TABLA: A Unified Template-based Framework for Accelerating Statistical Machine Learning

Divya Mahajan Jongse Park Emmanuel Amaro Hardik Sharma
Amir Yazdanbakhsh Joon Kyung Kim Hadi Esmaeilzadeh

Alternative Computing Technologies (ACT) Lab
Georgia Institute of Technology

{divya_mahajan, jspark, amaro, hsharma, a.yazdanbakhsh, jkkim}@gatech.edu hadi@cc.gatech.edu

ABSTRACT

A growing number of commercial and enterprise systems increasingly rely on compute-intensive Machine Learning (ML) algorithms. While the demand for these compute-intensive applications is growing, the performance benefits from general-purpose platforms are diminishing. Field Programmable Gate Arrays (FPGAs) provide a promising path forward to accommodate the needs of machine learning algorithms and represent an intermediate point between the efficiency of ASICs and the programmability of general-purpose processors. However, acceleration with FPGAs still requires long development cycles and extensive expertise in hardware design. To tackle this challenge, instead of designing an accelerator for a machine learning algorithm, we present TABLA, a framework that *generates* accelerators for a class of machine learning algorithms. The key is to identify the commonalities across a wide range of machine learning algorithms and utilize this commonality to provide a high-level abstraction for programmers. TABLA leverages the insight that many learning algorithms can be expressed as a stochastic optimization problem. Therefore, learning becomes solving an optimization problem using stochastic gradient descent that minimizes an objective function over the training data. The gradient descent solver is fixed while the objective function changes for different learning algorithms. TABLA provides a template-based framework to accelerate this class of learning algorithms. Therefore, a developer can specify the learning task by *only* expressing the gradient of the objective function using our high-level language. TABLA then automatically generates the synthesizable implementation of the accelerator for FPGA realization using a set of hand-optimized templates.

We use TABLA to generate accelerators for ten different learning tasks targeted at a Xilinx Zynq FPGA platform. We rigorously compare the benefits of FPGA acceleration to multi-core CPUs (ARM Cortex A15 and Xeon E3) and many-core GPUs (Tegra K1, GTX 650 Ti, and Tesla K40) using real hardware measurements. TABLA-generated accelerators provide 19.4× and 2.9× average speedup over the ARM and Xeon processors, respectively. These accelerators provide 17.57×, 20.2×, and 33.4× higher Performance-per-Watt in comparison to Tegra, GTX 650 Ti and Tesla, respectively. These benefits are achieved while the programmers write less than 50 lines of code.

1 Introduction

A wide range of commercial and enterprise applications such as health monitoring, social networking, e-commerce, and financial analysis, rely on Machine Learning (ML) to accomplish their objectives. In fact, the advances in machine learning are changing the landscape of computing towards a more personalized and targeted experience for users. For instance, services that provide personalized health-care and targeted advertisements are either prevalent or on the horizon. Nevertheless, machine learning algorithms are computationally intensive workloads. Specifically, learning a model from data requires substantial amount of computation that is repeated over the training data for a relatively large number of iterations. While the demand for these computationally intensive techniques is increasing, the benefits from general-purpose solutions is diminishing [1–3]. With the effective end of Dennard scaling [4], traditional CMOS scaling no longer provides performance and efficiency gains commensurate with increases in transistor density [1–3]. The current paradigm of general-purpose processor design falls significantly short of the traditional cadence of performance improvements [5]. These challenges have coincided with the explosion of data where the rate of data generation has reached an overwhelming level that is beyond the capabilities of current computing systems [6] to match.

As a result, both the industry and the research community are focusing on programmable accelerators, which can provide large gains in efficiency and performance by restricting the workloads they support [3, 7–11]. Using FPGAs as programmable accelerators has the potential for significant performance and efficiency gains while retaining some of the flexibility of general-purpose processors [12]. Commercial platforms incorporating general purpose cores with programmable logic are beginning to appear [13, 14]. For instance, Microsoft employs FPGAs to accelerate their Bing search service [7]. FPGA's increasing availability and flexibility makes them an attractive platform to accelerate ML algorithms. However, a major challenge in using FPGAs is their programmability. Development with FPGAs still requires extensive expertise in hardware design and implementation and the overall design cycle is long even for experts [7]. This paper aims to tackle this challenge for an important class of ML algorithms by presenting the TABLA[1] framework. TABLA is template-based solution–from programming model to circuits–that uses FPGAs to accelerate statistical machine learning. The objective of our solution is to devise the necessary programming abstractions and an automated framework that is uniform across a range of machine learning algorithms. TABLA aims to avoid exposing software developers to the details of hardware design by leveraging commonalities in ML algorithms.

While developing TABLA, we leveraged the insight that many learning algorithms can be expressed as stochastic optimization problems [15]. Examples of such learning algorithms are support vector machines, logistic regression, least square models,

[1] **T**emplate-based **A**ccelerator **B**uilder for **L**earning **A**lgorithms.

978-1-4673-9211-2/16/$31.00©2016 IEEE

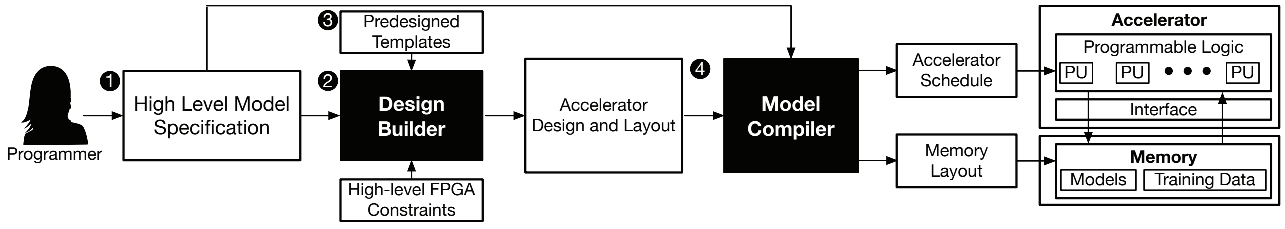


Figure 1: Overview of TABLA's workflow. The programmer only provides the gradient of the objective function, representing the learning algorithm, in TABLA's high-level programming language. The other major components of TABLA are: (a) the design builder that automatically generates the synthesizable Verilog implementation of the accelerator from a set of pre-designed templates; and (b) the model compiler that generates the execution schedule for the accelerator, the memory layout, and the memory access schedule.

backpropagation, conditional random fields, recommender systems, Kalman filters, linear and nonlinear regression models, and softmax functions. These types of learning algorithms can be optimized using stochastic gradient descent[16]. That is, the learning task becomes solving an optimization using stochastic gradient descent that iterates over the training data and minimizes an objective function. Although the objective function varies for different learning algorithms, the stochastic gradient descent solver is fixed. Therefore, the accelerator for these learning tasks can be implemented as a template design, uniform across a class of machine learning algorithms. This template design comprises the general framework for stochastic gradient descent.

TABLA automatically specializes the template design for a specific learning task by generating and integrating the hardware blocks that implement the gradient of the objective function. Therefore, a developer can specify the learning task by *only* writing the gradient of the objective function using our high-level language. The gradient function can be implemented with less than 50 lines of code for logistic regression, support vector machines, recommender systems, backpropagation, and linear regression. TABLA automatically generates a concrete accelerator (synthesizable Verilog code) for the specific learning algorithm using a set of hand-optimized template designs while considering high-level design parameters of the target FPGA. To this end, our work makes the following contributions:

(1) We observe that many common machine learning algorithms can be represented as stochastic optimization problems. This observation enables TABLA to provide a high-level, intuitive, uniform, and automated abstraction to use FPGAs to accelerate an important class of machine learning algorithms.

(2) Using this observation, we develop a comprehensive solution–from programming model to circuits–that abstracts away the details of hardware design from the programmer, yet generates accelerators for a range of machine learning algorithms.

(3) We use TABLA to generate accelerators for five different learning algorithms–logistic regression, SVM, recommender systems, backpropagation and linear regression–each with two different topologies. We use TABLA to generate ten different accelerators for these ten different learning tasks and evaluate them on the Xilinx Zynq FPGA platform.

We rigorously compare the benefits of the FPGA implementation to both multicore CPUs (ARM Cortex A15 and Xeon E3) and many-core GPUs (Tegra K1, GTX 650 Ti, and Tesla K40), using real hardware measurements. TABLA generated accelerators provide 19.4× and 2.9× average speedup over the ARM and Xeon processors, respectively. These accelerators provide 17.57×, 20.2×, and 33.4× higher Performance-per-Watt in comparison Tegra, GTX 650, and Tesla, respectively. These results suggest that TABLA takes an effective step toward widespread use of FPGAs as an accelerator of choice for machine learning algorithms.

2 Overview

Machine learning generally involves two phases: the learning phase and the prediction phase. The learning phase, which is precursory to the prediction phase, generates a model that maps one or more inputs (independent variables) onto one or more outputs (dependent variables). The generated model is used in the prediction phase to predict the dependent variables for new unseen inputs. The learning phase is more compute intensive and can benefit significantly from acceleration. Therefore, TABLA aims to provide a comprehensive solution from programming model down to circuits that can automatically generate accelerators for the learning phase of a class of ML algorithms as illustrated by Figure 1. We briefly discuss each component of TABLA below.

❶ **High-level programming model.** TABLA provides a high-level programming model that enables programmers to specify the gradient of the objective function which defines the learning algorithm. This mathematical function captures the learning algorithm. TABLA focuses on a class of learning algorithms that can be solved using stochastic gradient descent. The stochastic gradient descent solver is uniform across a range of ML algorithms and therefore, the gradient function is sufficient to generate the entire accelerator design. The programmer also provides the initial and meta-parameters of the learning algorithm.

