2023 Design, Automation & Test in Europe Conference (DATE 2023)

Energy-efficient Hardware Acceleration of Shallow Machine Learning Applications

Ziqing Zeng
University of Minnesota
Minneapolis, MN, USA

Sachin S. Sapatnekar
University of Minnesota
Minneapolis, MN, USA

***Abstract*—ML accelerators have largely focused on building general platforms for deep neural networks (DNNs), but less so on shallow machine learning (SML) algorithms. This paper proposes Axiline, a compact, configurable, template-based generator for SML hardware acceleration. Axiline identifies computational kernels as templates that are common to these algorithms and builds a pipelined accelerator for efficient execution. The dataflow graphs of individual ML instances, with different data dimensions, are mapped to the pipeline stages and then optimized by customized algorithms. The approach generates energy-efficient hardware for training and inference of various ML algorithms, as demonstrated with post-layout FPGA and ASIC results.**

I. Introduction

Rapid advances in machine learning (ML) have motivated extensive deployment of ML algorithms of two types:
(1) *Deep ML algorithms* implemented as deep neural networks (DNNs), e.g., ResNet, VGG-16, Mask R-CNN, using many convolution layers, plus operations such as pooling/activation. Their execution involves high runtime, energy, and power.
(2) *Shallow ML (SML) algorithms* are classical ML approaches, e.g., support vector machines (SVMs), logistic regression, backpropagation, k-nearest neighbors (KNNs), and decision trees. Unlike DNNs, they do not use neural network techniques, but are based on statistical methods.

In recent years, DNNs succeeded in solving a number of previously intractable problems [1]–[3], and have brought ML-based computation into the mainstream of computer architecture. Future systems that integrate distributed sensing, computation, and actuation can now realistically be built using ML-based techniques that can provide new computational capabilities that were impossible in prior systems. Designing these hardware systems requires extreme energy efficiency, particularly in the context of distributed systems. Using DNNs everywhere within a system is not realistic: their implementation on standard hardware platforms, such as CPUs, GPUs, FPGAs, or even on customized DNN-specific hardware accelerators, is not ultra-energy-efficient. In this context, SML can complement DNNs through energy-efficient execution with acceptable accuracy, especially in edge computing.

SML can provide high accuracy, appropriate to application needs, e.g., SVMs for Wi-fi localization [4], [5], pest detection [6], and cancer genomics [7]; logistic regression for relational database management [8]; and decision trees for detecting ionospheric scintillations [9] and screen content coding [10]. Some SMLs have been proven to have higher maximum accuracy than NN in many data sets [11], [12]. Improvements in execution time and energy efficiency are possible on custom hardware accelerators [13], [14]. Since SML algorithms are more heterogeneous in structure than DNNs, with different computational structures, most prior hardware acceleration methods have built dedicated hardware for each SML algorithm [15]–[19].

This paper proposes Axiline, a platform that automatically builds hardware for accelerating SML. Based on a representation of target SML algorithms using dataflow graphs (DFGs), we observe similarities in the dataflows for multiple SML algorithms. Recognizing similar computational structures in classes of SML algorithms, we motivate the use of *templates* for ML hardware acceleration, where each template provides an efficient hardware substructure for the class. We consider a library of templates for the most common computations, and for each target SML algorithm, we map its DFG to the templates. Axiline automatically synthesizes the register-transfer level (RTL) description of hardware accelerators, for either training or inference, for a class of SML algorithms. Using this approach, we can create a parameterized template-based design with various numbers of computational elements, bit widths, etc., to customize the hardware to target SML algorithms of specific sizes and power/energy specifications.

This work bridges the gap between general accelerators and algorithm-specific accelerators for SML. We gain computational efficiency through a template-based framework, which is general enough to address multiple SML algorithms, but creates custom hardware targeted to a specific algorithm, scoring highly on accelerator metrics (hardware power, performance, and area (*PPA*), and the *runtime* and *energy* for hardware execution of the ML workload), while also enhancing designer productivity through template reuse.

II. Related Work

SML accelerators can be classified as follows:
General Purpose Processors. SVM algorithms can be implemented on CPUs and GPUs, facilitated by ML frameworks such as PyTorch, TensorFlow, and MXNet. An example is the RISC-V based PULP [20] used to accelerate SML in [21].
Algorithm-specific Architectures. This class corresponds to dedicated architectures that are optimized for one specific algorithm to achieve efficiency in performance, utilization, and energy. These platforms sacrifice flexibility and generality to

achieve performance gains for the targeted algorithm. Many existing shallow ML accelerators [13], [15], [16], [22] fall into this category and are focused on a specific inference algorithm. These approaches [15] are reported to provide 2–10× lower LUT utilization than a more general approach such as [14].

