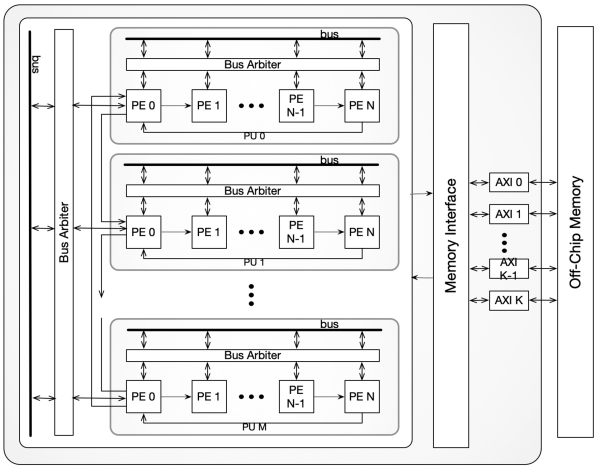


Figure 4: An overview of the TABLA template-based architecture.

as FIFOs. In each cycle, data is popped from the write buffer of the source PE and written to the read buffer of the destination PE.

Cycle-accurate software simulator: To facilitate testing and verification of the architecture, we have designed and developed a cycle-accurate simulator in software that emulates the architectural behaviors of the proposed system. The simulator allows the user to provide the input program as an *sr*-DFG file and a configuration file that sets the parameters – such as the number of PEs per PU – of the template architecture described in the above sections. Taking the configuration file as an input allows users to further test the behavior of the architecture with varying degrees of parameterization, e.g., to

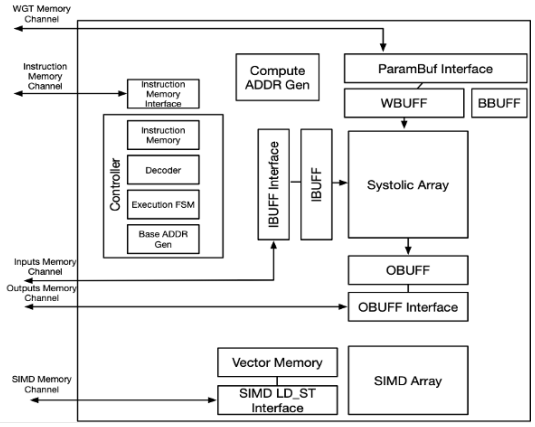


Figure 5: A block diagram of the overall system view of GeneSys, the VeriGOOD-ML DNN accelerator.

analyze the performance impact of changing the number of PEs per PU. Based on cycle-by-cycle analysis, the simulator can emulate the execution of a given program and output performance metrics such as total number of cycles, PE and PU utilization, and scratchpad utilization.

B. The GeneSys Platform for DNN Algorithms

Overview of the GeneSys architecture: The overall system view of the GeneSys DNN accelerator is shown in Fig. 5. The accelerator consists of two core components: a systolic array and a SIMD array. Data is supplied to the engine through the input buffer (IBUFF), output buffer (OBUFF), instruction memory (IMEM), weight buffer (WBUFF), and bias buffer (BBUFF). These interfaces harbor programmable data access modules and controller FSMs that together issue the addresses and requests to load or store a tile of data from/to off-chip memory. The data access creates strided patterns that access the off-chip memory to read/write the corresponding data from/to on-chip buffers and populate the on-chip memory. These interfaces also include tag logic that is in charge of handling double-buffered data transfer to hide the latencies of Load/Store operations and also facilitate prefetching. Among these interfaces, the interfaces for the OBUFF and SIMD array handle both load and store operations, while the other interfaces handle only load operations. These interfaces are fully programmable through the instruction set architecture (ISA) of the GeneSys accelerator.

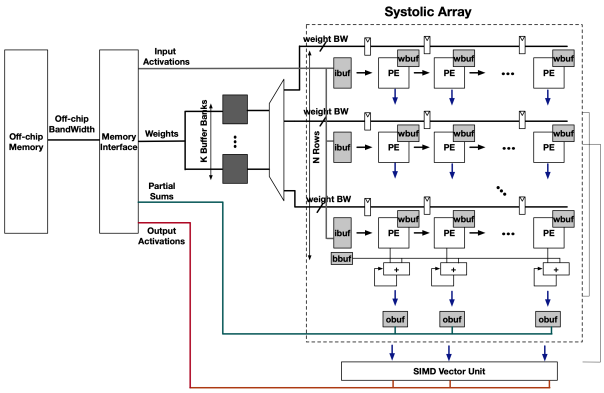


Figure 6: Execution flow of the GeneSys systolic array accelerator.