❷ **Design builder.** After the programmer provides the gradient of the objective function, TABLA's *design builder* automatically generates the accelerator and its interfacing logic to the external memory. The design builder uses a set of pre-designed templates to generate the accelerator. The output of the design builder is a set of synthesizable Verilog designs that concretely implements the accelerator. The inputs to the design builder are (1) the gradient function, (2) a high-level specification of the target FPGA, and (3) a set of pre-designed accelerator templates in Verilog. The FPGA specification constitutes the number of DSP slices, the number of SRAM structures (Block RAMs), the capacity of each Block RAM, the number of Block RAM read/write ports, and the off-chip communication bandwidth.

❸ **Pre-designed templates.** The design builder generates the accelerator design from a set of *pre-designed templates*. These templates are generic and uniform across a large class of statistical machine learning algorithms and support all the language constructs defined in TABLA's programming interface. The templates provide a general structure for the accelerator without making it specific to a certain algorithm or FPGA specification. The templates also contain the implementation for stochastic

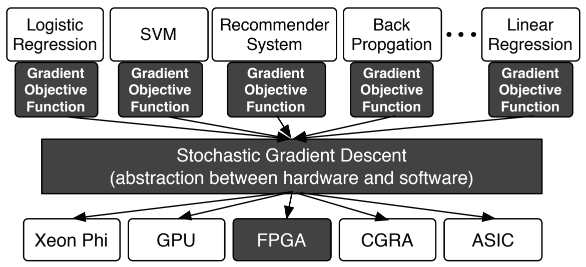

Logistic Regression
SVM
Recommender System
Back Propgation
Linear Regression
Gradient Objective Function
Stochastic Gradient Descent
(abstraction between hardware and software)
Xeon Phi
GPU
FPGA
CGRA
ASIC

Figure 2: TABLA leverages stochastic gradient descent as an abstraction between hardware and software to create a unified framework to accelerate a class of machine learning algorithms. The highlighted blocks are the focus of this work.

gradient descent, which is uniform across all the target machine learning algorithms. These predefined templates are designed by expert hardware designers and comprise both the accelerator and the interfacing logic that connects the accelerator to the rest of the system (e.g., memory).

❹ **Model compiler.** Another component of TABLA is the *model compiler* that statically generates an execution schedule for the accelerator and significantly simplifies the hardware. The inputs to the model compiler are (1) the structure of the accelerator and (2) the specification of the gradient function. The model compiler converts the gradient function to a dataflow graph and augments it with the dataflow graph of stochastic gradient descent. Then, it uses a Minimum Latency Resource-Constrained Scheduling algorithm [17] to generate the accelerator schedule. The model compiler also generates an order for the model parameters that will be learned. This order determines the layout of parameters in the memory and streamlines the interfacing logic that communicates with the memory. Finally, the model compiler generates the schedule for the memory interface.

As Figure 2 illustrates, TABLA uses stochastic gradient descent as the abstraction between hardware and software. This abstraction is basis for the templates from which TABLA generates the accelerator. As shown in Figure 2, TABLA can potentially target different platforms, including Xeon Phi, GPUs, FPGAs, CGRAs and ASICs. Specific backends need to be developed to support each of these platforms. In this paper, we focus on FPGAs since they represent a middle-ground between the efficiency of ASICs and programmability of CPUs. Before discussing the components of TABLA for FPGA platforms, the next section describes the theoretical foundation of stochastic gradient descent.

3 Background on Stochastic Gradient Descent

Stochastic gradient descent is an iterative optimization algorithm that finds a set of parameters which minimize an objective function. This section provides an in-depth detail of stochastic gradient descent and shows why we chose it as the mathematical foundation to implement TABLA.

Learning as an optimization problem. Each machine learning algorithm in our target class is distinguished by its objective function. The objective function has a set of parameters that are learned in accordance with the training data such that the learned model can make data-driven predictions or decisions on new unseen data. During each iteration, the objective function quantifies the error between the current model's output (prediction) and the expected output given in the training data. Thus, a machine learning algorithm learns a model by solving an optimization problem that minimizes the prediction error over the entire training data as shown in Equation (1).

$$\min_{W^{(t)} \varepsilon \mathbb{R}} \sum_i f(W^{(t)}, X_i) \tag{1}$$

In Equation (1), X_i is the ith input, $W^{(t)}$ is the model parameter at iteration t and $f(W_i^{(t)}, X_i)$ is the prediction error. The sum of the prediction errors across all training input vectors is the objective function that needs to be minimized. To learn a model W, optimization algorithms iterate over the training data and gradually reduce the prediction error by changing the model parameters. Gradient descent is one such common optimization algorithm. While the gradient descent algorithm is fixed across different machine learning algorithms, the objective function varies. Table 1 shows five sample machine learning algorithms and their corresponding objective function that can be trained using gradient descent.

Gradient descent. The gradient descent algorithm starts with an initial set of model parameters and iteratively moves towards a set of parameters that minimize the objective function. This iterative minimization is achieved by taking steps in the decreasing direction of the objective function's derivative or gradient. Hence, for each iteration, the parameters $W^{(t)}$ are updated as shown below.

$$W^{(t+1)} = W^{(t)} - \mu \times \frac{\partial(\sum_i f(W^{(t)}, X_i))}{\partial W^{(t)}} \tag{2}$$

As Equation (2) shows, $W^{(t+1)}$ is updated in the negative direction of the objective function's gradient ($\frac{\partial f}{\partial W^{(t)}}$) with learning rate, μ. In a single iteration of gradient descent, the gradient of the objective function calculates a sum over the entire training data to obtain the next set of parameters $W^{(t+1)}$. This process is repeated until the function is minimized and the final set of parameters $W^{(final)}$ is obtained. $W^{(final)}$ is the trained model of the machine learning algorithm for a given training dataset. For very large training datasets, gradient descent can impose a high overhead as it calculates a sum over the entire data in a single iteration. To avoid this computationally large overhead, stochastic gradient descent is generally used [15, 16, 18].

Stochastic gradient descent. Stochastic gradient descent is a modification of the conventional gradient descent algorithm. It divides the objective function into smaller differentiable functions. As Equation 1 shows, the objective function is a summation of a function over the entire training data. Instead of taking the deriva-

Table 1: Machine learning algorithms, their objective function, and the gradient of this objective function (used with TABLA). The $\delta()$ operator in the objective and gradient functions represents a complex nonlinear transformation. For example, in logistic regression $\delta(W,X_i)$ represents $sigmoid(\sum_i X_i \times W)$.

Machine Learning Algorithm	Objective Function (f)	Gradient of the Objective Function (∂f)
Logistic Regression	$\sum_i \{ Y_i \log(\delta(W,X_i)) + (1-Y_i) \log(1-\delta(W,X_i)) \} + \lambda\|\|W\|\|$	$(\delta(W,X_i)-Y_i)X + \lambda W$
Support Vector Machines	$\sum_i \{ 1-Y_i W X_i \} + \lambda\|\|W\|\|$	$Y_i X_i + \lambda W$
Recommender Systems	$\sum_{i,j} \{ (Y_{ij}-W_j X_i)^2 \} + \lambda\|\|W,X\|\|$	$\sum_i (Y_{ij}-W_j X_i)X_i + \lambda W$, $\sum_j (Y_{ij}-W_j X_i)W_j + \lambda X$
Backpropagation	$\sum_i \sum_k \{ (Y_i^k \log(\delta(W,X_i)^k)) + (1-Y_i^k) \log(1-\delta(W,X_i)^k) \} + \lambda\|\|W\|\|$	$\sum_k(\alpha_i^k \Delta_i^k) + \lambda W$
Linear Regression	$\sum_i \{ \frac{1}{2} (WX_i-Y_i)^2 \} + \lambda\|\|W\|\|$	$(WX_i-Y_i)X_i + \lambda W$

tive of the function calculated over the entire dataset, stochastic gradient descent divides the objective function into smaller functions requiring a single input vector. Therefore, the gradient of the smaller function is only calculated over a single vector. Thus, the parameter update rule changes from Equation (2) to Equation (3).

$$W^{(t+1)} = W^{(t)} - \mu \times \frac{\partial f(W^{(t)}, X_i)}{\partial W^{(t)}} \quad (3)$$

The calculation in Equation (3) is repeated individually for all input vectors X_i, until the function converges to its minimum value. Stochastic gradient descent typically takes more iterations to converge in comparison to conventional gradient descent. However, the benefits obtained by avoiding data accesses to all the input vectors for each iteration significantly outweigh the cost incurred by executing more iterations. Using stochastic gradient descent to find the minimum of the objective function is imperative for large training datasets across different domains of machine learning algorithms. This insight motivated us to choose stochastic gradient descent as the abstraction between the software and the hardware for TABLA, as shown in Figure 2. To specialize the hardware templates, TABLA only requires the programmer to specify the learning model as the gradient of the objective function ($\frac{\partial f}{\partial W^{(t)}}$). The only programming task is to implement this gradient function for the corresponding algorithm. Our experience shows that the gradient function can be implemented with less than 50 lines of code for logistic regression, SVM, recommender systems, back-propagation, and linear regression (summarized in Table 3). Other algorithms such as conditional random fields, Kalman filters, portfolio optimization, and least square models can also be accelerated using TABLA since they can be optimized using stochastic gradient descent.

After the programmer provides the gradient of the objective function, TABLA can automatically generate a concrete accelerator for the specific learning task. In the following sections, we describe TABLA's components: programming interface (Section 4), model compiler (Section 5), and template designs (Section 6).