Template-based Designs. These accelerators emplys a parameterizable number of hardware units. An example is TABLA [14], which uses a uniform array of processing elements and uses *software* templates to map standard building blocks onto this uniform hardware. Although hardware templates have been used in the HLS community [23], [24], they have seen limited use for building ML hardware architectures.

Our proposed Axiline platform differs significantly from all of these. Unlike general-purpose processors or algorithm-specific architectures, we build hardware that is specific to a class of SML algorithms; unlike TABLA's software templates or SODA's [25], [26] use of existing IP, we employ parameterizable *hardware* templates to build parameterizable platforms for entire classes of SML algorithms.

TABLE I
TARGET FUNCTIONS AND OBJECTIVE FUNCTIONS GRADIENTS FOR SEVERAL SHALLOW ML ALGORITHMS.

SML algorithm	**Inference Target Function**	**Gradient of the Training Objective Functions** ($\partial f/\partial W^{(t)}$)
Linear Regression	$W^T \cdot X_i$	$(W^T \cdot X_i - Y_i) \cdot X_i$
Logistic Regression	$\sigma(W^T \cdot X_i)$	$(\sigma(W^T \cdot X_i) - Y_i) \cdot X_i$
SVM	$\theta(W^T \cdot X_i)$	$(\theta(W^T \cdot X_i) - Y_i)X_i$
Recommender Systems	$\sum_i W_j^T \cdot X_i$, $\sum_j W_j^T \cdot X_i$	$\sum_i (W_j^T \cdot X_i - Y_{ij})X_i$, $\sum_j (W_j^T \cdot X_i - Y_{ij})X_j$
Backpropagation	$\sigma(W_2^T \cdot \sigma(W_1^T \cdot X_i))$	$(\sigma(W_2^T \cdot \sigma(W_1^T \cdot X_i)) - Y_i)W_2^T (\sigma(W_1^T \cdot X_i))(\sigma(W_1^T \cdot X_i) - 1)X_i$, $(\sigma(W_2^T \cdot \sigma(W_1^T \cdot X_i)) - Y_i)\sigma(W_1^T \cdot X_i)$

SML algorithm	**Inference Target Function**
Decision Tree	At each node n_i: $f_i(x) = W_i^T \cdot X + w_{i0} > 0$. Next node: For a path $n_0 \rightarrow \cdots \rightarrow n_L$, $n_{i+1} = LUT_{n_i}(f_{n_i})$. Prediction: $Y = LUT_{n_L}(f_{n_L}) = LUT(f_{n_1}, f_{n_2}, ...f_{n_L})$.
GBDT	For each tree: $Y_i = LUT_{i,L}(f_{i,L})$. GBDT: $Y = \sum_{i=1}^N Y_i$
Random forest	For each tree: $Y_i = LUT_{i,L}(f_{i,L})$. Forest: $Y = (1/N)\sum_{i=1}^N Y_i$ or $Y = \text{Maj}_{i=1}^N Y_i$.

III. Computational Structure of SML algorithms

The execution of an SML algorithm involves the evaluation of a specific target function. Performing *inference* involves the implementation of the target function, while training implements an optimization process that determines the weights associated with the target function. The *training* computation is driven by stochastic gradient descent (SGD), an iterative optimization algorithm, to find the set of weights that minimize an objective function. Next, we present the core operations for several sets of SML algorithms, summarized in Table I.

Classical SML algorithms Table I shows the operations for training/inference for linear regression, logistic regression, SVMs, recommender systems, and backpropagation.

Inference: The core operation for these algorithms is the inner product (IP) $W^T \cdot X_i$, and may be followed by a nonlinear transformation, such as the sigmoid (σ) or step function (θ).

Training: A key function is stochastic gradient descent (SGD), which divides the objective function into smaller functions of one input vector. The gradient of the smaller function over this vector [14] provides a weight update in iteration t:

$$W^{(t+1)} = W^{(t)} - \mu \cdot \left[\partial f(W^{(t)}, X_i)) / \partial W^{(t)} \right], \qquad (1)$$

where the gradient of the training objective function ($\partial f/\partial W^{(t)}$) is based on a (nonlinear) function of IP ($W^{(t)T} \cdot X_i$), minus the bias vector (Y_i) and multiplied with X_i (in Table I); t is the iteration index, indicating the processing of the t^{th} input vector set; and μ is the learning rate.