The **systolic array**, which performs convolution and matrix multiplication operations for the convolution and fully-connected layers, is a 2D array of $M \times N$ processing engines (PEs), equipped with dedicated on-chip weight buffers, as in [5], [6]. To boost the operating frequency, we pipeline the inputs and weights across the columns of the array and the partial sums across the rows of the array. In systolic execution, the inputs (activations) flow horizontally, are multiplied by the weights in each PE and are then accumulated vertically along the columns of the systolic array. This systolic execution also facilitates mapping the matrix-multiplications and convolutions to the array and simplifies the control logic. The IBUFF is multibanked and each bank feeds a row of the systolic array. The output buffers are also multibanked, each bank for each column of the systolic array, storing the partial sums and output activations.

Fig. 6 depicts a more detailed diagram of the implementation of the systolic array. Each processing engine consists of (1) a weight scratchpad that stores the weight values on-chip and (2) a multiply-accumulate unit that performs a multiplication between the inputs and weights and an accumulation of the partial results to perform the matrix-multiplication or convolution operation with the systolic array. Each PE is equipped with four registers that aim to support the pipelined execution: a register for the output results, a register for the received input that will be forwarded to the adjacent PE in the systolic array, and two registers for handling the read accesses from the weight scratchpad (one register for the read request and one for the read address; the read request and read addresses for the weight scratchpads are shared across the 2-D array of PEs). Each PE is a template design, with parameterizable parameters that include the size of the weight scratchpad, the precision of the input, weight, partial sum, and the bitwidth of the multiply-accumulate logic and registers. This parameterization is performed during architectural synthesis according to the demands of the application.

For address generation, we design a data access module that can automatically access memory to execute convolution/matrix-multiplication operations on the systolic array, leveraging the insight that the data layout and memory patterns of DNNs are generally regular, without branch/jump instructions. This module is configured with parameters such as the number of loop iterations.

The **SIMD vector unit** is a $1 \times N$ array that performs computations for DNN layers other than convolution and fully-connected layers, such as pooling, activation, and other element-wise operations. The pipeline stages of this SIMD processor are generally similar to a MIPS processor with a major difference: since memory access patterns in DNNs are regular, the register file is eliminated to save Load/Store instructions. With this design, we directly read from the on-chip scratchpads that store the data, execute the operations, and then write it back to the destination scratchpad. We have designed a custom ISA to program this architecture. There are two classes of instructions in this ISA: execution instructions (ALU, CALCULUS, COMPARISON, DATATYPE CAST), and setup instructions (DATATYPE CONFIG, ITERATOR CONFIG, LOOP).

A training-capable GeneSys implementation consists of additional layers and operations beyond the inference engine for performing gradient computations and parameter updates. Training operations must support computations of loss gradient with respect to input and weight: for a convolution layer, these are mapped to a convolution operation, and for a fully connected layer, they are implemented as a GEMM operation. For training, GeneSys supports a softmax layer, a common generic model for multiple operations (e.g., parameter updates for 1D, 2D, and 4D tensors; loss gradient computation for the ReLU layer and for element-wise addition of two tensors; reduction of a tensor along its dimensions), and estimated models for the batch normalization layer, including operations during the forward and backward pass.

GeneSys performance simulator: Our simulator for DNN execution on GeneSys takes the following two files as inputs: (1) a specification of the hardware configuration, in the form of a .json file, and (2) the compiler output, as a .json file containing a high-level description of each DNN layer, e.g., the dimensions of the input/output tensors, order of execution of the loops, tile sizes for the tensors, and datatypes.

The simulation framework is attuned to the fully parameterizable nature of GeneSys by accepting the specific hardware attributes:

- the dimensions of the 2D PE array, the sizes of each of the on-chip buffers, namely, WBUFF, IBUFF, OBUFF, and BBUFF for the systolic array, and vector memory, immediate memory, and instruction memory buffers for the SIMD array.
- bit-widths of all types of data (filter, input, bias, psum, output for the systolic array; input, psum, output for the SIMD array).
- the number of cycles required by various arithmetic operations.
- off-chip bandwidth of each memory interface.

