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

Divya Mahajan Jongse Park Emmanuel Amaro Hardik Sharma
Amir Yazdanbakhsh Joon Kim Hadi Esmaeilzadeh
Georgia Institute of Technology
{divya_mahajan, jspark, amaro, hsharma, a.yazdanbakhsh, jkim796}@gatech.edu hadi@cc.gatech.edu

ABSTRACT

A growing number of commercial and enterprise systems increasingly rely on compute-intensive machine learning algorithms. While the demand for these compute-intensive applications is growing, the performance benefits from general-purpose platforms are diminishing. To accommodate the needs of machine learning algorithms, Field Programmable Gate Arrays (FPGAs) provide a promising path forward and represent an intermediate point between the efficiency of ASICs and the programmability of general-purpose processors. However, acceleration with FPGAs still requires long design cycles and extensive expertise in hardware design. To tackle this challenge, instead of designing an accelerator for machine learning algorithms, we develop 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 stochastic optimization problems. Therefore, a learning task becomes solving an optimization problem using stochastic gradient descent that minimizes an objective function. The gradient solver is fixed while the objective function changes for different learning algorithms. TABLA provides a template-based framework for accelerating this class of learning algorithms. With TABLA, the developer uses a high-level language to *only* specify the learning model as the gradient of the objective function. TABLA then automatically generates the synthesizable implementation of the accelerator for FPGA realization.

We use TABLA to generate accelerators for ten different learning task that are implemented on a Xilinx Zynq FPGA platform. We rigorously compare the benefits of the FPGA acceleration to both multicore CPUs (ARM Cortex A15 and Xeon E3) and to many-core GPUs (Tegra K1, GTX 650 Ti, and Tesla K40) using real hardware measurements. TABLA-generated accelerators provide 15.0× and 2.9× average speedup over the ARM and the Xeon processors, respectively. These accelerator provide 22.7×, 53.7×, and 30.6× higher performance-per-Watt compare to Tegra, GTX 650, 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 mobile health monitoring, social networking, e-commerce, targeted advertising, and financial analysis, increasingly rely on Machine Learning (ML) techniques. In fact, the advances in machine learning are changing the landscape of computing towards a more personalized and targeted experience for the users. For instance, services that provide personalized health-care and targeted advertisement are prevalent or are on the horizon. Machine learning algorithms are among the computationally intensive workloads. Specifically, learning a model from data requires ample 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 increases, the benefits from general-purpose computing are diminishing [1, 2]. As shown in the Dark Silicon study [2] and others corroborate [1, 3], with the effective end of Dennard scaling [4], CMOS scaling is no longer providing performance and efficiency gains that are commensurate with the transistor density increases [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 such an overwhelming level that is beyond the capabilities of current computing systems to match [6].

As a result, both the industry and the research community are increasingly focusing on programmable accelerators, which can provide large gains in efficiency and performance by restricting the workloads [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 parts that incorporate general purpose cores with programmable logic are beginning to appear [13, 14]. For instance, Microsoft employs FPGAs to accelerate their Bing search service [7]. This increasing availability of FPGAs for acceleration and their flexibility makes them an attractive platform for accelerating machine learning 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 relatively long even for experts [7]. This paper aims to tackle this challenge for an important class of machine learning algorithms. To this end, we develop TABLA, a template-based solution – from circuit to programming model – for using FPGAs to accelerate statistical machine learning algorithms. The objective of our solution is to devise the necessary programming abstractions and automated frameworks that are 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 learning algorithms.

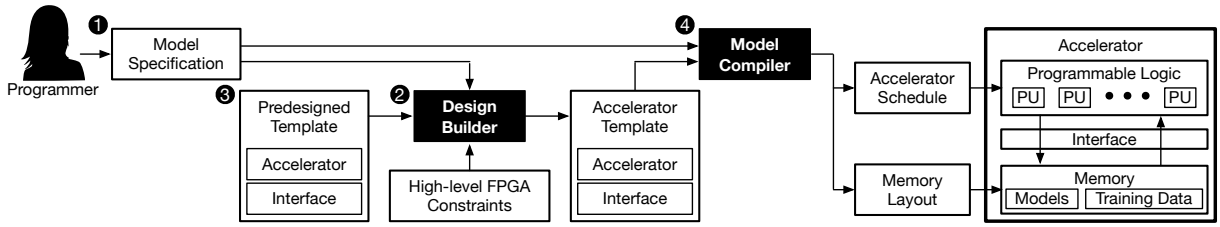


Figure 1: Overview of the workflow with Tabla. The programmer only provides the gradient of the objective function representing the learning algorithm in Tabla's high-level programming language. Tabla is comprised of two major components: (1) the design builder that automatically generates the synthesizable Verilog of the accelerator; and (2) the model compiler that generates the execution schedule for the accelerator. The design builder automatically generates the synthesizable code from a set of predesigned templates.

When developing TABLA, we leveraged the insight that many learning algorithms can be expressed as stochastic optimization problems [15]. Examples of such learning models are support vector machines, logistic regression, least square models, back-propagation, conditional random fields, recommender systems, Kalman filters, linear and nonlinear regression models, and soft-max functions. These types of learning models can be optimized using stochastic gradient descent [16]. That is, the learning task becomes solving an optimization problem using stochastic gradient descent that iterates over the training data and minimizes an objective function. Although the stochastic gradient descent solver is mostly fixed across different learning algorithms, the objective function varies. Therefore, the accelerator for these learning tasks can be implemented as a template design, uniform across a set of machine learning algorithms. This template design comprises the general framework for the stochastic gradient descent optimization.

To be able to specialize the template design for a specific learning task, a hardware block implementing the gradient of the objective function for the particular algorithm needs to be designed and integrated. TABLA provides a template-based framework to automatically generate the hardware block which implements the gradient of the objective function. Therefore, with TABLA, the developer only needs to specify the learning model as the gradient of the particular objective function. 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 while considering high-level design parameters of the target FPGA.

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

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

(3) We used TABLA to generate accelerators for five different learning algorithms – logistic regression, SVM, recommender systems, backpropagation, and linear regression – each with two different topologies. We implemented these accelerators on a Xilinx Zynq FPGA platform. We use TABLA to generate ten different accelerators for ten different learning task that are implemented on a Xilinx Zynq FPGA platform. We rigorously compare the benefits of the FPGA acceleration to both multicore CPUs (ARM Cortex A15 and Xeon E3) and to many-core GPUs (Tegra K1, GTX 650 Ti, and Tesla K40) using real hardware measurements. TABLA-generated accelerators provide 15.0× and 2.9× average speedup over the ARM and the Xeon processors, respectively. These accelerator provide 22.7×, 53.7×, and 30.6× higher performance-per-Watt compared to Tegra, GTX 650, and Tesla, respectively. These benefits are achieved while the programmer write less than 50 lines of code.

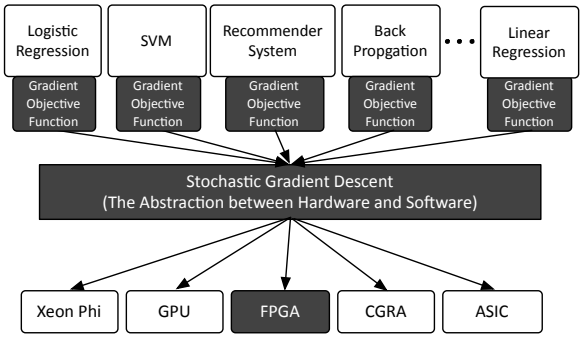


Figure 2: Tabla leverages stochastic gradient descent as an abstraction between the hardware and software to create a unified framework for accelerating machine learning algorithms. The highlighted blocks are the focus of this work.