Decision tree: The next category in Table I is the class of decision tree methods, including basic decision tree, gradient boosting decision tree (GBDT), and random forests. The table shows the inference target functions for these methods. Each node i of the decision tree includes a comparison function, f_i, that checks whether the sum of an IP ($W^T \cdot X$) of weight and multiple features, plus a constant (w_{i0}), is positive. Each decision tree can be transferred into a single-variate decision tree, where $W^T \cdot X$ is replaced with X_k, where k is a node index variable. The path for inference processes a group of nodes n_i. For an intermediate node of the decision tree, the next node n_{i+1} depends on the result of the comparison function f_{n_i} and is generated using a look-up table (LUT); for any leaf nodes, the LUT generates the final prediction, i.e., the final prediction is based on a LUT of all f_n in a path.

The GBDT and random forest algorithms process multiple decision trees: the final prediction of a GBDT is the summation of all decision tree predictions, while random forest computes an average or majority among the decision trees.

IV. Hardware Template Library

We introduce a library of hardware templates common to several algorithms in Table I, e.g., inner product (IP), SGD, and comparison units. These templates are specific to the SML algorithms in Table I, but the principles could be extended to optimally implement a wider range of SML algorithms. Note that the goal of the templates is to efficiently implement operations: the absence of a dedicated template does not mean that an SML algorithm cannot be implemented (because it can always be implemented using basic arithmetic operations): templates merely enable more efficient implementations.

Inner product templates: The basic IP unit, illustrated in Fig. 1, has parameterized hardware blocks that implement accumulated inner product computations in each cycle. The IP unit is parameterized by the dimension of the inner product and the bitwidth. Based on data stationarity pattern, there are two types of IP templates: input-stationary (IS) or output-stationary (OS). The difference is IS IP unit must load and store the partial inner product result, and requires a SRAM or RF, which has one write and one read port. The OS IP template has higher energy efficiency than IS IP template with the same parameters. The reason for two types of IP templates is that we can map two serial IP computations to an OS IP followed by a IS IP with the same dimension to enhance energy efficiency. As another optimization, multiple IPs could be used in parallel.

Constant comparison template: This unit compares one input with a constant and generates a binary output, depending on which is greater, and is parameterizable by the number of bits.
LUT template: For some computations, a lookup table (LUT)-based implementation can be more energy efficient than arithmetic units, e.g., a sigmoid with fixed-point inputs is often implemented as a LUT that is populated based on the bitwdith and the radix point location. This unit is parameterized by the number of bits, and the number of LUT entries.
SGD template: The SGD unit supports the SGD computation in training, as represented by Eq. (1). The SGD unit implements two slice multiplies with X_i and learning rate, and one subtraction from weight W. This unit is parameterizable by the number of bits for X_i, W and the learning rate.
Decision tree comparison template: The decision tree template has a set of comparison units and is parameterized with the number of units, bitwidth, and depth of local SRAM or register file (RF), as shown in Fig. 2. The comparison unit can access data from local memory, implement an IP operation and compare it with input data to generate a one-bit output for a decision tree, as shown in Table I. We can use parallel units to speed up the computation, as will be illustrated in Section VI.
Multi-functional template: We can combine the functionality of multiple operations to build a multi-functional template (e.g., combining arithmetic operations, sigmoids, comparators), with a control signal used to select the function to be used. The unit is parameterizable by the types and number of functions, bitwidths, and all parameters for involved functions. In Section V, we will show how SML can use these units.

V. Template-Based Design and Optimization

Template-Based Design Workflow. The SML algorithm, specified using ONNX, is converted to an intermediate representation, the *sr*-DFG, generated using the Polymath compiler [27]. The *sr*-DFG is a recursive representation of operations that allows simultaneous access to multiple levels of operation granularity. For example, the *sr*-DFG node could represent an operation such as matrix multiply or slice multiply at one level of granularity, which maps to a finer-grained DFG with operations at the level of multiplies/adds.