For each layer of a DNN, either executed on the systolic array or SIMD array, the simulator outputs the following performance statistics: the number of accesses for each of the on-chip buffers for each datatype, the number of accesses for the off-chip DRAM for each datatype, the number of accesses for the pipeline registers, the number of various arithmetic operations, the number of on-chip compute cycles, the number of stall cycles while the systolic array or SIMD array remain idle waiting for data to be fetched from the off-chip DRAM, and the total number of execution cycles.

C. The Axiline Approach for Hard-Coded ML Hardware

The Axiline generator develops dedicated, hard-coded implementations of small algorithms, for both ML training and inference, to achieve high performance and low power. For TABLA and GeneSys, the parameters for the platform can be selected according to target applications, but may be used to run other applications. In contrast, Axiline is intended to be very specific to the ML algorithm that it implements, and it trades off adaptability for performance. By building a hard-coded implementation, we can achieve maximum performance and efficiency, at the expense of flexibility.

The Axiline generator outputs RTL by creating a mapping from an *sr*-DFG input to unit constructs such as inner products, adders, and multipliers. The simplest version of Axiline begins with an *sr*-DFG without loops and translates it to a combinational implementation.

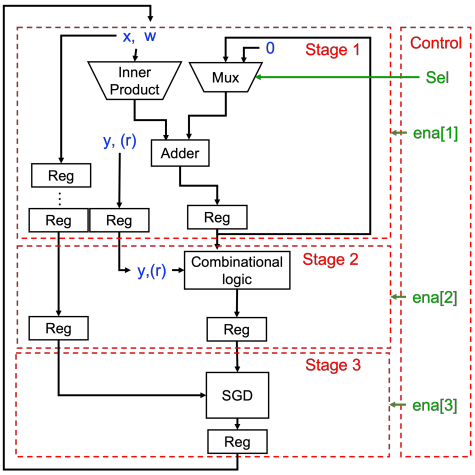


Figure 7: Pipeline implementation for Axiline benchmarks.

However, the cost of implementing a larger *sr*-DFG, or one with loops, may become prohibitive due to the large volume of data to be processed. For such scenarios, we develop an iterative architecture that serially processes parts of the input data over multiple cycles.

The generator works in three steps: first, it generates the lowered data flow graph for an Axiline ML algorithm; next, it calculates the bitwidth for each node, based on the given bitwidth of activation, weight and bias, and finally, it generates Verilog code for each node/block and combines them with the template. A representative multicycle pipelined architecture that can be used for several non-DNN benchmarks (e.g., SVM, logistical regression, and linear regression) is shown in Fig. 7. The architecture maps the *sr*-DFG into three pipeline stages: Stage 1 performs an inner product computation, and is followed by Stage 2, which implements a combinational function, where the precise function depends on the benchmark. For example, for linear regression, the combinational logic in block 2 would be a multiplier, and for logistic regression benchmark, it should be a sigmoid function and a multiplier. Block 3 is for stochastic gradient descent, consisting of two multipliers and one adder. The inner product size in Stage 1 is parameterized. Therefore, the input bandwidth can be parameterized. The computation proceeds iteratively by processing data through this pipeline.

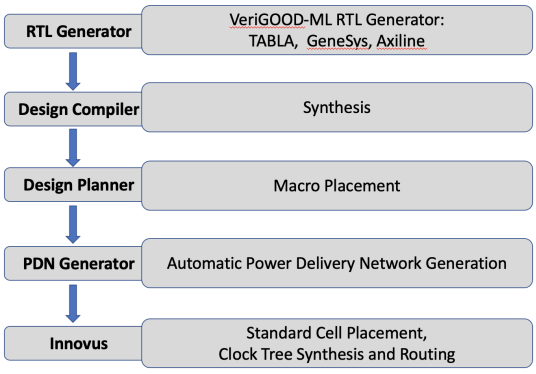


Figure 8: An overview of the SPR flow from RTL to GDSII.

IV. Synthesizing Hardware

The VeriGOOD-ML compiler takes an ML algorithm from an ONNX-level description to Verilog RTL. The next step in synthesis is to go from Verilog to GDSII. The key to successful back-end implementation of machine learning algorithms to advanced-node silicon, particularly with automatically generated RTL, is design planning. ML accelerators are inherently very structured, and optimal silicon implementation requires a design flow to leverage that structure to create a high-quality floorplan. This is a critical first step that is essential both for physical synthesis and place-and-route. A suboptimal floorplan can result in poor PPA and increased turnaround time for design closure.