These results suggests 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 a precursor to the prediction phase generates a model that maps one or more inputs (independent variables) onto one or more outputs (dependent variables). This generated model is used in the prediction phase to predict the dependent variable for a new unseen input. 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 to accelerate the learning phase of a class of machine learning algorithms. Figure 1 illustrates an overview of TABLA and its workflow. Below, we briefly discuss each component of TABLA.

❶ High-level programming model. TABLA provides a high-level programming model that enables the programmers to specify the gradient of the objective function that captures the learning algorithm. TABLA focuses on learning algorithms that can be implemented using stochastic gradient descent, therefore, the gradient function is sufficient to generate the entire accelerator design.

As TABLA's template, i.e. the stochastic gradient descent is uniform across a range of ML algorithms, this programming abstraction requires the programmer to only provide the gradient of the objective function. The programmer also provides the initial and meta parameters of the learning algorithm, such as learning rate.

❷ **Design builder.** After the programmer provides the gradient of the objective function, one of the major components of TABLA, named the *design builder*, automatically generates the accelerator and its interfacing logic. The design builder uses a predefined set of accelerator templates to generate the accelerator. The output of the design builder is a set of synthesizable Verilog codes that concretely implement the accelerator. The inputs to the design builder are the (1) gradient function, (2) a high-level specification of the target FPGA (number of hard DSP slices, number of hard SRAM structures (Block RAMs), the capacity of each Block RAM, number of Block RAM read/write ports, and off-chip communication bandwidth), (3) a predesigned set of accelerator templates in Verilog.

❸ **Predesigned template.** The design builder generates the accelerator design from a *predesigned template*. This template is generic and uniform across a large class of stochastic machine learning algorithms and supports all the language constructs that are defined in TABLA's programming language. The template provides a general structure for the accelerator without making it specific to a certain algorithm or accelerator specification. The unified template also contains a section that implements the stochastic gradient descent, which is uniform across all the target machine learning algorithms. These predefined templates are designed by expert hardware designers and comprise of both the accelerator and the interfacing logic that connects the accelerator to the rest of the system (e.g., the memory).

❹ **Model compiler.** Another component of TABLA is the *model compiler* that statically generates an execution schedule for the accelerator. Statically generating a schedule for the accelerator significantly simplifies the hardware. The inputs to the model compiler are the structure of the accelerator and 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 the 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 will determine the layout of parameters in the memory and streamlines the interfacing logic that communicates with the memory. The model compiler also generates the schedule for the memory interface.

As Figure 2 depicts, TABLA can potentially target different platforms, including Xeon Phi, GPUs, FPGAs, CGRAs, and ASIC. To support each of these target platforms, new backends need to be developed for each target. 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 discusses the theoretical foundation of stochastic gradient descent.

3 Background on Stochastic Gradient Descent

Stochastic gradient descent (SGD) forms the abstraction between hardware and software for TABLA that generates machine learning accelerators and therefore forms the template-base of TABLA. SGD is an optimization algorithm that aims to find the set of parameters that minimize a function. This function is more commonly referred to as the objective or the cost function.

Objective Function. Each machine learning task in our target class is characterized by its objective function. The objective function has a set of parameters that are learned in accordance to the training data such that the machine learning algorithm can make data-driven predictions or decisions on new unseen data. The objective function is a cost function that is defined over the training data for a given set of parameters. The cost function quantifies the error between the predicted value of the output and the actual output value corresponding to an input dataset. The ML algorithm learns the model by solving an optimization problem that minimizes the objective/cost function (or prediction error) over the entire training data as shown in Equation 1. In the equation, w^t represents the parameters of the model (at iteration t) over which the objective function is to be minimized for the training data (i). Here, x_i corresponds to the (i)th input element provided by the user and $f(w_i^t x_i)$ is the objective/cost function (prediction error over the training data). This objective function is minimized using the stochastic gradient descent optimization algorithms that iterate over the training data. While the stochastic gradient descent is fixed across different learning algorithms, the objective function varies. The Table 1 shows five sample machine learning tasks that can be trained using stochastic gradient descent. Table 1 also presents the objective function corresponding to each machine learning algorithm.

Stochastic Gradient Descent. Stochastic gradient is a derived gradient descent optimization problem that aims to minimize the following objective function:

$$\min_{w^t \varepsilon R} \sum_i f(w_i^t x_i) \tag{1}$$

The objective function to be minimized is defined by a set of parameters. Gradient descent starts with an initial set of parameter values and iteratively moves toward a set of parameter values that minimize the function. This iterative minimization is achieved by taking steps in the decreasing direction of the function's derivative or gradient. Hence the gradient algorithm can be written as:

$$w^{t+1} = w^t - \mu \times \frac{\partial(\sum_i f(w_i^t x_i))}{\partial w^t} \tag{2}$$

As the above equation shows, w^{t+1} goes in the negative direction of $\frac{\partial f}{\partial w}$ with a rate μ. That is, in a single iteration of gradient descent, it calculates the derivative of the objective function over the entire training data and generates the next set of parameters (w^{t+1}) as shown by equation 2. For very large training datasets, the gradient descent can impose a high overhead by iterating over all the data to just generate the next set of parameters. Furthermore, this process is repeated until the function reaches close to its minimum which is tested by convergence algorithms. To avoid this large data overhead, stochastic gradient descent (SGD) is used. SGD is a modification of conventional gradient descent as it divides the objective function into smaller differentiable functions. As it can be seen from Equation 1, the objective function is a summation of a function over all the training data. Instead of taking the derivative of the function calculated over the entire dataset, SGD divides the objective function into smaller functions requiring a single element. Therefore, the gradient of

Table 1: Machine learning algorithms, their objective function and gradient function that have been integrated with Tabla.

Machine Learning Algorithm	Objective Function (f)	Gradient of the Objective Function (∂f)
Logistic Regression	$\sum_i \{ y_i \bullet \log f(w^T x_i) + (1 - y_i) \bullet \log(1 - f(w^T x_i)) \} + \lambda \|\| w \|\|$	$\sum_i \{ (f(w^T x_i) - y_i) \bullet x_i \} + \lambda \bullet w$
Classification (SVM)	$\sum_i \{ 1 - y_i w^T x_i \} + \lambda \|\| w \|\|$	$\sum_i y_i x_i + \lambda \bullet w$
Recommender Systems	$\sum_{i,j} (Y_{ij} - w_j^T x_i)^2 + \lambda \|\| w, x \|\|$	$\sum_{i,j} (w_j^T x_i - Y_{ij}) \bullet x_i + \lambda \bullet w$, $\sum_{i,j} (w_j^T x_i - Y_{ij}) \bullet w_j + \lambda \bullet x$
Backpropogation	$\sum_k \sum_i \{ y_i^{(k)} \bullet \log f(w^T x_i)^{(k)} + (1 - y_i^{(k)}) \bullet \log(1 - f(w^T x_i)^{(k)}) \} + \lambda \|\| w \|\|$	$\sum_k \sum_i (\alpha_k^{(l)} \bullet \delta_k^{(l)})$
Linear Regression	$\sum_i \frac{1}{2} (w^T x_i - y_i)^2 + \lambda \|\| w \|\|$	$\sum_i (w^T x_i - y_i) \bullet x_i + \lambda \bullet w$

the smaller function is only calculated over a single element. The equation for stochastic gradient descent transforms into:

$$w^{t+1} = w^t - \mu \times \frac{\partial f(w_i^t x_i)}{\partial w^t} \quad (3)$$

This step of SGD is looped for all training elements individually until the function converges at its minimum value. SGD typically takes more iterations to converge in contrast to the conventional gradient descent, however, the benefits obtained by avoiding the data access to all the input elements for each iteration is significantly higher than the cost incurred by having more iterations. Using SGD 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 SGD as the abstraction between the software and the hardware for TABLA that generates accelerators as shown in Figure 2. To specialize the template, TABLA framework only requires the programmer to specify the learning model as the gradient of the objective function. The only programming task is to implement this gradient function for the respective algorithm. Our experience shows that this gradient function (as shown in Table 1) can be implemented with less than 50 lines of code for logistic regression, SVM, recommender systems, back-propagation, and linear regression. Other algorithms such as conditional random fields, Kalman filters, portfolio optimization, and least square models can be integrated with TABLA as they can be optimized using SGD. After the programmer provides the gradient of the objective function, TABLA workflow can automatically generate a concrete accelerator for the specific learning task. We now describe the different components of TABLA- programming interface, model compiler and accelerator design in Sections 4, 5 and 6, respectively.