4 Programming Interface

In TABLA, the programmer expresses ML algorithms by specifying the gradient of the objective function. The programmer uses our high-level programming interface[2] to specify this gradient function. Our programming interface provides the flexibility to represent a wide range of ML algorithms and possesses the following properties: (1) it is a high-level language that enables the representation of learning algorithms in a fashion that is familiar to ML experts and is close to their mathematical formulation (e.g., Table 1); and (2) it incorporates language constructs that are commonly seen in a wide class of statistical learning algorithms. The interface comprises two language constructs: data declarations and mathematical operations. Data declarations, detailed in Section 4.1, allow the programmer to express different data types that represent the training data and model parameters. Further, the mathematical operations, described in Section 4.2, enable the programmer to declare different numerical operations used to calculate the gradient of an objective function. Table 2 summarizes these language constructs.

[2]The details of TABLA's domain specific language, its formal syntax, grammar, and semantics of each construct is available in `http://act-lab.org/artifacts/tabla`.

Table 2: TABLA's language constructs that enable convenient representation of a wide class of learning algorithms.

Type	Connotation	Keyword
Data Declarations	Learning model inputs	`model_input`
	Learning model outputs	`model_output`
	Learning model parameters	`model`
	Gradient of the objective function	`gradient`
	Iterator variable	`iterator`
Mathematical Operations	Basic operations	`+,-,*,/`
	Group operations	`pi, sum, norm`
	Nonlinear transformations	`gaussian, sigmoid, …, log`

4.1 Data Declarations

Data declarations enable the programmer to specify different data types used in the gradient of the objective function. These data types include: model input, model output, model parameters, gradient, and iterators. The data declarations emphasize the different semantics held by the data in an ML algorithm. For example, the model_input keyword refers to an input vector (independent variables) while the model_output declaration refers to its corresponding output vector (dependent variables). Both model_input and model_output together form the training data. Both these data types are inputs to the learning algorithm while the algorithm learns the model. The gradient keyword declares the gradient of the objective function. Further, the model keyword declares the model parameters that are updated every iteration in accordance with the gradient of the objective function. Finally, the iterator keyword identifies arrays, their dimensions, and their operations. The following code snippet illustrates the use of iterators.

```
...
iterator i[0:n-1]; //iterator for arrays
Q[i] = A[i] * B[i]; //element-by-element multiplication
s = sum [i](Q[i]); //group summation
...
```

In this example, i is an iterator variable that ranges from 0 to n-1 and can iterate over arrays with n elements starting from index 0. For example, Q[i] = A[i] * B[i] statement uses i to perform an element-by-element multiplication between the two arrays, both of size n. Moreover, iterators can imply the autonomy of the array operations. For instance, the A[i] * B[i] cane parallelized over all the values of i. Iterators are also used in group operations to identify the array of operands. In the above example, sum[i](Q[i]) denotes that all the elements of Q need to be summed together.

As discussed, these data declarations enable programmers to specify the semantics and characteristics of different data elements in learning algorithms. Another major component of the learning algorithms is the mathematical operations that are defined over these data elements. Below, we discuss the language constructs that support these mathematical operations.

4.2 Mathematical Operations

Our language supports three types of mathematical operations: basic operations, group operations, and nonlinear transformations.

Basic operations. These operations are basic mathematical operations such as -, +, *, and /.

Group operations. These operations are performed over a group of elements and include sum ($\sum$), pi (Π), and norm ($\| \|$). Besides an operand, group operations require an iterator argument. The iterator specifies the elements on which the calculation is performed. These operations generate an output with dimension one less than the input operand's dimensionality. For instance, summing the elements of a one-dimensional array generates a scalar. **Nonlinear transformations.** These mathematical operations apply nonlinear functions (e.g., log, sigmoid, gaussian) over their operands. Since the transformation is applied to each element individually, the output has the same dimensions as the input.

Using these mathematical operations and data declarations, programmers can specify a wide range of learning algorithms at a high level without delving into the details of hardware implementation. We further demonstrate the capabilities of the language using a concrete example that implements logistic regression.

4.3 Example: Logistic Regression

As mentioned before, the programmer only needs to specify the gradient of the objective function for the learning algorithm. Equation (4) shows this gradient for logistic regression.

$$G_{n\times m} = \left(\left[\forall j \in [0,n) \middle| sigmoid(\sum_{i=0}^{m-1} X_i \times W_{j,i}) \right]_{1\times n} - Y'_{1\times n} \right)^T_{n\times 1} \times X_{1\times m} + lambda \times W_{n\times m} \quad (4)$$

In this equation, G is the gradient matrix with n rows and m columns; X is an input vector with m elements; Y' is the expected output vector with n elements; W is the matrix with $n \times m$ elements that contains the model parameters; and $lambda$ is the regularization factor, which is a scalar. The following code shows how this gradient function can be expressed in a few lines using TABLA's programming language.

```
m = 53 //number of input features
n = 3 //number of model outputs
lambda = 0.1 //regularization factor

model_input   X[m]; //model input
model_output  Y'[n]; //model output
model         W[n][m]; //model parameters
gradient      G[n][m]; //gradient

iterator i[0:m - 1]; //iterator for group operations
iterator j[0:n - 1]; //iterator for group operations

//m parallel multiplies followed by
//an adder tree; repeated n times in parallel
S[j] = sum [i](X[i] * W[j][i]);

Y[j] = sigmoid (S[j]); //n parallel sigmoid operations
E[j] = Y[j] - Y'[j]; //n parallel subtractions
G[j][i] = X[i] * E[j]; //n*m parallel multiplications
V[j][i] = lambda * W[i][j]; //n*m parallel multiplications
G[j][i] = G[j][i] + V[j][i]; //n*m parallel additions
```

The above listing shows how a complex mathematical function is implemented in a textual format using TABLA's programming language. The data declarations (e.g., model_input, model_output, model, and gradient) identify the semantics of different data types. The rest of the textual representation has a close correspondence to the mathematical formulation in Equation (4). The gradient formula is simply broken down to multiple statements that correspond to the mathematical operations. This correspondence and the simplicity of the statements makes programming with TABLA convenient for machine learning programmers.

In the code, the two iterators i and j correspond to the subscripts in Equation (4) and are used to iterate over the elements of the input (X) and output (Y) vectors as well as the matrices that store the model (W) and the gradient (G). The sum statement represents the $\sum_{i=0}^{m-1} X_i \times W_{j,i}$ part of the gradient. The iterator for sum is i, similar to the formula in which $\sum$ iterates over i. This statement first performs the multiplication $X_i \times W_{j,i}$ and then accumulates all the multiplication results into a single result S[j] assuming a constant j. The left hand side of the statement, S[j], mandates that the accumulation needs to be repeated n times using the j iterator. Next, the sigmoid statement continues the gradient calculation as shown by $\left[\forall j \in [0,n) \middle| sigmoid(\sum_{i=0}^{m-1} X_i \times W_{j,i}) \right]_{1\times n}$. The rest of the code similarly performs the remaining part of the gradient computation. At the end, the G[j][i] variable in the code corresponds to the elements of the $G_{n\times m}$ matrix in Equation (4).

A wide range of machine learning algorithms can be represented using TABLA's programming interface. Furthermore, the programming interface can be extended to accommodate the representation of an even wider range of learning algorithms. Although MATLAB and R can also be used to represent the same learning algorithms, we designed and used TABLA's own programming interface because of: (1) easier representation of gradient functions using the common mathematical constructs used in machine learning; (2) clear-cut identification of parallelism in the code; and (3) convenient conversion of gradient function into the final hardware design using our model compiler, described in the next section. We are also working on providing translators that convert MATLAB and R code to TABLA's language.

5 Model Compiler for TABLA

TABLA's model compiler statically generates an execution schedule for the accelerator using the gradient of the objective function provided by the programmer. The model compiler accomplishes this task in three steps. The first step integrates the gradient of the objective function with the stochastic gradient descent solver. In the second step, the compiler generates an intermediate representation, i.e., the Dataflow Graph (DFG) of the entire learning algorithm. Finally, in the last step, the compiler translates this Dataflow Graph (DFG) into a static schedule for hardware execution. We specifically use static scheduling since it simplifies the hardware and improves the efficiency of the accelerated execution. Each of these compilation steps are described in further detail in this section.

5.1 Integrating Stochastic Gradient Descent

After the programmer provides the gradient of the objective function, TABLA uses the stochastic gradient algorithm to learn the model parameters from the training data. Learning a model from the training data requires a solver that finds the minimum value of the objective function that represents the learning algorithm. Since stochastic gradient descent is independent of the learning model, we devise a general template to implement it. TABLA integrates this template with the programmer-provided gradient code using the gradient and model variables. These keywords explicitly identify the inputs to the stochastic gradient descent solver. The following code snippet shows the template code of the stochastic gradient descent solver.

```
gradient      G[n][m]; //gradient
model         W[n][m]; //model parameters

iterator i[0:m - 1]; //iterator for group operations
iterator j[0:n - 1]; //iterator for group operations

G[j][i] = u * G[j][i] //n*m parallel multiplications
W[j][i] = W[j][i]-G[j][i]; //n*m parallel subtractions
```

As the code shows, the model parameters (W[n][m]) are updated in the opposite direction of the gradient (G[n][m]) with a rate u, called the learning rate. Once the gradient function is integrated with the stochastic gradient descent solver, the model compiler generates the DFG of the entire learning algorithm.

5.2 Generating Dataflow Graph

TABLA's model compiler converts any code written in our programming interface into a dataflow graph. Each language construct corresponds to a small and simple DFG. The model compiler scans the code and replaces each construct with its corresponding DFG. The compiler then links these small DFGs to create the final DFG for the learning algorithm.

Dataflow graph of individual operations. Figure 3 shows sample DFGs of three types of mathematical operations that are supported by TABLA's language: basic, group, and nonlinear. The nodes are basic computations and the edges capture the dependencies. The group operations require more than one computational node. As Figure 3 depicts, the DFG for sum is an adder tree and the DFG for norm includes a layer of multiplications that feed into an adder tree. The DFG also captures the parallelism amongst the basic computations and enables the model compiler to generate an efficient execution schedule for the accelerator.