The workflow used to develop Axiline is shown in Fig. 3. We begin by initiating a DFG traversal to map the nodes to templates and basic arithmetic units. First, the compiler tries to match the *high-level* nodes in *sr*-DFG (e.g., slice_mul, mat_mul, sum) to the templates in our library (shown as "Template match" in the figure). With an appropriate template

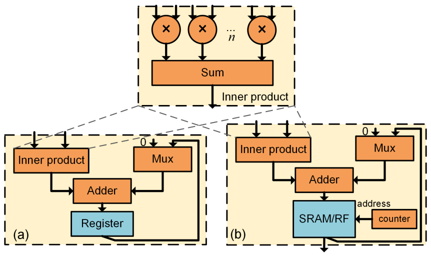

Fig. 1. Hardware templates for performing inner product (IP) computations: (a) OS IP template (b) IS IP template.

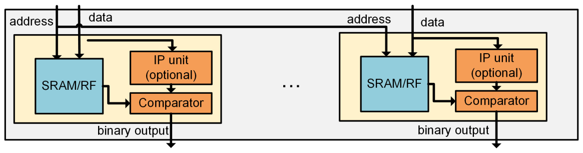

Fig. 2. Decision tree comparison template.

library, it is expected that the most complex functions will map onto an element of the library; the user has the opportunity to extend the library to add new functions if needed. Not all nodes in the *sr*-DFG will map to a template: the nodes that are not matched are passed to low-level mapping ("Direct transfer"). The compiler then uses the *sr*-DFG at the *lowest level*, where each node is an arithmetic operation with up to two inputs and one output (e.g., add, multiply, compare). The unmapped low-level nodes are directly mapped to basic arithmetic blocks. The traversal continues until all nodes in the *sr*-DFG are mapped. The time complexity of mapping is $O(N + E)$, where N and E are the maximum numbers of nodes and edges over all levels of granularity in the *sr*-DFG.

Because the structure of the *sr*-DFG is not always optimal for hardware implementation, the compiler initiates a post-mapping traversal to perform some optimizations ("Post-mapping optimization" in Fig. 3), which consists of matching and replacement for the templates. The details of these optimizations are provided in the later part of this Section. Finally, the resulting optimized design is taken through synthesis, placement, and route (SP&R) to determine the PPA, and the energy consumption for running the SML algorithm is determined, based on the total power (from the hardware PPA) and the number of cycles (from a system-level analysis).
Flexibility. Different from HLS tools that generate specific RTL for each algorithm, the Axiline template library involves multi-functional templates that cover more than one computation. Therefore, Axiline can map a class of SML algorithms with similar dataflow to the same hardware. In other words, our platform generates an algorithm-specific accelerator, but also can trade off energy efficiency for flexibility by building platforms that support multiple SML algorithms with similar dataflow. As an example, Fig. 4 shows the DFGs for training and inference for four SML algorithms – linear regression, logistic regression, SVM, and recommender system. Part of the dataflow for IP (for training and inference) and SGD (for training) is color-coded in yellow and blue, respectively, in Fig. 4: it is clear that these segments are the same among

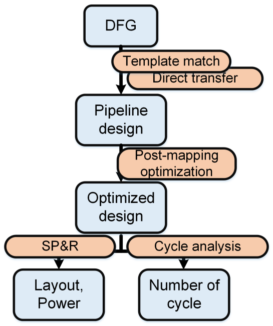

Fig. 3. Workflow for the Axiline generator.

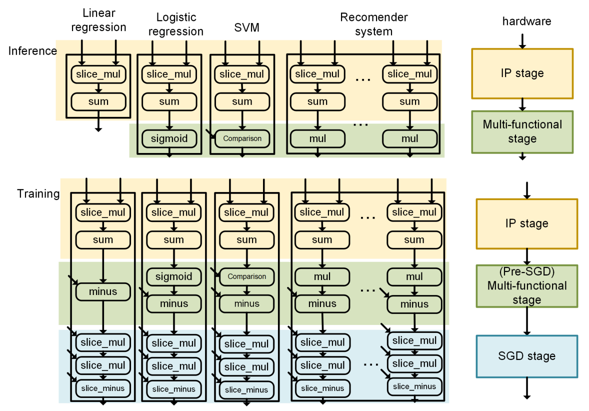

Fig. 4. Example of generating one accelerator for multiple SML algorithms.

these four algorithms. If we define a multi-function template that covers the computation for their distinct operations (in green in Fig. 4), then all four of these SML algorithms can be mapped into the same hardware. This is shown on the right of the figure: the upper half of the figure uses an IP stage and a multi-functional stage for inference; the lower half uses an IP stage, a multi-functional stage, and an SGD stage for training.