4 Programming Interface

The programmer needs to express different learning algorithms by specifying the gradient of the objective function. This programming interface possesses following properties that enable it to represent a wide range of ML algorithms: (1) It is a high-level interface that enables the representation of ML algorithms in a fashion that is familiar to ML experts and is close to the mathematical model presented in Table 1; (2) It comprises of language constructs and keywords that are commonly seen in several statistical ML algorithms. The programming interface comprises of two types of constructs, data declaration and mathematical operations. Data declarations allow the programmer to express different data types that represent the training data and model parameters. These data elements are used in expressing the gradient function. The mathematical operations enable the programmer to express different mathematical operations used in the gradient of a wide range of objective functions for different machine learning algorithms. Table 2 summarizes these language constructs and their corresponding keywords. Both declaration types are explained in further detailed in sections 4.1 and 4.2

Table 2: Language declarations that allow easy representation of several ML algorithms. Language declarations with their type and classification.

Type	Classification	Language Keywords
Data	Model inputs	model_input
	Model outputs	model_output
	Model Parameters	model
	Gradient of objective function	gradient
	Iterator variable	iterator
Operation	Basic	+,-, <, >, *
	Group	pi, sum, norm
	Non Linear	gaussian, sigmoid, sigmoid_symmteric, log

4.1 Data Declaration

As the name suggests these declarations enable the programmer to specify the different data elements that are 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 them in an ML algorithm. The *model_input* keyword refers to a single input dataset while the *model_output* declaration refers to the corresponding output provided as the training data. Both these data types are inputs to the machine learning task and are read-only while the ML algorithm learns the model. The *model* keyword refers to the model parameters that get updated every iteration in accordance to the gradient of the objective function. Parameters are read-only in the gradient function; however, the SGD algorithm both reads and writes to these parameters.

The *gradient* keyword in our programming interface numerically represents the gradient of the objective function. We have a separate keyword for these gradients, as it is the output of the gradient function and is used as an input to the SGD algorithm. Finally, the *iterator* declaration enables the programmer to declare the dimensions of arrays. For example, in our language a statement Q[j] = A[j] * B[j], means that element j in $\vec{A}$ is multiplied with the element j in $\vec{B}$ for all the values of j. *Iterator* provides a concise way to declare the dimensions of these vectors. That is, iterator j[0:n] declares that j starts from 0 and goes to n. Moreover, iterators also clearly depict the autonomy of operations. For example, A[j] * B[j] can be easily parallelized over all the values of j. In addition to the data, another major component of ML algorithms is the computation performed over the data. Therefore, we define several constructs to support these mathematical operations used in different machine learning algorithms. These mathematical operation constructs are discussed in detail in the next section.

4.2 Mathematical Operations

Mathematical operations allow the programmer to express different operations and functions. These declarations are further subdivided into three categories - basic, group and nonlinear. Each of the declaration types is described in further detail below:

Basic Operations. The basic operations constitute mathematical operations like -,+,<,>,* and require two arguments A and B.

Group Operations. These operations are performed over a group of elements and includes the following operations, $\sum$ (sum),

Π (group multiply), and ‖ ‖ (norm). The *pi* and *sum* keywords take in two arguments while the norm keyword only takes one. Apart from the input values, these operation types require an *iterator* argument to operate on a group of elements.

As these operations process groups of elements, they produce an output with dimension one less than the input dimension. For instance, operating on a vector input produces a scalar output.

Nonlinear Operations. These operations constitute nonlinear functions like Log, Sigmoid, Gaussian, and Sigmoid Symmetric. The output has the same dimensionality as the input as this operation is performed element by element.

Using the data and operation language declarations defined above, programmer can easily represent several statistical ML algorithms. One such example is Logistic Regression. The following code shows how the gradient of logistic regression (given in Table 1) can be expressed in a few lines using TABLA's programming interface.

```
model_input  x[m]; //model input features
model_output y'[n]; //model outputs
model        w[n][m]; //model parameters
gradient     g[n][m]; //gradient

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

//m parallel multiplications followed by
//an addition tree; repeat 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
rg[j][i] = λ * w[i][j]; //n*m parallel multiplications
g[j][i] = g[j][i] + rg[j][i]; //n*m parallel additions
```

The above code shows the simplicity of our programming interface and expresses a complicated mathematical function of the gradient of the objective function into smaller and much simpler operations. In this code the programmer first declares the data types: model_input, model_output, model and the gradient. Then, two iterators *i* and *j* are declared as the model values are two dimensional. Next, operations are performed over the declared data types beginning with the *sum* operation. This operation performs multiplication *x[i] * w[j][i]* and adds up all the multiplication results into a single result (s[j]) in the *i* dimension assuming a constant *j*. On the other hand, the left hand side of the statement *s[j]* shows that the summation operation is repeated n times using the *j* iterator. By following a similar concept where the iterator on the LHS of the equation signifies a loop over that iterator variable, other operations are also performed. Finally the result generated by this code is the gradient for the given model_input, model_output and model.

Several ML algorithms can be represented using the above mentioned language declarations. The simplicity of these declarations to express mathematical models makes the programming interface easy to use. Furthermore, the programming interface can be extended to incorporate more declarations so as to accommodate the representation of an even wider range of statistical ML algorithms.

Although MATLAB and R can also be used to represent the same ML algorithms, we designed and used TABLA's in-house programming interface due to the following reasons: (1) easier representation of gradient functions using the common mathematical constructs used in ML; (2) ease in identifying parts of code that can be made parallel; (3) convenient conversion of gradient function into the final hardware design using the model compiler described in the Section 5;

In the next section we detail the flow of the model compiler which illustrates how the gradient of the objective function provided by the programmer using our language declarations can be appended with the stochastic gradient descent. Furthermore, the model compiler also has the responsibility of converting the unified objective function gradient and SGD into a data-flow graph and finally into a schedule that can be mapped onto the hardware design.

5 Model Compiler for TABLA

After the programmer provides the gradient of the objective function, TABLA's model compiler first integrates this objective function with the stochastic gradient descent. To generate a concrete accelerator, the model compiler then generates a dataflow graph that can be mapped and scheduled on hardware. Dataflow graphs (DFGs) are intermediate representations that can be translated into the accelerator and its execution schedule. Thus, the final phase of compilation is the scheduling phase in which the compiler generates a static schedule for the learning task that is represented by a dataflow graph.

5.1 Integration of Stochastic Gradient Descent

An optimization algorithm is required to solve a machine learning task. The target is to find the minimum value of the objective function corresponding to the learning task. To solve this task, the programmer provides the gradient of the cost function and TABLA subsequently uses stochastic gradient descent as the optimization algorithm to determine the parameters best suited for the given training data. Since SGD is independent of the learning task, we devise a general template to implement it. As a result, we need a mechanism to integrate the gradient of the objective function with our template.