Historically, design planning has been initially performed by the front-end designer, who understands the RTL design hierarchy and connectivity, and further refined by the back-end engineer, who understands the floorplan effects and utilizes constraints from the SoC regarding block outline and pin positions. As design complexity increases, this becomes practically impossible; moreover, for auto-generated RTL, there is no front-end designer who understands the design. Hence there is a critical need for an automated design planning tool that is compatible with commercial EDA tools.

VeriGOOD-ML uses a design planning flow and key engines that have been implemented in the open-source OpenROAD tools [7]–[9] so as to bridge generated RTL Verilog to successful physical implementation outcomes. In our flow, we pass the result of design planning to a place-and-route flow using commercial tools; in future, a fully OpenROAD-based flow will be targeted. The overall synthesis, place and route (SPR) flow is shown in Fig. 8.

Our in-house design planner is designed to mimic the way expert chip designers perform floorplanning. A significant challenge is related to the fact that these designs are dominated by macros that correspond to memory modules that implement various on-chip buffers. This adds complexity to the tasks of floorplanning, which must leverage design regularity, and power delivery network (PDN) generation, which must handle PDN blockages in several metal layers at the macro locations.

The design planner first creates an efficient abstraction model of the netlist by analyzing attributes such as the logical hierarchy, data flow, connections between macros and input-output pins, and timing-critical paths. The planner then uses the abstraction model to guide the generation of the floorplan. This model helps back-end engineers to gain better insights into the design and therefore reduces the number of iterations required to make the design flow converge. The following four engines are invoked sequentially:

(1) The *auto-clustering engine* converts the gate-level netlist representation of the design into a clustered netlist, in which nodes are clusters and nets are bundled connections between clusters. To generate this clustered netlist, we first create clusters based on logical hierarchy and then group small clusters based on connection signatures. To handle macro regularity, we group macros with different sizes into different hard macro clusters. We then add virtual connections between hard macro clusters and input/output IOs based on dataflow and latency.

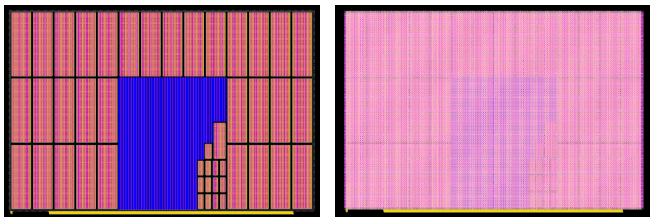
(2) The *shape engine* determines possible aspect ratios and area for each cluster based on the floorplan size and target utilization. For each hard macro cluster, we enumerate all minimum-area packings.

(3) The *macro placement engine* places all the clusters and finalizes the shape of each cluster. We use a sequence-pair representation of clusters in the (clustered) netlist, and simulated annealing to optimize the cost function. The cost function includes area, wirelength, and several penalty function terms, e.g., for overflowing the given layout region (fixed-outline constraint), or for notches or blocked pin accesses in the macro placement.

(4) Finally, the *pin alignment engine* determines the location and orientation of each individual macro. In this phase, we pack macros within each hard macro cluster, again using simulated annealing of a sequence-pair representation.

(a) (b) (c)

Figure 9: Back-end design of a GeneSys engine showing (a) signal flow on primary interconnects, (b) the floorplan, (c) layout after SPR.



(a) (b)

Figure 10: The PDN on (a) layers M1–M7, and (b) layers M1–M13.

We implement our designs based on the GF12LP technology using 13 metal layers. An Arm memory compiler is used to build dual-port register files. For each logical memory size (address and bit width), the configuration that yields the smallest area is chosen. Fig. 9 shows the data flow, the floorplan from our design planner, and the final place-and-route on a commercial back-end for the GeneSys SIMD example. The automatically generated PDN is illustrated in Fig. 10.

Using this back-end implementation flow, we are currently in the process of taping out a chip that implements a GeneSys engine. Aside from core GeneSys components, the design includes an on-chip global buffer that interacts with the external off-chip memory, as well as mixed-signal circuits such as VCOs, synthesized using ALIGN [10], [11].

