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

Hadi Esmaeilzadeh1, Soroush Ghodrati1, Jie Gu2, Shiyu Guo2, Andrew B. Kahng1, Joon Kyung Kim1, Sean Kinzer1, Rohan Mahapatra1, Susmita Dey Manasi3, Edwin Mascarenhas1, Sachin S. Sapatnekar2, Ravi Varadarajan3, Zhiang Wang4, Hanyang Xu5, Brahmandra Reddy Yatham5, Ziqing Zeng6
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 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. ONNX represents ML algorithms as a standardized graph to facilitate interoperability across various development 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 “Codelet” 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 that provide PPA tradeoffs that generate Verilog with physical implementation considerations.

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...
Modeling hardware using a HAG: We model the structure of specific accelerator platforms by introducing a reusable hardware abstraction called a hierarchical architecture graph (HAG), 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 HAG for a specific target accelerator for mapping, scheduling, and optimizing programs on the accelerator. Each HAG is comprised of three types of nodes: for computations, for on- and off-chip communication, and for storage. In interaction with the sr-DFG, the HAG 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 HAG. Such an abstraction enables the compiler to expand its capability from optimizing for single piece of hardware to a heterogeneous computing environment where there are 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, there are several primary attributes that must be included in the abstraction: the ability to (a) model hierarchy (as fine-grained as a single ALU, or as coarse-grained as an entire systolic array); (b) specify compute, storage, and communication components; and (c) annotate 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. Note that the architecture description is primarily intended for compilation purposes, and captures design information at a high level, eschewing a more detailed gate-level description.

Compilation to target accelerators: Having devised an abstract representation of different types of accelerator architectures, a multi-stage compilation process can be reused across different HAGs. The stages of compilation, illustrated in Fig. 2 consist of:

- Operation mapping/scheduling: An sr-DFG is ordered to a sequence of operations, and each operation will be mapped to a particular component in the HAG according to the sr-DFG node operation and the sequence of capabilities that produce the equivalent operation. In addition, sequences of operations can be fused together according to user-supplied parameters.
- Compilation optimization: A search for optimal compilation parameters is performed using specifications of the HAG, 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 HAG: This step is based on the instantiated capabilities from the two previous steps. The compiler combines code templates called Codelets with the sr-DFG node attributes and HAG attributes. Codelets 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 Codelets: (i) Compute Codelets that represent instructions for performing computations on data; (ii) Memory Codelets for instructions that move data from one memory location to another (e.g., load from/store to off-chip or on-chip memory); (iii) Loop Codelets that repeat operations over a number of loop levels; and (iv) Control Codelets for instructions that determine program flow. These Codelets 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 Codelets, 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 hardcoded 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 Figure 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 busses for inter-PU communication, there are two busses for inter-PE communication. The bus...
provide the input program as an sr-DFG file and a configuration file that sets the parameters of the template architecture described in the above sections such as number of PEs per PU. 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 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 address generator modules and controller FSMs that together generate the addresses and requests to load or store a tile of data from/to off-chip memory. The address generators perform strided address pattern generation and generate addresses in the off-chip memory and 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 interface for OBUFF and SIMD array handles 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.

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.

Figure 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 and the size of the weight scratchpad, precision of the input, weight, partial sum and also the bitwidth of the multiply-accumulate logic in addition to the registers are parameterizable during architectural synthesis, according to the demands of the application.

For address generation, we design a memory walker module that can automatically generate the addresses for executing 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 a set of parameters such as the number of loop iterations and the base address in the memory, and can then generate addresses automatically as:

\[
\text{address} = \text{base_address} + \text{loop_iter} \times \text{stride}
\]

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 vectory 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
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. A critical first step in 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 initially been 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],[8] 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, the connection 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. Four engines that 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 macros in a way that maximizes performance, we group macros that correspond to memory modules and use different sizes into different hard macro clusters. We then add virtual connections between those hard macro clusters and input/output IOs based on dataflow and latency. For each hard macro cluster, we enumerate all possible minimum-area packings.

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

3. The macro placement engine places all the clusters and finalizes the shape of each cluster. In this phase, we use a sequence-pair representation of clusters in the netlist, and simulated annealing to optimize the cost function. The cost function includes area,
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. Using the cycle-accurate 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.

Classification and localization problem using SVM on TABLA:
We exercise an SVM on the WLAN Indoor Localization benchmark [10] 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-
### TABLE II: Inference results for ResNet50 on GeneSys.

<table>
<thead>
<tr>
<th>PE array size</th>
<th>Bitwidth</th>
<th>Frequency</th>
<th>Area</th>
<th>Power</th>
<th>Execution time*</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 x 16</td>
<td>4</td>
<td>1.09GHz</td>
<td>2.0mm²</td>
<td>0.44W</td>
<td>25.6s</td>
</tr>
<tr>
<td>16 x 16</td>
<td>4</td>
<td>0.27GHz</td>
<td>3.0mm²</td>
<td>0.10W</td>
<td>89.1s</td>
</tr>
<tr>
<td>32 x 32</td>
<td>8</td>
<td>1.04GHz</td>
<td>8.5mm²</td>
<td>1.04W</td>
<td>10.0s</td>
</tr>
<tr>
<td>64 x 64</td>
<td>4</td>
<td>0.97GHz</td>
<td>18.9mm²</td>
<td>1.31W</td>
<td>6.9s</td>
</tr>
</tbody>
</table>

(*reported for 1024 single-stream inference)

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># Features</th>
<th>Frequency</th>
<th>Area</th>
<th>Execution time</th>
<th>On-chip power</th>
<th>Total power</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logistic regression</td>
<td>54</td>
<td>495MHz</td>
<td>0.02mm²</td>
<td>4.70ms</td>
<td>24mW</td>
<td>0.47W</td>
</tr>
<tr>
<td>SVM</td>
<td>200</td>
<td>500MHz</td>
<td>0.04mm²</td>
<td>6.01ms</td>
<td>46mW</td>
<td>3.42W</td>
</tr>
<tr>
<td>Linear regression</td>
<td>784</td>
<td>492MHz</td>
<td>0.09mm²</td>
<td>0.37ms</td>
<td>84mW</td>
<td>4.43W</td>
</tr>
</tbody>
</table>

### ACKNOWLEDGMENTS

This work was supported in part by AFRL under the DARPA RTML program under award FA8650-20-2-7009. The authors would like to acknowledge the contributions of Pichet (Louii) Chaiyakul, Sayak Kundu, and Nikhil Dakwala.

### REFERENCES