As seen in Section 4, the programmer provided code generates a final result, which is the gradient of the cost function and is represented using the *gradient* argument. Stochastic gradient descent is then executed using this *gradient* result and *model* declarations provided by the programmer using the following code:

```
model        w[n][m]; //model parameters
gradient     g[n][m]; //gradient

g[j][i] = μ * g[j][i] //n*m parallel multiplications
g[j][i] = w[j][i]-g[j][i]; //n*m parallel subtractions
```

The gradient of the objective function provided by the programmer and the stochastic gradient descent together form the entirety of the learning task. Once the entire algorithm is available, the data-flow graph (DFG) is generated. This graph represents the entire ML algorithm.

5.2 Data-Flow Graph Generation

The code provided by the programmer is converted into a dataflow graph. The model compiler appends the above generate DFG with the of stochastic gradient descent. Before delving into further details of generating a DFG for a particular ML algorithm, we visualize the DFG of each language construct in our programming interface as shown in Table 2.

Dataflow graph of individual operations. Figure 3 shows the data-flow graph for at least one operation of each type - basic, group and nonlinear. These dataflow graphs show the input and

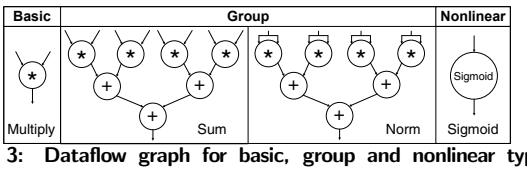


Figure 3: Dataflow graph for basic, group and nonlinear type of operations. The DFG for multiply, sum, norm and sigmoid operations are shown.

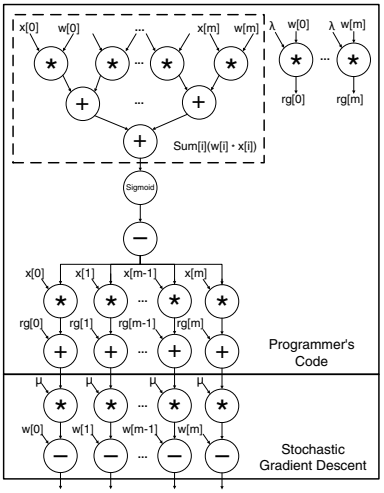


Figure 4: A complete dataflow graph of the logistic regression algorithm

output edges along with the intermediate nodes that perform the computation of each operation. The group operations involve more than one computational node. Figure 3 shows the dataflow graph for *sum* and *norm*. They both involve multiplication of input elements and use an adder tree to generate the final results. The dataflow graph also depicts the opportunities for parallelism that will be exploited by the hardware accelerator. Once the dataflow graphs for individual operations are available, the model compiler can combine these dataflow graphs in accordance with the code that expresses the gradient of the learning task.

Dataflow graph of the learning algorithm. The DFG for a ML task can be generated by combining the DFG of each operation with the code of the gradient function, while maintaining the dependencies. The DFG for the Logistic Regression along with the appended stochastic gradient descent step is presented in Figure 4. This DFG corresponds to the example code given in section 4.2 with n = 1. As illustrated in Figure 4, the gradient function can be easily converted to a DFG using the individual DFGs of the operations. For example, the summation operation in the programmers code (*sum[i](x[i] * w[j][i])*) is directly converted to a series of multiplications followed by an adder tree by the compiler. In addition to the DFG of the objective function, the DFG in the figure is appended with the DFG of stochastic gradient descent, thereby generating a complete dataflow graph for the entire machine learning algorithm.

After the compiler framework generates the DFG, different scheduling algorithms can be used to schedule each operation in the DFG. We perform this scheduling using a Minimum Latency - Resource Constrained Scheduling algorithm, which schedules operations given a limited set of resources. The details of this scheduling algorithm are presented in the next subsection.

5.3 Scheduling

Once the compiler generates the DFG for the entire ML algorithm, the scheduler can generate a step-by-step schedule of the operations for a given resource constraints. The DFG for Logistic Regression shown in Figure 4 is an As-Soon-As-Possible (ASAP) graph. This graph can be easily scheduled using the ASAP algorithm. An ASAP algorithm schedules an operation as soon as all predecessors of the particular operation are completed. This ASAP schedule generated as the result, achieves minimum latency however assumes infinite resources. However, generating accelerators with unlimited resources is infeasible. Thus, there is a need to use a more practical algorithm that aims to reduce latency for a given resource constraint. The scheduling algorithm presented in this paper is referred to as the Minimum Latency - Resource Constrained Scheduling (ML-RCS).

Inputs: $\mathbb{R}$: 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 *cycle_count* $\leftarrow 0$
while ($O \neq \emptyset$) **do**
  for ($r \in R$) **do**
    Initialize $s \leftarrow \emptyset$
    if $o \subset O$ *where* o.predecessors = DONE & D[s] = max(D) **then**
      s.op = o; s.cycle = *cycle_count*
      S.append(s)
      O.remove(o)
    end if
  end for
  cycle_count = *cycle_count* + 1
end while

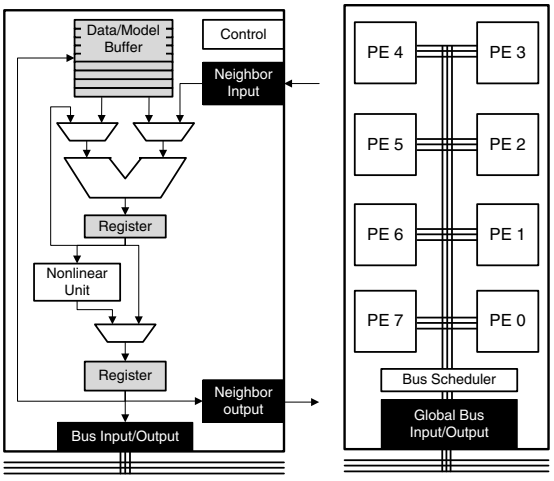
Algorithm 1: ML-RCS Scheduling for the Dataflow graph.

The algorithm shown above is commonly referred to as Hu's scheduling algorithm [17]. This algorithm schedules operations by making the following assumptions: (1) all the operations are single step; and (2) all operations use a single type of resource. These assumptions hold valid for our accelerator design, as our base design comprises of only one type of resource called the Processing Engine that takes one step to generate the result. The details of our design and processing engine are provided in section 6. By making these assumptions, Hu's scheduling provides an optimal solution for the ML-RCS optimization problem. Before delving deeper into the intricacies of the scheduling algorithm, we define a term – *distance from sink*. *Distance from sink* of an operation (op) is the number of operations that need to be performed after the op to reach the final output/sink. *Distance from sink* is an important metric that quantifies the priority of each operation. The higher the *distance from sink*, the higher its priority is. In algorithm 1, an operation (op) is scheduled at cycle (c) if all the following conditions are met: (1) all the predecessors have been scheduled and completed; (2) it has the highest priority (or *distance from sink*) among the unscheduled ready ops; (3) resource is available to accommodate the op. The algorithm terminates when all the operations are successfully scheduled.

After the schedule for all the operations is generated using the ML-RCS algorithm by the compiler, TABLA framework then generates the design for the hardware accelerator that can accommodate this schedule. This hardware generation procedure and our basic accelerator design is described in Section 6.

Table 3: Benchmarks, their brief description, size of the training data sets, and the model topology.