Dataflow graph of the learning algorithm. Figure 4 shows the complete DFG for logistic regression. This DFG corresponds to the example code provided in Section 4.3 when n is 1. As the figure illustrates, the model compiler combines and links the DFG of each operation to generate the entire DFG. For example, the compiler converts the sum[i](X[i]*W[j][i]) statement to a series of multiplications followed by an adder tree, which is the DFG for sum (Figure 3). Translating code to DFG is straightforward since the dataflow graphs of each operation is predetermined. As shown at the bottom of Figure 4, the DFG of logistic regression also includes the computation of stochastic gradient descent solver.

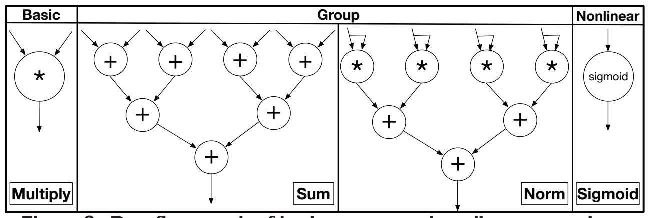


Figure 3: Dataflow graph of basic, group, and nonlinear operations.

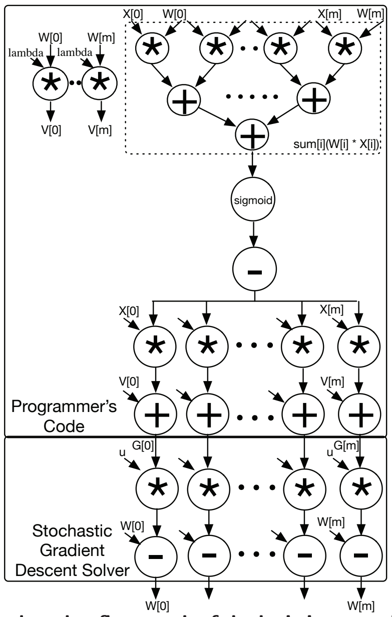


Figure 4: Complete dataflow graph of the logistic regression algorithm.

5.3 Static Scheduling

Once the DFG is generated, the model compiler statically generates a step-by-step schedule of each operation. We use the Minimum Latency–Resource Constrained Scheduling (ML–RCS) algorithm [17] to generate this schedule[3]. This algorithm aims to optimize for minimum execution latency while being constrained by the limited set of resources available on the accelerator platform. Algorithm 1 presents this scheduling algorithm.

```
Inputs:  $R$: Set of available resources
         $O$: Set of all the operations to be scheduled
         $D$: Distance to sink for each operation
Output:  $S$: Final schedule
  Initialize $S \leftarrow \emptyset$
  Initialize $current\_cycle \leftarrow 0$
  while ($O \neq \emptyset$) do
    for ($r \in R$) do
      if $o \in O$ where $o$.predecessors = DONE & $o$.distance = max($D$) then
        $schedule$.op = $o$; $schedule$.resource = $r$; $schedule$.cycle = $current\_cycle$
        $S$.append($s$)
        $O$.remove($o$)
      end if
    end for
    $current\_cycle = current\_cycle + 1$
  end while
```

Algorithm 1: Minimum-latency resource constrained scheduling.

To understand the algorithm, we first define a property of each operation (o) called *distance from sink*, denoted as "o.distance" in the Algorithm 1. *Distance from sink* of o is the number of dependent operations between o and the final output or the sink of the DFG. This distance captures the criticality of an operation. The higher an operation's *distance from sink*, higher its scheduling priority. Algorithm 1 picks an available resource r and schedules an operation o at the *current_cycle* on r if the following two conditions are satisfied: (1) all the predecessors of o have already been scheduled, i.e., o is ready; and (2) o is on the critical path of execution, i.e., its *distance from sink* is the maximum among all unscheduled ready operations. The algorithm picks the next available resource or increments the *current_cycle* if all the resources are being utilized. The algorithm terminates when all the operations are scheduled. To generate this schedule, TABLA's design builder first needs to generate the skeleton of the accelerator and determine the number of available resources. The next section discusses this process and the template architecture of the accelerator.

6 Design Builder and Template Designs

6.1 Design Builder

TABLA's design builder generates synthesizable Verilog code of the learning accelerator given the DFG and schedule of the learning algorithm, and the high-level specification of the target FPGA. The FPGA specification comprises the number of DSP slices, the ALU operations supported in the DSP slices, the number of SRAM structures (Block RAMs), the capacity of each Block RAM, the number of read/write ports on a Block RAM, and the off-chip communication bandwidth. Given this informa-

[3] Note that the DFG itself represents the As-Soon-As-Possible (ASAP) schedule of the operations. In the ASAP schedule, an operation is scheduled for computation as soon as it is ready, i.e., all of its predecessors have finished their computation. The ASAP schedule provides minimum latency; however, it assumes infinite resources. We use ML–RCS since it considers the limited availability of compute resources.

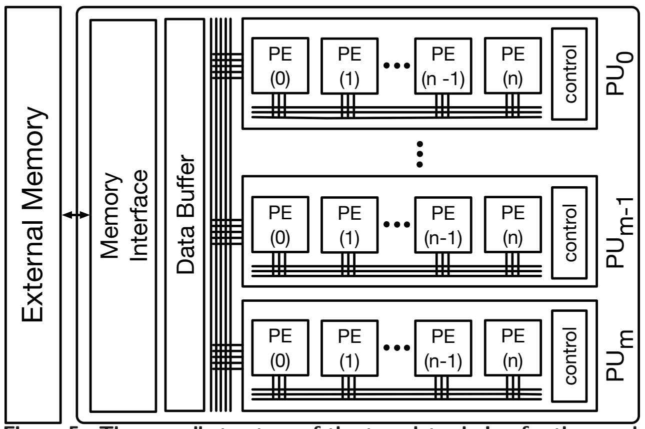
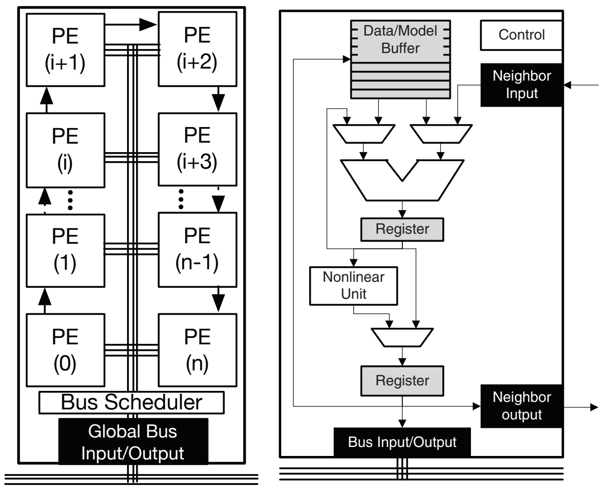


Figure 5: The overall structure of the template design for the accelerators, which is a scalable, general, modular, and highly customizable architecture. The design builder shrinks or expands the template architecture based on the requirements of the DFG and the availability of resources on the target FPGA. This hierarchical design is clustered into a set of PUs that comprise of a number of PEs. The PU are connected through an inter-PU bus that is also connected to the memory interface. The PEs use a dedicated intra-PU bus to communicate.



(a) Processing Unit (PU) (b) Processing Engine (PE)

Figure 6: (a) Template PU design comprising a set of PEs that are connected through an intra-PU bus. This bus is also connected to the global inter-PU bus. (b) Template PE design with ALU, control logic, data buffer, nonlinear unit, and the links to the neighboring PEs.

tion and the DFG of the learning algorithm, the design builder customizes our hand-optimized template accelerator architecture for the specified machine learning algorithm.

As Figure 5 shows, the template design is a clustered hierarchical architecture. A series of Processing Units (PUs) that include a set of Processing Engines (PEs) constitute this hierarchical architecture. This clustered template architecture is scalable, general, and highly customizable. The design builder shrinks or expands this template design considering the degree of parallelism in the DFG and the availability of the resources in the target FPGA. The design builder first extracts the maximum number of parallel operations from the DFG and select the total number of the PEs accordingly. Based on the DFG, the design builder also determines the ALU operations and the nonlinear transformation units that need to be included in the PEs. If an ALU operation or a nonlinear function is not used in the DFG, the corresponding hardware unit is excluded from the final accelerator design. The design builder also generates the control unit of the PEs, PUs, and the buses according to the schedule of operations. The scheduling algorithm and the design builder work in tandem. The design builder determines the number of PEs (compute resources) for the scheduler to generate the execution schedule. Then, based on the schedule, the design builder generates the control logic. The design builder also determines the number of PEs per each PU depending on the target FPGA as we will discuss in Section 7.3. Finally, the design builder adds the memory interface unit and generates the access schedule to the memory according to the execution schedule. The remainder of this section discusses the PU and the PE designs.

6.2 Template Design for Processing Units

As shown in Figure 5, the processing units construct the first level of hierarchy in our template design. The PUs are self-contained structures that comprise a set of identical processing engines as depicted in Figure 6a. Grouping PEs as PUs makes the template design modular and localizes the majority of data traffic within PUs. Both modularity and locality of traffic enhances the scalability and the customizability of the template design. This characteristic of the template enables the design builder to generate a concrete accelerator design with any number of PUs. Conceptually, a single PU can carry out the computation of an entire learning algorithm. However, the design builder scales up the number of PUs if the DFG can utilize the additional resources. The PUs are connected through a pipelined global bus. The communication between PUs is statically orchestrated by the model compiler and is loaded into the PUs as part of the accelerator configuration. The PUs are also connected to the memory interface through a data buffer. The PUs are merely consumers of data and do not initiate requests. The data buffer fetches data from the external memory and sends the data to the PUs according to a static schedule generated by the model compiler. The static scheduling of communication significantly simplifies the PU design and the busing logic. This hardware-software co-design approach makes the accelerator template design scalable and enables the design builder to cater the needs of a variety of learning algorithms.