Optimization rules. To support a group of SML algorithms, the optimization must handle both their similarities and discrepancies. We create optimization rules with a list of pattern matching and replacement methods, implemented as functions in a traversal through the graph of pipeline stages. We define a set of original patterns and optimized patterns. If the pattern of nodes in traversal matches an original pattern in the optimization rule, we can replace them with an optimized pattern that is functionally equivalent but more efficient. For example, any *basic computations* – add, subtract, multiply – connected to an SGD template are replaced by optimized blocks (pre-SGD stage, parallel SGD stage), as shown in Fig. 5. Optimizations in this example include switching the order of slice multiplies to reduce computations (multiplication by μ in the optimized implementation to the right appears before multiplication by X_{ij}), and delay-balancing between the two pipeline stages.

Other optimization rules include: (1) A LUT and constant comparison block can be merged into a following pipeline stage. (2) Two or more basic computations can be combined into one pipeline stage. (3) For a training algorithm, the dimension of the SGD unit must be the same as the IP unit that accepts its output. Because the space of SML algorithms is much narrower than the wide scope of HLS, which covers a much wider range of architectures, it is possible to compactly define these rules for SML algorithms. The worst-case time complexity for optimization is $O(R(N'+E'))$, where N' and E' are the numbers of nodes and edges in the implementation graph, and R is the number of defined rules.

VI. Example Hardware Implementations

Logistic regression. Fig. 6 shows an example of transferring an *sr*-DFG of logistic regression training into pipeline stages. Fig. 6(a) shows the coarse-grained *sr*-DFG, where the nodes “slice_mul”, “sum”, “minus”, and “sigmoid” correspond to slice multiplication, summation, subtraction, and the sigmoid

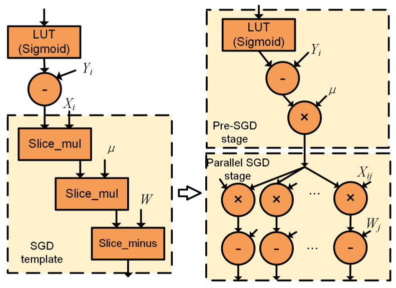

Fig. 5. An example of optimization rules for an SGD instance in the *sr*-DFG.

function, respectively. In the first traversal, W, X, Y, and μ are mapped to memory transfer operations to bring data on-chip; the adjacent “slice_mul” and “sum” nodes are recognized as an IP operation and mapped into an IP template, two “slice_mul” and one “slice_minus” and recognized as an SGD operation and transferred into an SGD stage. The sigmoid function and “minus” nodes are mapped to basic templates: a LUT and an elemental subtractor, creating a four-stage pipeline.

After generating the hardware graph in Fig. 6(b), the compiler starts the optimization traversal with the original four-stage pipelined architecture. Using the rules in Section V, the sigmoid stage is merged with the minus stage and delay-balanced with the SGD stage. The architecture is then transformed into a three-stage pipeline in a post-mapping optimization (Fig. 6(c)). Any one-dimensional basic operations before the SGD stage are balanced with the SGD stage to generate a pre-SGD stage and an optimized SGD stage. The optimization includes dimension-matching: the parallel SGD output must have the same dimension as the IP in the next pipeline stage, where it sends data for the next training iteration.

Decision tree. Fig. 7 shows the implementation of hardware for inference on a decision tree. The dimension of the decision tree comparison units in each pipeline stage is user-defined by the user: here, we choose 2 units for the first stage and 3 for the second. In the mapping process, we group the nodes in the graph based on the dimension of each stage. We map nodes (1,2) to stage 1, with 2 units. Supernode (1,2) has 3 branches with the root nodes (4, 5, 3). Because stage 2 has 3 units, (4, 5, 3), (8, 6) and (9, 7) are mapped to each of these units; if the nodes exceed the limit of defined stages, more units or stages are needed. The memory size of each comparison unit depends on the number of mapped nodes. Depending on the result of stage 1, the local memory address for stage 2 indicates the branch selected in the grouped node. A LUT-based combinational logic is generated to transfer the

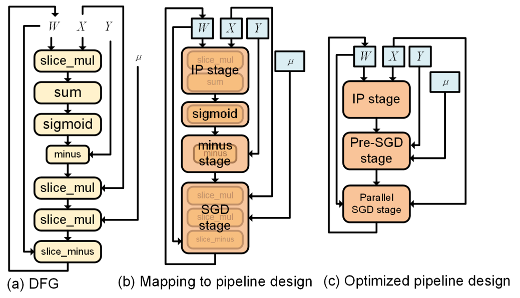

Fig. 6. Using our approach for to build hardware for logistic regression.