Name	Model	Algorithm Name	Description	Input Vectors	# of Features	Model Topology	Lines of Code	Optimal # of PE/PU
LogisticR	M1	Logistic Regression	Estimates the probability of dependent variable given one or more independent variables	581,000	54	54	20	32/4
	M2			500,000	200	200	20	64/8
SVM	M1	Classification (SVM)	Classifies data into different categories by identifying support vectors	581,000	54	54	23	32/4
	M2			500,000	200	200	23	64/8
Reco	M1	Recommender Systems	Information filtering system that predict the preference a user would give to an item	1,700,000	27,000	1700x1000	31	64/8
	M2			24,000,000	100,000	6000×4000	31	64/8
Backprop	M1	Backpropogation	Trains a neural network that model the mapping between the inputs and outputs of the data	38,000	10	10 -> 9 -> 1	48	16/2
	M2			90,000	256	256 -> 128 -> 256	48	64/8
LinearR	M1	Linear Regression	Models relationship between a dependent variable and one or more explanatory variables	10,000	55	55	17	32/4
	M2			10,000	784	784	17	64/8



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

Figure 5: (a) A processing engine comprising of compute and memory units.(b) Processing unit comprising of 8 processing engines connected through a intra-PU bus

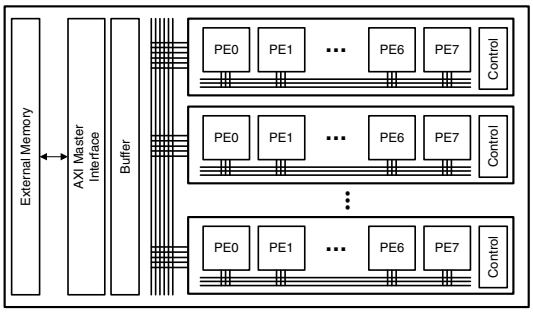


Figure 6: Accelerator design showing the processing units and processing engines. The processing engines are connected through intra-PU bus and the processing units are connected through the inter-PU bus. The 4 AXI interfaces provide communication between external memory and the accelerator.

Table 4: CPU and GPU platforms used for Tabla evaluation

Platform	Cores	Core Clock Freq (GHz)	Memory Avail (GB)	Year	TDP (W)	Technology (nm)	Cost
ARM Cortex A15	4+1	2.3	2 (shared)	2014	5	28	$191
Intel Xeon E3-1276v3	4	3.6	16	2014	84	22	$339
Tegra K1 GPU	192	0.852	2 (shared)	2014	5	28	$191
GeForce GTX 650 Ti	768	0.928	1	2012	110	28	$150
Tesla K40	2880	0.875	12	2013	235	28	$5,499

6 Accelerator Design

Due to the flexibility and reconfigurability provided by FPGAs we choose FPGAs to accelerate the machine learning task represented as a schedule provided by the compilation framework. In this section, we describe the architecture of the hardware that accelerates this learning task.

Table 5: FPGA hardware platform.

FPGA hardware platform	
Model	Xilinx Zynq ZC702
Technology	TSMC 28nm
FPGA	Artix-7
FPGA Capacity	53K LUTs
	106K Flip-Flops
Peak Frequency	250MHz
BRAM	630 KB
DSP Slice	DSP48E1
MACC Count	220
TDP (W)	2

6.1 Processing Unit

TABLA's compilation framework produces different schedule for different learning algorithms. Thus, we propose a reconfigurable accelerator design that can accommodate these schedules and can be specialized to accelerate a range of ML algorithms. The fundamental component of this accelerator architecture is a Processing Engine (PE). The components within a PE and its interconnection with other PEs is designed to accelerate the learning algorithm. The design tradeoffs and variability of a PE is discussed in further detail below.

Processing Engine (PE). Processing engine (PE) is the fundamental and reconfigurable unit of our design and hence needs to be customized according to the ML task. A comprehensive and reconfigurable design of processing engine is shown in Figure 5a. As the figure illustrates, the processing engine comprises of a computational unit (ALU) that performs calculations and a storage unit (data/model buffer) that stores the model parameters and data elements. Some of the components shown in the Figure 5a are fixed within a PE, while the others can be reconfigured.

The *fixed components* in PE include the ALU, Data/Model Buffer, Register and Bus. As all the statistical machine learning tasks have some form of mathematical operation making the ALU a crucial component of PE. In addition to the ALU, a buffer is necessary to store the model or any other incoming data from external DRAM. The register is essential for some of the group operations such as sum ($\sum$) or pi (Π). Finally, a bus interface is crucial and is reserved for retrieving data from the external memory. Communication between a PE and the external memory is inevitable as the external DRAM provides all the training data and the initial model parameters. Contrastingly, the communication with other PEs is not always required and is dependent on the algorithm.

The *exchangeable components* in a PE include – specialized control unit, nonlinear unit, multiplexers, and the neighbor input and output communication. The reconfigurability of the nonlinear unit is due to the fact that some algorithms like SVM, recommender system and linear regression, do not require any nonlinear operations. Furthermore, even the algorithms that employ a nonlinear operation, use this unit sporadically. The DGF of

Logistic Regression in Figure 4 demonstrates one such scenario. The figure shows that the nonlinear function is only employed on the summation output. Therefore, incorporating a nonlinear unit in every PE is a waste of area and power. Therefore, a nonlinear unit is only provided in the PE which accrues all the results and generates the final *sum* result. Finally, communication between neighboring PUs is useful for algorithms that combine data. Reading the neighbor's result would avoid contention on the bus if multiple PEs need data from the other PEs. Allowing neighbor input and output in our PEs conforms with the traditional concept of spatial locality in computer architecture. It has been observed that the likelihood of referencing a resource is more if the resource is spatially close. Once the PE is finalized in congruence with the ML algorithm, it can be incorporated into the bigger design i.e the processing unit.

Processing Unit (PU). The PU contains eight identical processing engines (PEs) as shown in Figure 5b. Although the design can scale to larger or smaller numbers of PEs, we find that the allowable frequency is maximum with 8 PEs (see section 7). The bus between the PEs is referred to as the intra-PU bus and the one between the PUs is referred to as an inter-PU bus. This design comprising of PUs and PEs is implemented on a Xilinx Zynq FPGA as described in the next section.

6.2 Target FPGA Platform

We use a Xilinx Zynq-7000 programmable SoC evaluation platform. Figure 6 illustrates our entire design synthesized on an FPGA platform. The design comprises of multiple processing units connected through a pipelined global bus. This pipelined global bus also connects our design to the AXI interface which in turn connects to an external memory. The training input and output data in the training data is transferred from the external memory to the programmable logic after every iteration of the learning algorithm. The initial model parameters are only transferred once at the start of the execution. The number of processing units in the design are dependent on the algorithm being implemented. For example, the DFG of logistic regression, shown in Figure 4, can have a maximum of 54 parallel operations. This model characteristics implies that 64 PEs or 8 PUs are sufficient to cater for the requirements of the algorithm. Similarly, a range of accelerators can be generated to cater for different machine learning algorithms supported by the TABLA framework. This reconfigurability and flexibility provided by the TABLA framework is evaluated using FPGAs and the results are presented in the next section. For example, the DFG of logistic regression, shown in Figure 4, can have a maximum have of 54 parallel operations. This implies that 64 PEs or 8 PUs are sufficient to cater for the requirements of logistic regression. This reconfigurability and flexibility provided by the TABLA framework is evaluated using the Zynq FPGA and the results are presented in the next section.

7 Evaluation

We evaluate the TABLA framework by implementing the hardware accelerator using the Xilinx Zynq ZC702 off-the-shelf FPGA platform (see Table 5) and compare its performance and energy utilization against different CPU and GPU platforms. We rigorously compare the benefits of the FPGA acceleration to both multicore CPUs (ARM Cortex A15 and Xeon E3) and to many-core GPUs (Tegra, GTX 650 Ti, and Tesla K40) using real hardware measurements.

7.1 Experimental Setup

Benchmarks. Table 3 lists the machine learning algorithms that are 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 statistical learning processes encompassing regression analysis, statistical classification, information filtering systems, recommender systems, and backpropagation. Table 3 also includes some of the most pertinent learning parameters such as the number of training vectors, model topology and the number of lines required to implement their gradient function in the TABLA programming interface. Each algorithm is evaluated with two models M1 and M2 each corresponding to a different topology. This evaluation across multiple models allows us to evaluate the flexibility of the TABLA framework in accommodating the machine learning algorithms when their topology changes. These models are used in the literature for different machine learning task. For LogisticR and SVM, we use two model topologies from the UCI repository. One dataset has 54 features and the other has 200. We modified the datasets to incorporate binary output values. For Reco, we used two different topologies from movieLens [18, 19], a movie database. For BackProp we use two topologies - one with a larger neural network topology (256→128→256) [20] and the other with a smaller neural network topology (10→9→1) [21]. For LinearR we use one topology from the UCI repository and one from MNIST.