6.3 Template Design for Processing Engine

Figure 6b depicts the template design of the processing engines. PEs are the basic blocks of our template design and are customized according to the DFG of the learning algorithm. As illustrated, each PE contains an ALU that performs the calculations and a local memory (data/model Buffer) that stores the model parameters and data elements. Some of the components are fixed within a PE, while the others are customizable based on the learning algorithm's DFG.

The *fixed components* in a PE are the ALU, data/model buffer, registers, and busing logic. All the learning algorithms have some form of mathematical operations, making the ALU a fixed component. Although the ALU is fixed, the operations that it supports changes according to the learning algorithm's DFG. Additionally, a buffer is necessary to store the model parameters or any other incoming data. The buffer retains the model parameters that are updated (learned) during the execution. The PE's share of training data is also stored in this buffer. The registers are essential to a PE as they enable the storage of intermediate results. Finally, bus interfaces are always needed to channel the incoming data from memory or other PEs and PUs.

Table 3: Benchmarks, their brief description, size of the training datasets, number of input features, model topology, lines of code to express the gradient function of the learning algorithm with TABLA's programming interface, and the number of PEs in the TABLA generated accelerators.

Name	Model	Algorithm Name	Description	Input Vectors	# of Features	Model Topology	Lines of Code	# of PEs
LogisticR	M1	Logistic Regression	Estimates the probability of dependent variable given one or more independent variables	581,000	54	54	20	32
	M2			500,000	200	200	20	64
SVM	M1	Support Vector Machines	Classifies data into different categories by identifying support vectors	581,000	54	54	23	32
	M2			500,000	200	200	23	64
Reco	M1	Recommender Systems	Information filtering system that predicts the preference a user would give to an item	1,700,000	2,700	27,000	31	32
	M2			24,000,000	10,000	100,000	31	64
Backprop	M1	Backpropogation	Trains a neural network that models the mapping between the inputs and outputs of the data	38,000	10	10 -> 9 -> 1	48	64
	M2			90,000	256	256 -> 128 -> 256	48	64
LinearR	M1	Linear Regression	Models relationship between a dependent variable and one or more explanatory variables	10,000	55	55	17	64
	M2			10,000	784	784	17	64

The highly *customizable components* in the PE are the control unit, the nonlinear unit, and the neighbor input/output communication links. Firstly, the control unit stores the PE's schedule of operations. This schedule is a queue of predetermined control signals that directs different components of the PE. This schedule changes with the DFG of the learning task. Secondly, the nonlinear unit is not required by some algorithms such as SVM, recommender system, and linear regression. This unit is excluded or customized according to the algorithm. Finally, communication between neighboring PEs is only useful for learning algorithms that aggregate data (e.g., use sum ($\sum$) or pi (Π)). During aggregation, the short direct links between neighboring PEs enable parallel exchange of data without serializing computation by requiring PEs to contend for the intra-PU bus. Once the design builder customizes the PE design in congruence with the DFG of the learning algorithm, it groups the PEs as PUs and generates the final concrete accelerator as synthesizable Verilog code.

In the next section, we evaluate TABLA-generated accelerators for ten different learning tasks.

7 Evaluation

We evaluate TABLA using an off-the-shelf Xilinx Zynq ZC702 FPGA platform, specifications of which are summarized in Table 4. We synthesize the TABLA-generated accelerators with 64-bit Vivado v2015.1. The accelerators are connected to the external memory via four Xilinx Advanced eXtensible Interface (AXI) controllers and operate at 100 MHz. We compare the performance and energy benefits of these FPGA accelerators to a diverse set of high-performance and low-power CPUs and GPUs. We use hardware measurements to rigorously compare the accelerator benefits to both multicore CPUs (ARM Cortex A15 and Xeon E3) and many-core GPUs (Tegra, GTX 650 Ti, and Tesla K40).

7.1 Experimental Setup

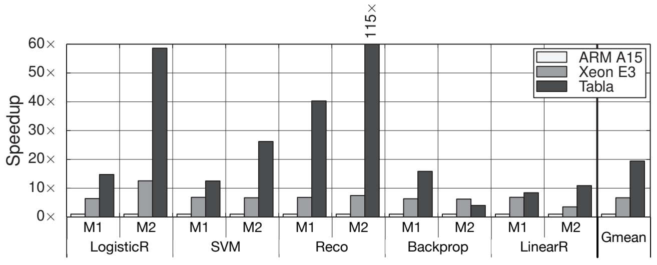
7.1.1 Benchmarks and Training Datasets

Table 3 lists the machine learning algorithms used to evaluate TABLA. We study five popular machine learning algorithms: Logistic Regression (LogisticR), Support Vector Machines (SVM), Recommender Systems (Reco), Backpropagation (BackProp), and Linear Regression (LinearR). These algorithms represent a wide range of learning algorithms encompassing regression analysis, statistical classification, information filtering systems, recommender systems, and artificial neural networks. Table 3 also includes some of the most pertinent learning parameters such as the number of training vectors, the model topology, the number of lines required to implement the gradient function in the TABLA programming interface, and the number of PEs/PUs constituting the design of each algorithm. Each algorithm is evaluated with two model topologies M1 and M2. The evaluation across multiple models allows us to evaluate the flexibility of the TABLA framework to accommodate changes in the topology of a machine learning algorithm. For LogisticR and SVM, we use two model topologies from the UCI repository[19]. One dataset comprises 54 features and the other dataset consists of 200 features. We modified the datasets to incorporate binary output values. For Reco, we use two different topologies from movieLens [20, 21], a movie database. For BackProp we use two topologies: a large neural network topology (256→128→256) [22] and a small neural network topology (10→9→1) [23]. For LinearR we use one topology from the UCI repository and one from MNIST [24]. The Input Vectors column in Table 3 shows the number of input vectors in each training data set. The # of Features column denotes the number of independent variables. The Model Topology column shows the topology and the number of parameters that are trained via the learning task. Finally, the Lines of Code column lists the number of lines of code that were required to implement each learning task's gradient function using TABLA's programming interface. The number of lines vary from 17 for LinearR to 48 for Backprop depending on the complexity of the gradient function for a given algorithm . These numbers suggest that TABLA establishes an effective high level programming interface that abstracts away the intricate details of hardware design from its users. Finally, the # of PEs column gives the total number of PEs for each benchmark in the final accelerator design generated by TABLA.

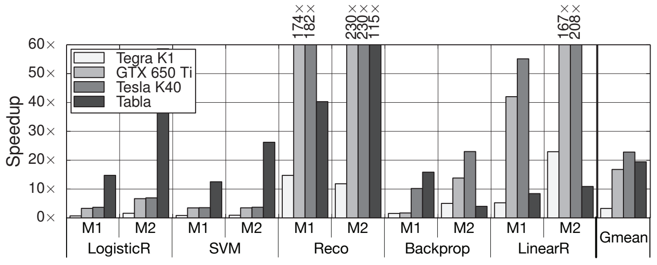
7.1.2 CPU and GPU Platforms

As shown in Table 5, we compare TABLA-generated accelerators with two multicore CPU processors: (1) the low-power quad-core ARM A15 available on the Nvidia Jetson TK1 platform [25] that operates at 2.3 GHz; and (2) the high performance quad-core Intel Xeon E3 with hyper-threading support that operates at 3.6 GHz. We also compare TABLA-generated accelerators to three GPU processors: (1) the low-power Tegra K1 GPU, which is available on the Jetson TK1 board with 192 SIMD cores; (2) the desktop-class GeForce GTX 650 Ti with 768 SIMD cores; and (3) the high-performance Tesla K40 GPU accelerator with 2880 SIMD cores. All the platforms run Ubuntu Linux version 14.04.

Multithreaded vectorized CPU execution. To compare TABLA with the CPU platforms, we use optimized open-source multithreaded implementations. We use Liblinear [26] for logistic regression and SVM; MLPACK [27] for recommender sys-



(a) Speedup of Xeon and TABLA in comparison to ARM A15.



(b) Speedup of GPUs and TABLA design in comparison ARM A15.

Figure 7: Speedup of TABLA in comparison to a diverse set of CPU and GPU platforms. The baseline is ARM A15.

Table 4: FPGA platform specifications.

FPGA hardware platform	
Model	Xilinx Zynq ZC702 (Artix-7)
Technology	TSMC 28nm
FPGA Capacity	53K LUTs
	106K Flip-Flops
Peak Frequency	250MHz
BRAM	630 KB
DSP Slices	220 Count of type DSP48E1
MSRP	$129

Table 5: Specifications of the CPU's and GPU's used to evaluate TABLA.

Platform	Cores	Clock (MHz)	Memory (GB)	TDP (W)	Process (nm)	MSRP (USD)
ARM Cortex A15	4+1	2300	2	5	28	$191
Intel Xeon E3-1246 v3	4	3600	16	84	22	$290
Tegra K1 GPU	192	852	2	5	28	$191
NVIDIA GTX650Ti	768	928	1	110	28	$150
Tesla K40	2880	875	12	235	28	$5,499

tems and linear regression; and Caffe [18] for backpropagation. The code is compiled using gcc 4.8 with *-O3 -ftree-vectorize -march=native* flags in order to enable aggressive compiler optimizations and utilize vector instructions. All benchmarks use four threads on ARM and eight threads on Xeon. The ARM CPU does not support simultaneous multithreading (SMT) while the Xeon CPU does. Multithreading support is either implemented using OpenMP (Liblinear) or using OpenBLAS [28] (MLPACK and Caffe). In addition to libraries reported in this paper, we tried a wide spectrum of other available libraries (LibFM [29], Libsvm [30], FANN [31]). However, these libraries provided inferior performance in comparison to the ones presented.