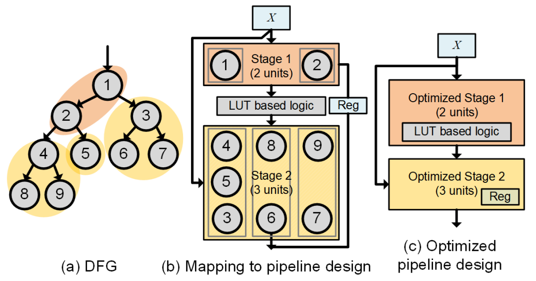
X
Stage 1 (2 units)
LUT based logic
Reg
Stage 2 (3 units)
Optimized Stage 1 (2 units)
LUT based logic
Optimized Stage 2 (3 units)
Reg
(a) DFG
(b) Mapping to pipeline design
(c) Optimized pipeline design

Fig. 7. Using our approach for hardware implementation of a decision tree.

binary output of one stage into the address for the next stage. The final output is the combination of outputs for each stage (Table I). During optimization, the LUT is embedded into the parallel comparison stage, as shown in the figure.

VII. Experiments

We evaluate our architecture with on-chip energy and runtime for the SML algorithms in Table II, comparing Axiline against TABLA [14] (both as ASIC) and against HLS (both on FPGA). Axiline and HLS both custom-generate the hardware for each ML algorithm, while TABLA uses the same platform. For FPGA, we use Xilinx Vivado for an Artix-7 board in 28nm. For ASIC, we use Design Compiler+Innovus to run synthesis, placement and routing (SP&R) at 12nm. The RTL template in TABLA uses fixed point arithmetic, with input bitwidth of 16 and internal bitwidth of 32. We use the same setup for Axiline and HLS, and assume off-chip memory bandwidth loads up to 20 features/weights per cycle. The platform specifics are:
Axiline: We use Axiline directly transferring the *sr*-DFG into pipeline design and synthesize the RTL on ASIC or FPGA.
TABLA [28]: We use three configurations: 4×4, 4×8, and 8×8 (#PUs $\times$ #PEs_per_PU). We use the static scheduling algorithm in [14] to analyze the number of cycles.
HLS: We use C++ code to convey the same dataflow information as *sr*-DFG and then use the Vitis HLS platform to transfer the C++ code into RTL. We add a pipeline directive in the outer loop (among input vectors), and use the same initiation interval as Axiline to generate pipeline designs. The number of cycles is generated from Vitis.
Axiline+HLS: We combine Axiline and HLS for FPGA implementation, first using Axiline to generate architectures, then Vitis HLS to optimize the C++ description of the architectures.

We select different target clock periods (TCPs) during SP&R as timing constraints. For SP&R on the Artix FPGA, the TCPs range from 10–100ns in steps of 5ns, and we only keep the data points with slack in (−5ns, 5ns). For ASIC SP&R, we use TCP = 500ps, 1000ps, 1500ps, 2000ps, 3000ps, 4000ps.

For each benchmark, we generate the hardware PPA tradeoff curve and report the design point with the lowest energy. We report the energy consumption and runtime within the same plot by normalizing the number of input vectors to 10,000. In FPGA implementation, we report the resource usage, which is the summation of LUT, register, and DSP usage.
Comparison to TABLA. We compare the on-chip energy and runtime of accelerators generated through Axiline and TABLA, both in ASIC implementations, for all benchmarks

TABLE II
A list of evaluated SML benchmarks.

Algorithm	Number of Weights	Input Vectors	Model Topology
Linear Regression	55	10000	55
Support Vector Machine (SVM)	200	500000	200
Logistic Regression	54	581000	54
Recommender System	240	1161	10–24
Backpropogation	192	10000	8–16–4
Decision Tree	169	5620	depth=15