CPU and GPU platforms. As Table 4 shows, we evaluated TABLA through comparison with a wide range of diverse platforms using direct hardware measurements. We compare TABLA to two multicore CPU processors: (1) the low-power quad-core ARM A15 that is available on the Nvidia Jetson TK1 development kit [22] and operates at 2.3 GHz; (2) the high performance quad-core Intel Xeon E3 with hyper-threading support that operates at 3.6 GHz. Further, we compare TABLA 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; (3) and the high-performance Tesla K40 GPU accelerator with 2880 SIMD cores. All the platforms run Ubuntu Linux version 14.04.

Multithreaded code for CPU execution. To compare TABLA with the CPU platforms, we use optimized open-source multithreaded implementation of the machine learning algorithms. We use Liblinear [23] for logistic regression and SVM, Mlpack [24] for recommender systems and linear regression and Caffe [25] for backpropagation. All the code are compiled with gcc 4.8 with the -O3 -ftree-vectorize -march=native flags to enable aggressive compiler optimizations and utilize vector executions. All the codes runs 4 threads on ARM and 8 threads on Xeon since the quad-core ARM does not support multithreading where the quad-core Xeon does. The multithreaded support is either implemented using OpenMP (Liblinear) or using OpenBLAS [26] (Mlpack and Caffe). In addition to libraries which were reported in the paper, we also tried a wide spectrum of other libraries (LibFM [27], Libsvm [28], FANN [29]). These libraries provide inferior performance compared to the ones reported in the paper which provide the highest performance on the CPUs.

Optimized CUDA implementation for GPU execution. For the GPU platforms, we use highly optimized CUDA implementations from [30], Caffe+cuDNN [25], and LibSVM-GPU [31].
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 FPGA runtime obtained from the hardware counters synthesized on the programmable logic that measure the cycle counts.
FPGA synthesis and hardware utilization. We synthesize the hardware with 64-bit Vivado v2015.1, which also generates the area utilization. The FPGA area and resource utilization is provided in Table 6. The accelerators operate at 150 MHz.

7.2 Experimental Results

Performance improvements. Figure 7a shows the benchmark speedups of TABLA-generated FPGA accelerator and the Xeon E3 CPU in comparison to ARM A15 CPU. The ARM is the baseline in all the speedup graphs. ARM is a low power CPU and therefore is outperformed by both Xeon and TABLA-generated accelerators. TABLA outperforms ARM by an average speedup of 15.0× while Xeon outperforms ARM by an average speedup of 5.2×. TABLA exhibits a higher speedup than Xeon by a factor of 2.9×. The maximum speedup of 46× is seen by the TABLA design while a maximum speedup of 10.5× is seen by the Xeon, both for the Reco M2 benchmark. This result is observed because of the relatively larger model topology of Reco M2, which provides greater opportunities for parallelism that can be exploited by the accelerator more than the multicore CPUs. In only one case Backprop M2, TABLA design provide lower speedup in comparison to Xeon (0.59×) but still outperforms Xeon for Backprop M1 by 1.6×. This observation can be attributed to the fact that even though the backpropagation algorithm has some intra-operation parallelism, the parallelism among different operations is fairly limited. The bottleneck from the serialization of operations is not as limiting in the smaller model of Backprop M1, however clearly shows its impact for the much larger topology (Backprop M2). One possible solution to avoid this bottleneck can be to simultaneously run multiple iterations of the gradient function over different training input elements.

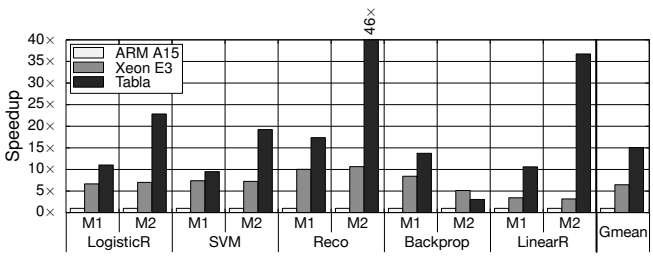
We further compare the speedup benefits with different GPU platforms in Figure 7b. The baseline is the ARM multicore processor. Tesla provides an average speedup of 59×, followed by GTX 650 Ti with an average speedup of 15.5×. TABLA closely follows GTX 650 Ti by providing a speedup of 15.0×. However, Tegra K1, TABLA provides only 2× speedup. Furthermore, in comparison to Xeon, Tesla provides an average speedup of 8.95×, followed by GTX 650 speedup of 2.4×. Tegra, on the other hand shows a slow down of 2.43× in comparison to Xeon. The higher power consumption of Tesla (235W of TDP) and GTX 650 (110W of TDP) explain their higher speedup numbers. Comparatively the TDP of Zynq is 2W.

These results conform with three other recent investigations which reported GPUs to have 15× to 49× [32], 10× to 60× [33] and 10× to 100× [30] speedups over CPU for ML applications.
Performance-per-Watt comparison. As the speedup results show, the TABLA-generated FPGA accelerators provide significant speedup over both multicore CPUs and the Tegra K1 GPU within a limited power budget of 2W. The performance benefits of our FPGA accelerators are on average in par with the desktop-grade GTX 650 Ti GPU with the TDP of 110W. However, the more high-performance Tesla K40 with the TDP of 235W provides higher performance as expected. We compare the performance-per-watt to understand the benefits of FPGA acceleration without the variations in the power budget. Figure 8a compares the performance-per-Watt for ARM A15, Xeon E3 and TABLA. Similarly, Figure 8b illustrates the performance-per-Watt for the GPU platforms. TABLA, on average, achieves 30.1× and 81.7× over ARM and Xeon, respectively. On the GPU side, TABLA's FPGA accelerators provide 22.7×, 53.7×, and 30.6× higher performance-per-Watt compare to Tegra, GTX 650, and Tesla, respectively.

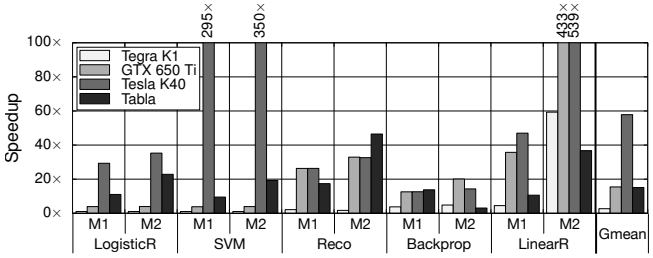
Interestingly, ARM A15 achieves 2.7× higher performance-per-Watt than Xeon. It is also interesting that Tesla achieves a level of efficiency which is similar to ARM, yet it provides much higher performance. The TABLA-generated FPGA accelerators close the performance gap to a large extent but provide much higher efficiency and operate at significantly lower power budget. In any case, GPUs are an attractive back-end for TABLA that can be explored in future work. However, as Table 4 shows, they require much higher power.
Area and FPGA utilization. Table 6 shows the resource utilization for different components on the FPGA for each learning algorithm. Backprop M1 utilizes the least area among all the learning algorithms as it has a relatively smaller model and requires only 16 PEs for its default configuration. On the other hand, Reco M1 and M2 and Backprop M2 utilize a much larger area and BRAM as their default configuration is large and they also have a significantly higher numbers of parameters that need to be stored within 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 8 PEs strikes a balance between frequency and intra-PU parallelism. Note that this design space exploration is not the responsibility of the programmer and is part of TABLA. It is usually done when the templates are tuned for a FPGA platform by expert hardware designers.
Number of processing engines. TABLA provides a template-based architecture that can generate reconfigurable hardware accelerators. As described in section 6.1, the number of processing engines and processing units can be customized in accordance with the algorithm. Figure 9, shows the effect of varying the number of processing engines on the speedup for each benchmark. The baseline is the ARM multicore CPU. As expected, initially the increase in number of PEs leads to a fairly linear increase in the speedup. However, beyond a certain number of PEs we either observe diminishing returns or a decrease in the speedup. This observation can be attributed to the fact that the parallelism in the algorithms is limited and increasing the number of PEs beyond a point would just lead to waste of resources. For example, in the LogisticR M1 benchmark, a maximum of 54 operations can be done in parallel at a given time. Therefore, providing PEs beyond 54 is inconsequential. However, our base design only allows 8 PEs per PU, hence the design for LogisticR M1 has 32 PEs or 4