Optimized CUDA implementation for GPU execution. For the GPU platforms, we use highly optimized CUDA implementations from [32], Caffe+cuDNN [18], and LibSVM-GPU [33]. Caffe was configured to use the latest version of Nvidia cuDNN library [34]. The cuDNN library is a dynamic library provided as a binary without source code and is pre-optimized by Nvidia for our target GPUs. For the other benchmarks, we made our best effort to hand-tune their CUDA code for each GPU platform and optimized the number of blocks and number of threads-per-block. Moreover, all of the benchmarks are compiled separately for each GPU using target-specific flags.

Execution time measurements. The execution time for both CPU and GPU implementations are obtained by measuring the wall clock time, averaged over 100 runs. The CPU and GPU execution times are compared with the FPGA runtime obtained from the hardware counters synthesized on the programmable logic.

7.1.3 Power Measurements

We employ a variety of strategies in order to measure each benchmark's power consumption on different platforms.

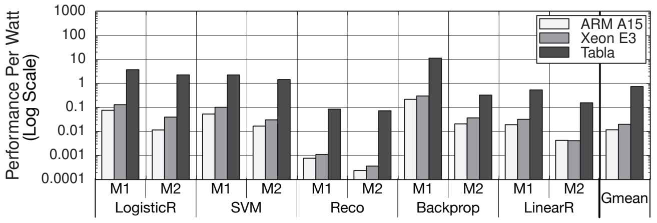
Power measurements using vendor libraries. For Xeon E3, we utilize the Intel Running Average Power Limit (RAPL) energy consumption counters available in the Linux kernel. For Tesla K40, we use the Nvidia Management Library (NVML) to obtain the average power while running each benchmark. GTX 650 Ti does not support the NVML library; however, GTX 650 Ti and Tesla K40 share the same microarchitecture. Hence, we make a conservative estimation of the GTX650 Ti power consumption by scaling the Tesla K40 measurements using the ratio of the two chips' Thermal Design Powers (TDPs). For each benchmark, we calculate the ratio between the measured power in Tesla and its TDP. We multiply this ratio with the GTX 650 Ti's TDP, and use 95% of the resulting value as its estimated power.

Power measurements in hardware. ARM Cortex A15 CPU and Tegra K1 GPU are a part of Jetson TK1's development board. Jetson TK1 does not provide a software mechanism to measure energy consumption. Therefore, we use the Keysight E3649A Programmable DC Power Supply to measure its power consumption. During each benchmark execution, we subtract the idle power from the obtained readings. The ZYNQ platform uses Texas Instruments UCD9240 power supply controllers that enable us to measure the power consumption from the power plane of the board.

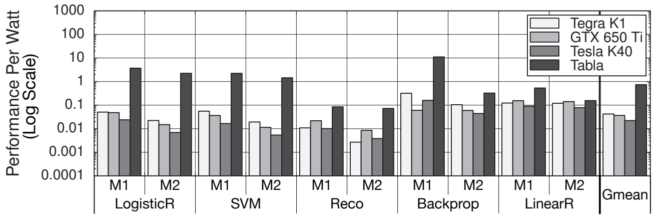
7.2 Experimental Results

7.2.1 Performance Comparison

Comparison with CPUs. Figure 7a shows the speedup of TABLA-generated FPGA accelerators and the Xeon E3 CPU when compared with the ARM A15 CPU. ARM is the baseline in all the speedup graphs. Henceforth, we refer to the TABLA-generated accelerators as Tabla. On average, Tabla outperforms ARM by 19.4× and Xeon by 2.9×. Furthermore, the high-performance Xeon is 6.7× faster than the low-power ARM. Tabla outperforms both the CPUs since our careful compiler-architecture co-design alleviates the Von Neumann overhead of instruction fetch, decode, etc. By leveraging static scheduling for both computation and memory accesses, the accelerators can carry out the calculations efficiently. In comparison to ARM, the performance improvements for Tabla range from 4× to 115×. This variation in performance benefits comes from the disparity in the model topology, which in turn leads to different levels of parallelism in the DFG. For instance, the relatively large model topology of Reco M2 provides greater opportunities for parallelism that can be exploited by



(a) Performance-per-Watt of ARM, Xeon and TABLA.



(b) Performance-per-Watt of Tegra, GTX 650 Ti, Tesla and TABLA

Figure 8: Comparison of Performance-per-Watt between CPUs, GPUs and TABLA.

the accelerator and provides the maximum speedup of 115×. On the other hand, for the Backprop M2 benchmark, Tabla provides the least speedup of 4× in comparison to ARM and a slowdown of 56% in comparison to Xeon. However, Tabla is still faster than Xeon by 2.5× for Backprop M1 benchmark. In the backpropagation algorithm, there are several dependent operations that lead to serialization of the computation and limit the opportunities for parallelism. These dependencies are not as limiting in the smaller model Backprop M1; however, their effect exacerbates as the model topology grows (Backprop M2). To overcome this challenge, one possible solution would be to simultaneously run more iterations of the gradient function over different training input vectors. Similar optimizations can be integrated into TABLA's framework owing to its cross-layer nature. Ultimately, such optimizations would be applied during the compilation stage in accordance to the DFG of the learning task in order to exploit more resources that are available on the target FPGA platform.

Comparison with GPUs. Figure 7b depicts the speedups with different GPU platforms and Tabla. As mentioned before, ARM is the baseline. As the results show, Tesla provides an average speedup of 22.8× , followed by GTX 650 Ti with an average speedup of 16.8× . Finally, the low power Tegra K1 GPU only provides a speedup of 3.3× over ARM. In comparison to Xeon, Tesla and GTX 650 Ti provide an average speedup of 3.42× and 2.51× , respectively. However, Tegra 2.04× is slower than Xeon. These results can be attributed to the fact that Tesla (TDP of 235W), GTX 650 Ti (TDP of 110W), and Tegra (TDP of 5W) are GPUs with decreasing order of TDP (power envelope). The higher power consumption of Tesla and GTX 650 Ti justify their higher speedup numbers. On the other hand, even though Tabla operates in a lower power budget (TDP of 2W), it marginally outperforms GTX 650 Ti by 16%. However, Tabla is surpassed by Tesla with a margin of 18%. Tabla is 5.9× faster in comparison to Tegra. These results show that Tabla either follows or outperforms the GPU platforms due to its specialized hardware design tailored for a particular learning task while operating under a low power budget of 2W. For benchmarks LogisticR M1, LogisticR M2, SVM M1, SVM M2, and Backprop M1, Tabla shows higher performance than Tesla. These benchmarks have relatively small topologies and hence the coarse-grained parallelism that can be exploited by the GPUs is fairly limited. On the other hand, the TABLA-generated accelerators are able to take advantage of the available fine-grained parallelism. As the size of the topology increases, GPUs tend to exhibit higher speedups due to the availability of more computation that can be parallelize. However, GPUs that outperform Tabla require significantly higher power.

7.2.2 *Performance-per-Watt Comparison*

The performance benefits vary significantly across the platforms as these platforms occupy different points in the performance-power design space. To understand the performance benefits for fixed energy efficiency, we use the Performance-per-Watt as a unifying metric to compare these platforms.

Comparison with CPUs. Figure 8a compares the Performance-per-Watt for ARM A15, Xeon E3 and Tabla. On average, Tabla achieves 62.7× and 37.4× higher Performance-per-Watt over ARM and Xeon, respectively. Xeon provides 67% higher Performance-per-Watt than ARM. Even though ARM is a low power CPU, Xeon shows better Performance-per-Watt due to its significantly higher performance.

Comparison with GPUs. Figure 8b illustrates the Performance-per-Watt for the GPU platforms. Tabla provides 17.57×, 20.2× and 33.4× higher Performance-per-Watt in comparison to Tegra, GTX 650 Ti, and Tesla, respectively. In comparison to Tesla, Xeon achieves just 11% higher Performance-per-Watt, however, Tesla provides much higher performance gains. Similarly, Tegra, GTX 650 Ti, and Tesla provide 3.57×, 3.1×, and 1.88× higher Performance-per-Watt than ARM, respectively, while achieving higher speedup gains. The TABLA-generated FPGA accelerators close this performance gap to a large extent and provide much

Table 6: Resource utilization on the FPGA for each benchmark.