V. Results

We have applied the VeriGOOD-ML flow to perform training and inference on a variety of ML algorithms, exploring the space of design configurations to optimize application-level performance metrics. For a variety of design configurations of a specific platform (TABLA or GeneSys), we generate the Pareto-optimal PPA curves for the hardware engine using our back-end implementation methodology. This yields the power and frequency characteristics of the platform. For power analysis, the activity factors at all inputs are assumed to be 0.1. Using the simulator, we track the performance of the ML algorithm on the platform, e.g., the number of cycles required to perform the computation and the memory access patterns that dictate stalls and power dissipation. Based on this, we determine the power and execution time of the ML algorithm on the platform. For example, for DNN execution on GeneSys, we combine the performance statistics provided by the simulator with the power-performance characteristics (i.e., energy per operation, clock frequency, dynamic and leakage power of various hardware components) of Pareto-optimal PPA design points provided by our backend Synthesis Place-and-Route flow to compute the energy consumption, power (both on-chip and off-chip), and runtime. For Axiline, the mapping is performed directly to report the power and execution time. In this section, we provide a snapshot of a set of results obtained from exercising VeriGOOD-ML. A variety of design implementations have been built, up to post-SPR; a sample set is shown in Fig. 11. These implementations create a Pareto-optimal set of designs that form the basis for the results shown below.

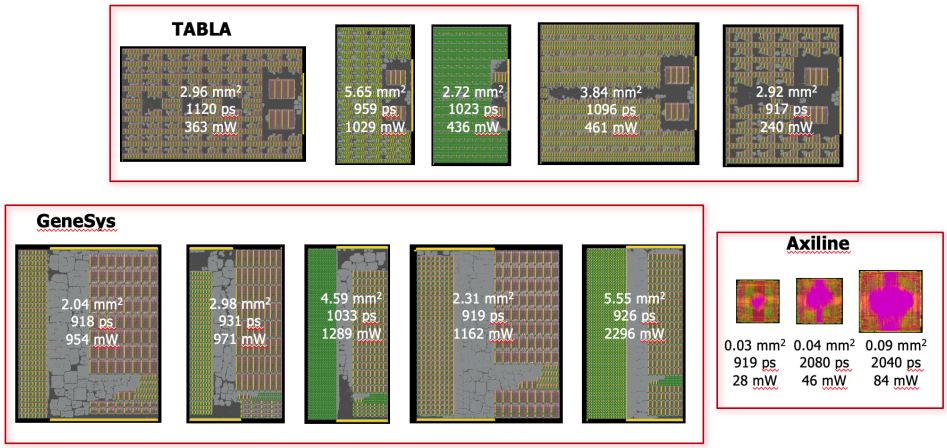
#PUs	#PEs/PU	Frequency	Area	Power	Training Runtime	Inference Runtime
8	8	1GHz	2.96mm^2	1.28W	30.6min	0.21ms
8	8	0.25GHz	2.96mm^2	0.29W	122.3min	0.85ms
8	16	1GHz	5.65mm^2	1.90W	26.3min	0.17ms
8	16	0.25GHz	5.65mm^2	0.56W	105.1min	0.68ms

Table I: Training and inference results for the SVM on various TABLA configurations.

Classification and localization problem using SVM on TABLA: We exercise an SVM on the WLAN Indoor Localization benchmark [12] dataset. Data preparation consists of the following steps. We first import the WiFi RSSI dataset, the smartphone geomagnetic dataset, the timestamp datafile, and the PointsMapping dataset that contains the placeID-to-XY coordinate mapping. Next, we merge the RSSI dataset with PointsMapping dataset by PlaceID, so that we have XY coordinate and placeID data for RSSI measurements. Finally, we merge the RSSI dataset and Smartphone Geomagnetic dataset together according to the timestamp datafile. The final preprocessed dataset after these operations consists of a table with 11,498 rows and 143 columns that contains all the relevant feature data.

Next, we implement both training and inference for the SVM algorithm in the PolyMath domain-specific language and compile it to the *sr*-DFG representation, followed by a TABLA-backend translation pass, which produces the binary executable as well as necessary configuration and RTL files for TABLA. We consider multiple design implementations of the TABLA platform, and report a set of Pareto-optimal points in Table I.

ResNet50 on GeneSys: We implement ResNet50 on multiple instantiations of GeneSys, each with a different configuration, corresponding to a different size for the PE and SIMD arrays, and different bitwidths. The results for these configurations for single-stream inference, where a query is sent after a previous query is complete, are summarized in Table II. The designs correspond to different Pareto-optimal points, e.g., a design that is optimized for area; a slower design at a low power point; a higher-bitwidth design optimized for classification accuracy; and the largest design that is optimized for speed. The memory interface is assumed to connect to



(a)

Figure 11: Configurations at multiple PPA points for TABLA, GeneSys, and Axiline with post-SPR layout (on-chip power only reported).