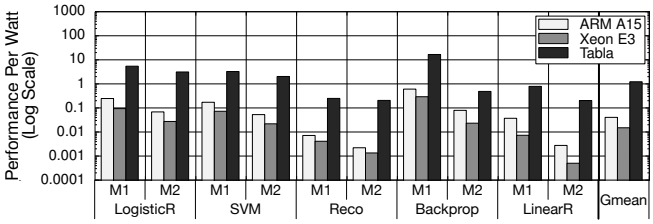


(a) Speedup of Xeon E3 and TABLA design in comparison to ARM A15.

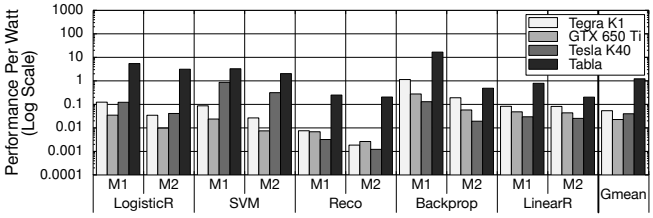


(b) Speedup of the GPUs and TABLA design in comparison to ARM A15.

Figure 7: Speedup of Tabla in comparison to a range of CPU and GPU platforms. The baseline is ARM.



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



(b) Performance-per-Watt comparison between Tegra, GTX 650 , Tesla and TABLA

Figure 8: Comparison of performance-per-Watt between CPUs, GPUs and Tabla

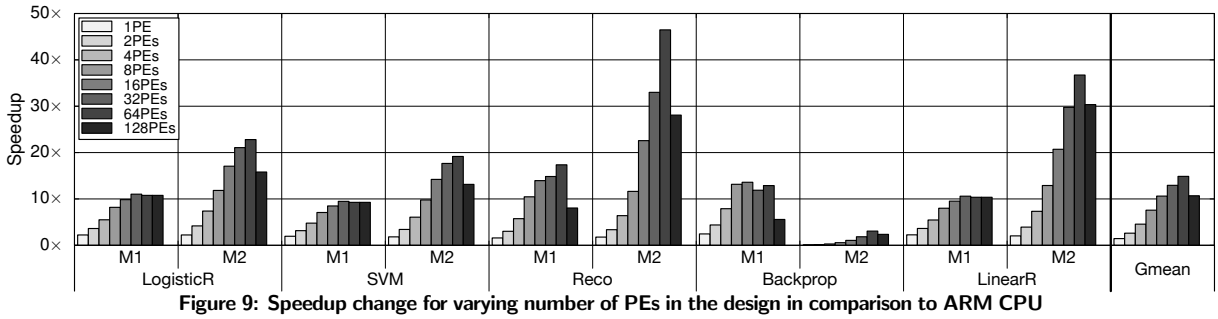


Figure 9: Speedup change for varying number of PEs in the design in comparison to ARM CPU

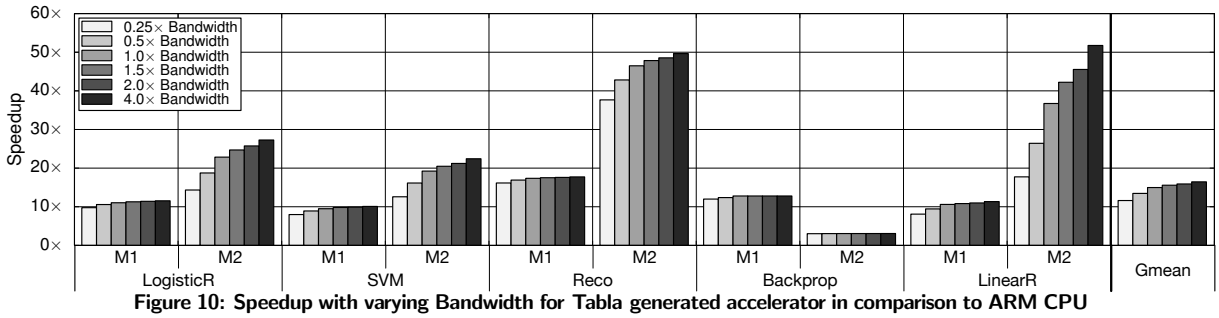


Figure 10: Speedup with varying Bandwidth for Tabla generated accelerator in comparison to ARM CPU

Table 6: Resource utilization on the FPGA for each learning algorithm.

Benchmark		LUT (Total Available: 53200)		BRAM (Total Available: 630KB)		Flip-Flops (Total Available: 106400)		DSP Slices (Total Avilable: 220)	
Name	Model	Total Used	Utilization	Total Used(Bytes)	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%

PUs. Increasing the PEs beyond 32 leads to either decrease in the speedup or has no impact on the speedup for all the benchmarks. This anomaly is observed because as the number of PEs increase the operational frequency also decreases due to the requirement of a wider and bigger global bus. Therefore, adding more PEs might not improve speedup due to lack of abundant parallelism but rather decreases the speedup due to a slower hardware. The varying number of PEs also gives us the optimal design that utilizes minimum resources and still provides maximum benefits from acceleration.

Bandwidth sensitivity. Machine learning algorithms are both compute and data intensive tasks. We design the accelerator to exploit the fine-grained parallelism in the computational component of the algorithm. On the other hand, the data is provided to the compute elements of the design either through a memory buffer in the PE or external memory. The data transferred from external memory to the accelerator uses the AXI interface which has limited bandwidth. We do an analysis of trends observed in speedup with this changing bandwidth between external memory and the accelerator. Figure 10, shows the speedup for each benchmark if the bandwidth is varied from 0.25× of the default to 4× the default. The figure shows that bandwidth can be bottleneck at very low values such as 0.25× of the default bandwidth. As the bandwidth increases the speedup starts to increases but observe diminishing returns beyond a point. By providing a bandwidth that is 4× the default value the speedup numbers only increase by 60% of the default speedup. The bandwidth sweeps are done by creating a cycle-accurate simulator of the accelerator which is validated against hardware.

8 Related Work

There have been several proposed architectures that accelerate machine learning algorithms [21, 30, 34–46]. However, TABLA fundamentally differs from these works, as it is not an accelerator. TABLA is an accelerator generator for an important class of machine learning algorithms, which can be expressed as stochastic optimization problems. Using this insight, TABLA 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. There have also been architectures that accelerate gradient descent [47] and conjugate gradient descent [47–50]. However, these works do not specialize their architectures in machine learning algorithms or any specific objective function. They neither provide specialized programming models nor generate accelerators. Below, we discuss the most related work.

Gradient descent accelerators. The work by Kesler [47] focuses only on designing an accelerator suitable for different linear algebra operations to facilitate the gradient descent and conjugate gradient algorithms.