Benchmark		LUT (Total Available: 53200)		Block RAM (Total Available: 630KB)		Flip-Flops (Total Available: 106400)		DSP Slices (Total Available: 220)	
Name	**Model**	**Total Used**	**Utilization**	**Total Used (B)**	**Utilization**	**Total Used**	**Utilization**	**Total Used**	**Utilization**
LogisticR	**M1**	1873	3.52%	440	0.07%	1230	1.16%	32	14.55%
	M2	3843	7.22%	1612	0.25%	2446	2.30%	64	29.09%
SVM	**M1**	1326	2.49%	440	0.07%	1206	1.13%	32	14.55%
	M2	3296	6.20%	1612	0.25%	2422	2.28%	64	29.09%
Reco	**M1**	1326	2.49%	115504	17.90%	1206	1.13%	32	14.55%
	M2	3296	6.20%	439652	68.15%	2422	2.28%	64	29.09%
Backprop	**M1**	1916	3.60%	400	0.06%	648	0.61%	16	7.27%
	M2	7672	14.42%	262148	40.64%	2602	2.45%	64	29.09%
LinearR	**M1**	3296	6.20%	444	0.07%	2422	2.28%	64	29.09%
	M2	3296	6.20%	6284	0.97%	2422	2.28%	64	29.09%

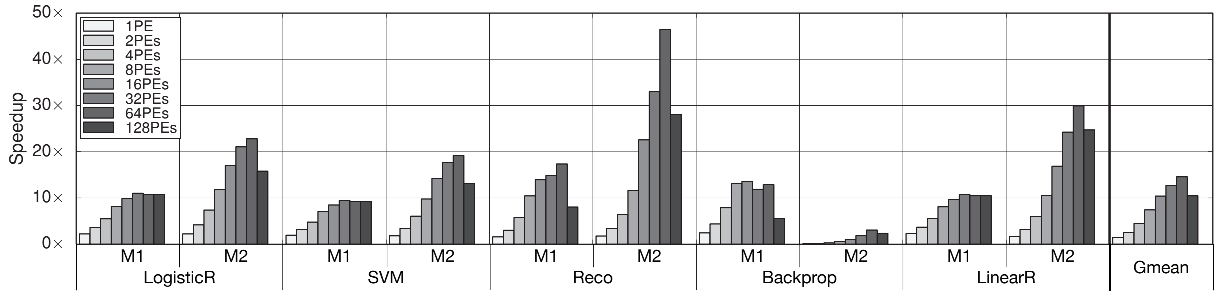


Figure 9: Speedup change for varying number of PEs in the design with ARM CPU as the baseline

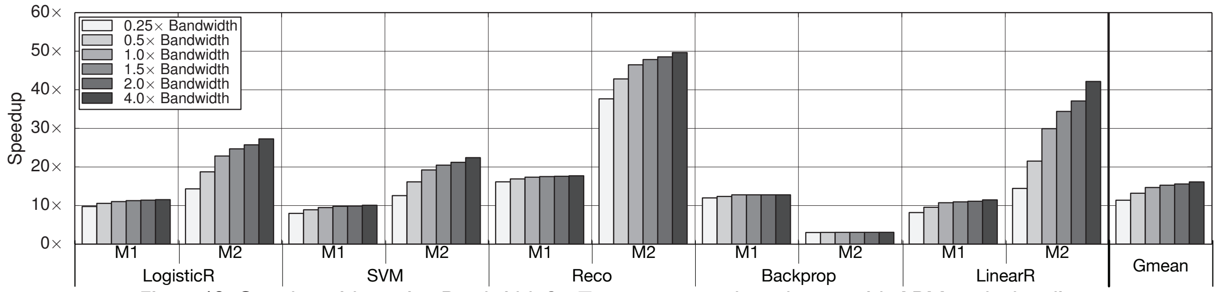


Figure 10: Speedup with varying Bandwidth for TABLA generated accelerator with ARM as the baseline

higher efficiency and operate with a lower power budget. In any case, GPUs can be explored as an alternative back-end for TABLA.

As the results show, TABLA framework provides significant speedup over the multicore CPUs and higher efficiency over the many-core GPUs. These results can be attributed to the fact that TABLA streamlines the execution by generating a static schedule even for memory accesses. TABLA's compiler also tries to maximize the data transfer bandwidth by marshaling the data. It carefully lays out the parameters in local memory and data in external memory in order to reduce the accesses to the external memory.

7.2.3 *Area and FPGA Utilization*

Table 6 shows the resource utilization for different components on the FPGA for each learning task. Backprop M1 utilizes the least area among all the learning algorithms as it has a relatively small model and requires only 16 PEs for its default configuration. On the other hand, Reco M1, Reco M2, and Backprop M2 utilize a larger area in their default configuration. These learning tasks also occupy more BRAM (FPGA Memory Slices) to accommodate the large number of parameters that need to be stored in the accelerator.

7.3 Design Space Exploration

Number of PEs per PU. During the development of the template-based designs, we perform a design space exploration to find the PE and PU configuration that provides the highest frequency while maintaining parallelism within each PU. Empirically, a PU design with eight PEs strikes a balance between frequency and intra-PU parallelism. Note that this design space exploration is not the responsibility of the programmer but part of TABLA.

Number of processing engines. While the number of PEs in each PU is fixed for the target FPGA, TABLA's design builder determines the number PUs (total number of PEs) in accordance with the algorithm's DFG. We perform a design space exploration by varying the total number of PEs. When the number of PEs exceeds eight, they are grouped into a PU with eight PEs each. Figure 9 shows the effect of this sweep on the speedup results. The baseline is the A15 ARM multicore CPU. As expected, the initial increase in the number of PEs leads to a linear increase in speedup. However, beyond a certain number of PEs we either observe diminishing returns or a decrease in speedup. Since the available parallelism in the algorithms is limited, increasing the number of PEs beyond a point leads to underutilization of the added PEs. For instance, for LogisticR M1, a maximum of 54 operations can be performed in parallel. Therefore, providing more than 54 PEs is inconsequential. In some cases such as LogisticR M1, increasing the number of PEs beyond 32 leads to a decrease in the speedup. When the number of PEs is greater than 32, the operational frequency decreases due to the requirement of a wider global bus. Therefore, in this case (LogisticR M1), adding more PEs does not improve performance due to the lack parallelism but rather decreases the speedup due to slower hardware. The last column in Table 3 shows the total number of PEs for each benchmark that are grouped in PUs with 8 PEs each.

Bandwidth sensitivity. Machine learning algorithms are both compute and data intensive tasks. We design the accelerators to exploit the fine-grained parallelism in the computational component of the algorithm. In addition to the compute units, the training data is streamed to the PEs from the external memory while the PEs store the model parameters locally. The AXI interfaces offer a fixed bandwidth for the training data transfer. We perform a speedup sensitivity analysis while varying the bandwidth between external memory and the accelerator. We perform the bandwidth sweeps using a cycle-accurate simulator, which is validated against the hardware. Figure 10 shows the speedup for each benchmark when the bandwidth varies from 0.25× to 4× of the default bandwidth. The baseline is ARM. The bandwidth can be a bottleneck at low values such as 0.25× of the default bandwidth. As the bandwidth increases, the speedup starts to increase but we observe diminishing returns after a certain point since computation dominates the execution time. By providing a bandwidth that is 4× the default value, the performance only improves by 60%. This limited improvement is in part due to the fact that the model compiler stores the most frequently accessed data (the model parameters and intermediate results) in the PE's local buffers. This limits the accesses to the external memory and attenuates the effects of external memory bandwidth on performance.

8 Related Work

There have been several proposed architectures that accelerate machine learning algorithms [23, 32, 35–47]. However, TABLA fundamentally differs from these works, as it is not an accelerator. TABLA framework generates accelerators for an important class of machine learning algorithms, which can be expressed as stochastic optimization problems. TABLA uses the commonalities across a wide range of learning algorithms and provides a high-level abstraction for programmers to utilize FPGAs as the accelerator of choice for machine learning algorithms without exposing the details of hardware design.

TABLA also automatically incorporates stochastic gradient descent solver into its learning accelerators. There have been past proposals that focus solely on accelerating gradient descent [48] and conjugate gradient descent [48–51] solvers. The most recent work [48] focuses merely on designing hardware units for different linear algebra operations that are used in gradient descent and conjugate gradient solvers. However, these works do not specialize their architectures for machine learning algorithms or any specific objective function. Moreover, they neither provide domain-specific programming models nor generate accelerators.
Machine learning accelerators. There have been several successful works that focus on accelerating a single or a range of fixed learning tasks. Several efforts have focused on designing accelerators for a specific algorithm (K-Nearest Neighbor) [35, 36, 52]. Others propose accelerator designs for k-Means [37–39], support vector machines (SVM) [40, 41], deep neural networks [44–46], and multilayer perceptrons [23, 47] However, all these efforts are focused on accelerating a particular learning algorithm.

Several inspiring works propose accelerator designs that support a number of learning algorithms [32, 42, 43]. MAPLE [42, 43] profiles five learning algorithms, identifies their compute-intensive kernels, and devises an accelerator that efficiently executes the kernels. PuDianNao [32] provides an ASIC design that can accelerate seven different learning algorithms. We, on the other hand, delve into the theory of machine learning, identify the theoretical commonalities across a wide range of learning algorithms, devise an abstraction between hardware and software, and provide a unified framework that generates accelerators.
FPGA as an acceleration platform. FPGAs have gained popularity due to their flexibility and capability to provide high execution performance by exploiting copious fine-grained irregular parallelism in the applications. Several works [35, 37, 40, 41, 52–56] utilize FPGAs to accelerate a diverse set of workloads, validating the efficacy of FPGAs. LINQits [57] provides a template architecture for accelerating database queries. The work by King et al. [58] uses Bluespec to automatically generate a hardware-software interface for the applications partitioned for hardware acceleration and software execution. The work by Putnam et al. [7], designs an FPGA fabric for accelerating ranking algorithms in the Bing server. This FPGA-based fabric in deployed with 1632 servers. TABLA provides an opportunity to utilize this integrated reconfigurable fabric for machine learning algorithms. Conclusively, TABLA is a comprehensive solution–from programming language down to circuit design–that provides a unified abstraction based on the theory of machine learning for accelerating an important class of learning algorithms.

9 Conclusions

Machine learning algorithms are compute-intensive workloads that can benefit significantly from acceleration. FPGAs are an attractive platform for accelerating these important applications. However, FPGA design still requires relatively long development cycles and extensive expertise in hardware design. This paper described TABLA that aims to bridge the gap between the machine learning algorithms and the FPGA accelerators. TABLA dives into the theory of machine learning and takes advantage of the insight that a large class of learning algorithms can be expressed as stochastic optimization problems. TABLA leverages stochastic gradient descent as the abstraction between hardware and software to automatically generate accelerators for this class of statistical machine learning algorithms. We used TABLA to generate accelerators for a variety of learning algorithms targeting an off-the-shelf FPGA platform, Xilinx Zynq. In comparison to a multicore Intel Xeon with vector execution, the TABLA-generated accelerators deliver an average speedup of 2.9×. Compared with the high-performance Tesla K40 GPU accelerator, TABLA achieves 33.4× higher Performance-per-Watt. These gains are achieved while the programmers write less than 50 lines of code. These results suggest that TABLA takes an effective step towards making FPGAs widely available to the machine learning developers. We have made TABLA publicly available (`http://act-lab.org/artifacts/tabla`) in order to facilitate research and development in using FPGAs for learning.