PE array size	Bitwidth	Frequency	Area	Power	Execution Time*
16×16	4	1.09GHz	2.0mm^2	0.44W	25.6s
16×16	4	0.27GHz	3.0mm^2	0.10W	89.1s
32×32	8	1.04GHz	8.5mm^2	1.04W	10.0s
64×64	4	0.97GHz	18.9mm^2	1.31W	6.9s

(*reported for 1024 single-stream inference)

Table II: Inference results for ResNet50 on GeneSys.

Benchmark	# Features	Frequency	Area	Execution time	On-chip power	Total power
Logistic regression	54	495MHz	0.024mm^2	4.70ms	24mW	0.47W
		500MHz	0.014mm^2	6.98ms	13mW	0.31W
SVM	200	500MHz	0.042mm^2	6.01ms	46mW	3.42W
		497MHz	0.030mm^2	10.05ms	27mW	2.04W
Linear regression	784	492MHz	0.091mm^2	0.37ms	84mW	4.45W

Table III: Training results for non-DNN benchmarks on Axiline.

an external HBM2 memory.

Axiline results: Table III shows the result of implementing Axiline for a training on a set of non-DNN benchmarks. For the logistic regression and SVM benchmarks, two different design points are shown. In all cases, the execution times (which exclude memory fetch times) for Axiline, area, and on-chip power are smaller than those for a platform-based method due to the custom-optimized nature of the engine. The total power is dominated by the off-chip power: in this case, we also assume an HBM2 external memory interface.

VI. Conclusion

In this paper, we have presented the VeriGOOD-ML flow for automated ML hardware synthesis. The ONNX representation of an ML algorithm is represented as an IR in the form of a *sr*-DFG, which is then translated to one of the three VeriGOOD-ML engines. Based on the HAD that represents the architecture configuration, the flow translates the IR to an implementation on TABLA (for non-DNN algorithms) or GeneSys (for DNNs), including code generation for the ISA for the corresponding platform. The translation to Axiline is performed directly from the *sr*-DFG. The design then goes through back-end synthesis. Results on a variety of ML algorithms illustrate the efficacy of the flow at multiple Pareto points.

Acknowledgments and Disclaimers

The authors would like to acknowledge the contributions of Pichet (Louii) Chaiyakul, Sayak Kundu, and Nikhil Dakwala.

This material is based on research sponsored in part by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-20-2-7009. The U. S. government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFRL, DARPA, or the U. S. government.

References

[1] D. Kehlet, "Accelerating innovation through a standard chiplet interface: The advanced interface bus (aib)." https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/accelerating-innovation-through-aib-whitepaper.pdf.

[2] "VeriGOOD-ML: Verilog generator, optimized for designs for machine learning." https://github.com/VeriGOOD-ML/public.

[3] S. Kinzer, *et al.*, "A Computational Stack for Cross-Domain Acceleration," in *Proc. HPCA*, 2021.

[4] D. Mahajan, *et al.*, "TABLA: A unified template-based framework for accelerating statistical machine learning," in *Proc. HPCA*, pp. 14–26, March 2016.

[5] H. Sharma, *et al.*, "From high-level deep neural models to FPGAs," in *Proc. MICRO*, Oct. 2016.

[6] H. Sharma, *et al.*, "DnnWeaver v2.0: From tensors to FPGAs," in *Proc. Hot Chips*, Oct. 2016.

[7] T. Ajayi, *et al.*, "Toward an open-source digital flow: First learnings from the openroad project," in *Proc. DAC*, 2019.

[8] "The OpenROAD project." github.com/The-OpenROAD-Project.

[9] A. B. Kahng and T. Spyrou, "The OpenROAD project: Unleashing hardware innovation," in *Proc. GOMAC*, 2021.

[10] K. Kunal, *et al.*, "ALIGN: Open-source analog layout automation from the ground up," in *Proc. DAC*, pp. 77–80, 2019.

[11] T. Dhar, *et al.*, "ALIGN: A system for automating analog layout," *IEEE Des. Test*, vol. 38, pp. 8–18, Apr. 2021.

[12] "Geo-magnetic field and WLAN dataset for indoor localisation from wristband and smartphone data set." http://archive.ics.uci.edu/ml/datasets/Geo-Magnetic+field+and+WLAN+dataset+for+indoor+localisation+from+wristband+and+smartphone.