in Table II, with the exception of decision trees, which are not supported by TABLA. Figs. 8 show the comparisons, respectively, for training and inference hardware. Our designs achieve an average of 98% energy saving and 3.5× speedup for the inference benchmarks, and 94% energy saving and 3.1× speedup for the training benchmarks. The energy consumption of the TABLA PE array is at least an order of magnitude higher than our algorithm-specific designs for several reasons. (1) The TABLA design needs more instances (e.g., buffers, buses, memories, etc.) and complex control logic as it is a general-purpose accelerator for multiple ML algorithms, while we focus on improved system performance, energy, and hardware PPA by customizing our solution. (2) Our design maps high-level operations (e.g., IP, MatMul, SGD) onto dedicated IP units and SGD units, which is more efficient than mapping low-level operations (e.g., multiplication, addition) to arithmetic units, as in TABLA. (3) TABLA does not utilize all PEs/PUs in every cycle due to its generality; in contrast, our dedicated hardware has better utilization.
Comparison to HLS. The energy and runtime comparison between our design and HLS-generated design on the Xilinx Artix-7 FPGA platform for all benchmarks (including decision trees) in Table II is illustrated in Fig. 9. Axiline-generated accelerators save 29.7% of energy on average compared to HLS-generated accelerators, with a speedup of 1.35×. Axiline+HLS designs can save 39.2% of energy on average compared to HLS-generated designs, with a speedup of 1.38×. For training benchmarks, both Axiline-generated designs and Axiline+HLS designs achieve better energy efficiency and shorter runtime than HLS-generated designs. For inference benchmarks, especially the two small benchmarks (Linear-55 and Logistic-54) which are simpler to synthesize, HLS-generated designs can achieve a shorter runtime than Axiline designs, but Axiline-generated designs can save 23.7% of energy on average. Axiline+HLS design can achieve 1.06 ×. Most importantly, Axiline and Axiline+HLS designs significantly outperform HLS on the larger benchmarks with more operations (e.g., backpropagation and recommender systems). In addition, given that the performance of Axiline+HLS designs significantly depends on the optimization implemented in HLS, our Axiline+HLS results are not optimal. This indicates that Axiline and HLS are not mutually exclusive: Axiline can be improved with HLS technologies. HLS may generate an efficient design if the SML algorithm has a very simple dataflow, e.g., inference for linear regression, which only includes one IP operation. For more computational algorithms, acceleration by directly using HLS on dataflow is less energy-efficient than using Axiline or Axiline+HLS methods.

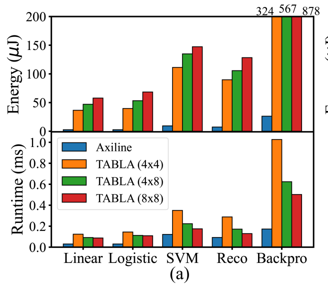

Fig. 8. Energy, runtime vs. TABLA (training benchmarks, ASIC).

VIII. Conclusion

This paper presents Axiline, a simple and effective template-based methodology for accelerating SML algorithms. Axiline first maps the nodes among any levels of granularities in *sr*-DFG of SML algorithms to pipeline templates, then optimizes the pipeline design by matching and replacement of the predefined patterns among pipeline stages. and finally generates algorithm-specific accelerators to achieve higher energy efficiency. We compared The results to TABLA and directly using HLS. Axiline-generated designs can achieve 96% lower energy, 3.3× speedup compared to TABLA and 29.7% lower energy, 1.35× speedup compared to HLS. However, Axiline and HLS are not mutually exclusive, and Axiline can be improved with HLS tools. A combined Axiline+HLS design can achieve 39.2% lower energy, 1.38× speedup than HLS.

Acknowledgments and Disclaimers

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] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[2] D. Silver *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[4] UCI Machine Learning Repository, "Geo-magnetic field and WLAN dataset for indoor localisation," 2017. [Online]. Available: https://archive.ics.uci.edu/ml/datasets.php

[5] H. Esmaeilzadeh *et al.*, "VeriGOOD-ML: An open-source flow for automated ml hardware synthesis," in *Proc. ICCAD*, 2021.

[6] M. Ebrahimi, M. H. Khoshtaghaza, S. Minaei, and B. Jamshidi, "Vision-based pest detection based on SVM classification method," *Computers and Electronics in Agriculture*, vol. 137, pp. 52–58, 2017.

[7] S. Huang *et al.*, "Applications of support vector machine (SVM) learning in cancer genomics," *Cancer Genomics & Proteomics*, vol. 15, no. 1, pp. 41–51, 2018.

[8] X. Feng *et al.*, "Towards a unified architecture for in-RDBMS analytics," in *Proc. ACM SIGMOD*, 2012, pp. 325–336.