10 Acknowledgements

We thank the anonymous reviewers for their insightful comments and feedback. We thank Chenkai Shao, Balaji Chandrasekaran, and Thierry Moreau. This work was supported in part by a Qualcomm Innovation Fellowship, NSF awards CCF #1553192 and CNS #1526211, Semiconductor Research Corporation contracts #2014-EP-2577 and #2015-TS-2636, and a gift from Google.

References

[1] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, July–Aug. 2011.
[2] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *ISCA*, 2011.
[3] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: Reducing the energy of mature computations. In *ASPLOS*, 2010.
[4] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9, October 1974.
[5] Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. CPU DB: Recording microprocessor history. *ACM Queue*, 10(4):10:10–10:27, April 2012.
[6] John Gantz and David Reinsel. Extracting value from chaos.
[7] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James R. Larus, Eric Peterson, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *ISCA*, June 2014.
[8] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam.

Dynamically specialized datapaths for energy efficient computing. In *HPCA*, 2011.

[9] Ganesh Venkatesh, John Sampson, Nathan Goulding, Sravanthi Kota Venkata, Steven Swanson, and Michael Taylor. QsCores: Trading dark silicon for scalable energy efficiency with quasi-specific cores. In *MICRO*, 2011.

[10] Shantanu Gupta, Shuguang Feng, Amin Ansari, Scott Mahlke, and David August. Bundled execution of recurring traces for energy-efficient general purpose processing. In *MICRO*, 2011.

[11] Johann Hauswald, Michael A. Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ron Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, and Jason Mars. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *ASPLOS*, 2015.

[12] Scott Sirowy and Alessandro Forin. Where's the beef? why FPGAs are so fast. Technical Report MSR-TR-2008-130, Microsoft Research, September 2008.

[13] Xilinx. Zynq-7000 all programmable soc, 2014.

[14] Intel Corporation. Disrupting the data center to create the digital services economy.

[15] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[16] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. Towards a unified architecture for in-RDBMS analytics. In *Proceedings of the International Conference on Management of Data*, SIGMOD '12, 2012.

[17] David C Ku and Giovanni De Micheli. *High level synthesis of ASICs under timing and synchronization constraints*. Kluwer Academic Publishers, 1992.

[18] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[19] M. Lichman. UCI machine learning repository, 2013.

[20] Iván Cantador, Peter Brusilovsky, and Tsvi Kuflik. Second workshop on information heterogeneity and fusion in recommender systems (HetRec). In *Proceedings of the ACM conference on Recommender systems*, RecSys 2011, 2011.

[21] Grouplens. Movielens dataset.

[22] Kamil A. Grajski. Neurocomputing, using the MasPar MP-1. In K. W. Przytula and V. K. Prasnna, editors, *Parallel Digital Implementations of Neural Networks*, chapter 2, pages 51–76. Prentice-Hall, 1993.

[23] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *MICRO*, 2012.

[24] Yann Lecun, LÃl'on Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

[25] Nvidia. Jetson. `http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html`, 2015.

[26] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear: A library for large linear classification. *J. Mach. Learn. Res.*, 9:1871–1874, June 2008.

[27] Ryan R. Curtin, James R. Cline, Neil P. Slagle, William B. March, P. Ram, Nishant A. Mehta, and Alexander G. Gray. MLPACK: A scalable C++ machine learning library. *Journal of Machine Learning Research*, 14:801–805, 2013.

[28] Zhang Xianyi, Wang Qian, and Zhang Yunquan. Model-driven level 3 BLAS performance optimization on loongson 3A processor. In *ICPADS*, 2012.

[29] Steffen Rendle. Factorization machines with libFM. *ACM Trans. Intell. Syst. Technol.*, 3(3):57:1–57:22, May 2012.

[30] Chih-Chung Chang and Chih-Jen Lin. Libsvm: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.*, 2(3):27:1–27:27, May 2011.

[31] S. Nissen. Implementation of a fast artificial neural network library (FANN). Technical report, Department of Computer Science University of Copenhagen (DIKU), 2003. http://fann.sf.net.

[32] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. PuDianNao: A polyvalent machine learning accelerator. In *ASPLOS*, 2015.

[33] Andreas Athanasopoulos, Anastasios Dimou, Vasileios Mezaris, and Ioannis Kompatsiaris. GPU acceleration for support vector machines. In *12th International Workshop on Image Analysis for Multimedia Interactive Services*, 2011.

[34] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, 2014.

[35] Ioannis Stamoulias and Elias S. Manolakos. Parallel architectures for the knn classifier – design of soft IP cores and FPGA implementations. *ACM Trans. Embed. Comput. Syst.*, 13(2):22:1–22:21, September 2013.

[36] E.S. Manolakos and I. Stamoulias. IP-cores design for the knn classifier. In *ISCAS*, May 2010.

[37] H.M. Hussain, K. Benkrid, H. Seker, and A.T. Erdogan. FPGA implementation of K-means algorithm for bioinformatics application: An accelerated approach to clustering microarray data. In *AHS*, June 2011.

[38] Tsutomu Maruyama. Real-time K-Means clustering for color images on reconfigurable hardware. In *ICPR*, pages 816–819, 2006.

[39] A.Gda.S. Filho, A.C. Frery, C.C. de Araujo, H. Alice, J. Cerqueira, J.A. Loureiro, M.E. de Lima, Mdas.G.S. Oliveira, and M.M. Horta. Hyperspectral images clustering on reconfigurable hardware using the k-means algorithm. In *SBCCI*, Sept 2003.

[40] M. Papadonikolakis and C. Bouganis. A heterogeneous FPGA architecture for support vector machine training. In *FCCM*, May 2010.

[41] S. Cadambi, I. Durdanovic, V. Jakkula, M. Sankaradass, E. Cosatto, S. Chakradhar, and H.P. Graf. A massively parallel fpga-based coprocessor for support vector machines. In *FCCM*, April 2009.

[42] A. Majumdar, S. Cadambi, and S.T. Chakradhar. An energy-efficient heterogeneous system for embedded learning and classification. *Embedded Systems Letters, IEEE*, 3(1):42–45, March 2011.

[43] Abhinandan Majumdar, Srihari Cadambi, Michela Becchi, Srimat T. Chakradhar, and Hans Peter Graf. A massively parallel, energy efficient programmable accelerator for learning and classification. *ACM Trans. Archit. Code Optim.*, 9(1):6:1–6:30, March 2012.

[44] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun. NeuFlow: A runtime reconfigurable dataflow processor for vision. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, pages 109–116, June 2011.

[45] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ASPLOS*, 2014.

[46] A.A. Maashri, M. DeBole, M. Cotter, N. Chandramoorthy, Yang Xiao, V. Narayanan, and C. Chakrabarti. Accelerating neuromorphic vision algorithms for recognition. In *DAC*, June 2012.

[47] Thierry Moreau, Mark Wyse, Jacob Nelson, Adrian Sampson, Hadi Esmaeilzadeh, Luis Ceze, and Mark Oskin. SNNAP: Approximate computing on programmable socs via neural acceleration. In *HPCA*, 2015.

[48] D. Kesler, B. Deka, and R. Kumar. A hardware acceleration technique for gradient descent and conjugate gradient. In *SASP*, June 2011.

[49] Antonio Roldao and George A. Constantinides. A high throughput FPGA-based floating point conjugate gradient implementation for dense matrices. *ACM Trans. Reconfigurable Technol. Syst.*, 3(1):1:1–1:19, January 2010.

[50] G.R. Morris, V.K. Prasanna, and R.D. Anderson. A hybrid approach for mapping conjugate gradient onto an fpga-augmented reconfigurable supercomputer. In *FCCM*, April 2006.

[51] D. DuBois, A. DuBois, T. Boorman, C. Connor, and S. Poole. An implementation of the conjugate gradient algorithm on fpgas. In *FCCM*, April 2008.

[52] Yao-Jung Yeh, Hui-Ya Li, Wen-Jyi Hwang, and Chiung-Yao Fang. FPGA implementation of kNN classifier based on wavelet transform and partial distance search. In *SCIA*, 2007.

[53] Andrew R. Putnam, Dave Bennett, Eric Dellinger, Jeff Mason, and Prasanna Sundararajan. CHiMPS: A high-level compilation flow for hybrid CPU-FPGA architectures. In *FPGA*, 2008.

[54] Yi Shan, Bo Wang, Jing Yan, Yu Wang, Ningyi Xu, and Huazhong Yang. Fpmr: Mapreduce framework on fpga. In *FPGA*, 2010.

[55] Jeremy Fowers, Kalin Ovtcharov, Karin Strauss, Eric Chung, and Greg Stitt. A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication. In *FCCM*. IEEE, May 2014.

[56] Eric S. Chung, James C. Hoe, and Ken Mai. CoRAM: An in-fabric memory architecture for fpga-based computing. In *FPGA*, 2011.

[57] Eric S. Chung, John D. Davis, and Jaewon Lee. LINQits: Big data on little clients. In *ISCA*, 2013.

[58] M. King, A. Khan, A. Agarwal, O. Arcas, and Arvind. Generating infrastructure for FPGA-accelerated applications. In *FPL*, Sept 2013.