Machine learning accelerators. There have been several successful works in the past that focus on accelerating a single or a range of fixed ML tasks. Yeh et al. and Manolakos and Stamoulias focused on designing accelerators for a particular ML algorithm (k-NN) [34, 35, 51]. Furthermore, work has also been done on accelerating k-Means [36–38] and Support Vector Machines (SVM) [39, 40] due to their wide applicability. These acceleration techniques have also been extended to conventional and deep neural networks [21, 43–46]. However, all these accelerators are focused on accelerating a particular ML task.

To add more flexibility and accelerate beyond one algorithm, several works focus on designs that accelerate a range of learning algorithms [30, 41, 42]. The work by Majumdar – MAPLE, focuses on classification and learning, while PuDianNao accommodates seven representative ML algorithms. Even though PuDianNao does cover a large spectrum of ML algorithms, it does not provide the flexibility to extend the accelerator for new ML tasks. Besides, PuDianNao is an ASIC accelerator, whereas TABLA can generate accelerators for any platforms if proper backend support is provided.

FPGA accelerators. FPGAs have gained popularity due to their flexibility and capability to exploit copious fine-grained irregular parallelism for higher performance execution. Furthermore, the work in [34, 36, 39, 40, 51–55] utilize FPGAs to accelerate a diverse set of workloads, validating the efficacy of FPGAs. LINQits [56] provides a template architecture for accelerating database queries. The work by King et. al. [57] uses Bluespec to automatically generate a hardware-software interface for programmer-specified hardware-software partitions. The work by Putnam et. al. [7], designs an FPGA fabric for accelerating ranking algorithms in the Bing server. They integrate this fabric in 1632 servers at Microsoft. TABLA provides an opportunity to utilize this integrated fabric in the servers for machine learning algorithms. Conclusively, TABLA provides a comprehensive solution – from programming language down to circuit design – that can generate ML accelerators.

9 Conclusion

Machine learning algorithms include 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 design 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 leverages stochastic gradient descent as the abstraction between hardware and software to automatically generate accelerators for a class of statistical machine learning algorithms. We used TABLA to generate accelerators for a verity of learning algorithms targeting an off-the-shelf FPGA platform, Xilinx Zynq. Compared to a multicore Intel Xeon with vector execution, the TABLA-generated accelerators deliver an average speedup of 2.9×. Compared to the high-performance Tesla K40 GPU accelerator, TABLA achieves 30.6× higher performance-per-Watt. These gains are achieved while the programmers only write less than 50 lines of code. These results suggest that TABLA takes an effective step in a widespread use of FPGAs for machine learning algorithms. We plan to make TABLA publicly available to the larger research community in order to facilitate FPGA acceleration of machine learning algorithms.

10 Acknowledgements

This work was supported by a Qualcomm Innovation Fellowship, NSF award CCF #1553192, Semiconductor Research Corporation contract #2014-EP-2577, 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 *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ASPLOS '15, 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 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 325–336, New York, NY, USA, 2012. ACM.

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

[18] Iván Cantador, Peter Brusilovsky, and Tsvi Kuflik. 2nd workshop on information heterogeneity and fusion in recommender systems (hetrec 2011). In *Proceedings of the 5th ACM conference on Recommender systems*, RecSys 2011, New York, NY, USA, 2011. ACM.

[19] Grouplens. Movielens dataset.

[20] 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, Englewood Cliffs, New Jersey, 1993.

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

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

[23] 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.

[24] 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.

[25] 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.

[26] Zhang Xianyi, Wang Qian, and Zhang Yunquan. Model-driven level 3 blas performance optimization on loongson 3a processor. In *Proceedings of the 2012 IEEE 18th International Conference on Parallel and Distributed Systems*, ICPADS '12, pages 684–691, Washington, DC, USA, 2012. IEEE Computer Society.

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

[28] 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.

[29] 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.

[30] 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 *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 369–381, New York, NY, USA, 2015. ACM.

[31] Andreas Athanasopoulos, Anastasios Dimou, Vasileios Mezaris, and Ioannis Kompatsiaris. Gpu acceleration for support vector machines. In *WIAMIS 2011: 12th*

International Workshop on Image Analysis for Multimedia Interactive Services, Delft, The Netherlands, April 13-15, 2011. TU Delft; EWI; MM; PRB, 2011.

[32] G. Teodoro, R. Sachetto, O. Sertel, M.N. Gurcan, W. Meira, U. Catalyurek, and R. Ferreira. Coordinating the use of gpu and cpu for improving performance of compute intensive applications. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1–10, Aug 2009.

[33] Dan C. Cireşan, Ueli Meier, Jonathan Masci, Luca M. Gambardella, and Jürgen Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, pages 1237–1242. AAAI Press, 2011.

[34] 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.

[35] E.S. Manolakos and I. Stamoulias. Ip-cores design for the knn classifier. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 4133–4136, May 2010.

[36] 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 *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, pages 248–255, June 2011.

[37] Tsutomu Maruyama. Real-time k-means clustering for color images on reconfigurable hardware. In *Proceedings of the 18th International Conference on Pattern Recognition - Volume 02*, ICPR '06, pages 816–819, Washington, DC, USA, 2006. IEEE Computer Society.

[38] 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 *Integrated Circuits and Systems Design, 2003. SBCCI 2003. Proceedings. 16th Symposium on*, pages 99–104, Sept 2003.

[39] M. Papadonikolakis and C. Bouganis. A heterogeneous fpga architecture for support vector machine training. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 211–214, May 2010.

[40] 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 *Field Programmable Custom Computing Machines, 2009. FCCM '09. 17th IEEE Symposium on*, pages 115–122, April 2009.

[41] 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.

[42] 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.

[43] 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.

[44] 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 *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 269–284, 2014.

[45] A.A. Maashri, M. DeBole, M. Cotter, N. Chandramoorthy, Yang Xiao, V. Narayanan, and C. Chakrabarti. Accelerating neuromorphic vision algorithms for recognition. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 579–584, June 2012.

[46] 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.

[47] D. Kesler, B. Deka, and R. Kumar. A hardware acceleration technique for gradient descent and conjugate gradient. In *Application Specific Processors (SASP), 2011 IEEE 9th Symposium on*, pages 94–101, June 2011.

[48] 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.

[49] G.R. Morris, V.K. Prasanna, and R.D. Anderson. A hybrid approach for mapping conjugate gradient onto an fpga-augmented reconfigurable supercomputer. In *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, pages 3–12, April 2006.

[50] D. DuBois, A. DuBois, T. Boorman, C. Connor, and S. Poole. An implementation of the conjugate gradient algorithm on fpgas. In *Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on*, pages 296–297, April 2008.

[51] 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 *Proceedings of the 15th Scandinavian Conference on Image Analysis*, SCIA'07, pages 512–521, Berlin, Heidelberg, 2007. Springer-Verlag.

[52] 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.

[53] Yi Shan, Bo Wang, Jing Yan, Yu Wang, Ningyi Xu, and Huazhong Yang. Fpmr: Mapreduce framework on fpga. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '10, pages 93–102, New York, NY, USA, 2010. ACM.

[54] Jeremy Fowers, Kalin Ovtcharov, Karin Strauss, Eric Chung, and Greg Stitt. A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication. In *International Symposium on Field-Programmable Custom Computing Machines*. IEEE, May 2014.

[55] Eric S. Chung, James C. Hoe, and Ken Mai. Coram: An in-fabric memory architecture for fpga-based computing. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 97–106, New York, NY, USA, 2011. ACM.

[56] Eric S. Chung, John D. Davis, and Jaewon Lee. Linqits: Big data on little clients. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 261–272, New York, NY, USA, 2013. ACM.

[57] M. King, A. Khan, A. Agarwal, O. Arcas, and Arvind. Generating infrastructure for fpga-accelerated applications. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–6, Sept 2013.