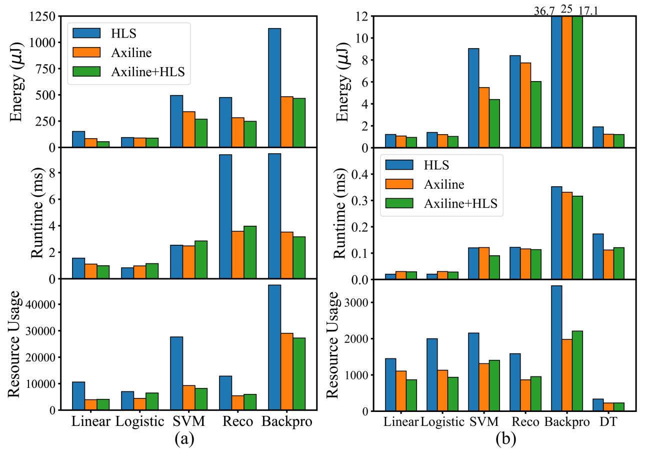

Fig. 9. Energy, runtime comparison vs. HLS (training benchmarks, FPGA).

[9] N. Linty *et al.*, "Detection of GNSS ionospheric scintillations based on machine learning decision tree," *IEEE Aerosp. Electron. Syst. Mag.*, vol. 55, no. 1, pp. 303–317, 2018.

[10] W. Kuang *et al.*, "Machine learning-based fast intra mode decision for HEVC screen content coding via decision trees," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 30, no. 5, pp. 1481–1496, 2019.

[11] M. Fernández-Delgado *et al.*, "Do we need hundreds of classifiers to solve real world classification problems?" *JMLR*, vol. 15, no. 90, pp. 3133–3181, 2014.

[12] M. W. Ahmad *et al.*, "Trees vs neurons: Comparison between random forest and ann for high-resolution prediction of building energy consumption," *Energ. Buildings*, vol. 147, pp. 77–89, 2017.

[13] A. Lu *et al.*, "CHIP-KNN: a configurable and high-performance k-nearest neighbors accelerator on cloud FPGAs," in *Proc. ICFPT*, 2020, pp. 139–147.

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

[15] T. Koide *et al.*, "FPGA implementation of type identifier for colorectal endoscopie images with NBI magnification," in *Proc. APCCAS*, 2014, pp. 651–654.

[16] Y. Ago *et al.*, "A classification processor for a support vector machine with embedded DSP slices and block RAMs in the FPGA," in *Proc. MCSoC*, 2013, pp. 91–96.

[17] M. Qasaimeh *et al.*, "FPGA-based parallel hardware architecture for real-time image classification," *IEEE Trans. Comput. Imaging*, vol. 1, no. 1, pp. 56–70, 2015.

[18] S. Afifi *et al.*, "Dynamic hardware system for cascade SVM classification of melanoma," *Neural Computing and Applications*, vol. 32, no. 6, pp. 1777–1788, 2020.

[19] O. Elgawi *et al.*, "Energy-efficient embedded inference of SVMs on FPGA," in *Proc. ISVLSI*, 2019, pp. 164–168.

[20] A. Pullini *et al.*, "Mr. Wolf: An energy-precision scalable parallel ultra low power SoC for IoT edge processing," *IEEE J. Solid-State Circuits*, vol. 54, no. 7, pp. 1970–1981, 2019.

[21] E. Tabanelli *et al.*, "DNN is not all you need: Parallelizing non-neural ML algorithms on Ultra-Low-Power IoT processors," *arXiv preprint arXiv:2107.09448*, 2021.

[22] F. Saqib *et al.*, "Pipelined decision tree classification accelerator implementation in FPGA (DT-CAIF)," *IEEE Trans. Comput.*, vol. 64, no. 1, pp. 280–285, 2015.

[23] M. Corazao *et al.*, "Performance optimization using template mapping for datapath-intensive high-level synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 15, no. 8, pp. 877–888, 1996.

[24] A. Ayupov *et al.*, "A template-based design methodology for graph-parallel hardware accelerators," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 2, pp. 420–430, 2018.

[25] M. Minutoli *et al.*, "SODA: A new synthesis infrastructure for agile hardware design of machine learning accelerators," in *Proc. ICCAD*. IEEE, 2020, pp. 1–7.

[26] J. J. Zhang *et al.*, "Towards automatic and agile ai/ml accelerator design with end-to-end synthesis," in *Proc. ICCAD*, 2021, pp. 218–225.

[27] S. Kinzer *et al.*, "A computational stack for cross-domain acceleration," in *Proc. HPCA*, 2021, pp. 54–70.

[28] "TABLA," 2021, github.com/VeriGOOD-ML/public/tree/main/tabla.