Scale-Out Acceleration for Machine Learning

Jongse Park† Hardik Sharma† Divya Mahajan† Joon Kyung Kim† Preston Olds† Hadi Esmaeilzadeh†‡
Alternative Computing Technologies (ACT) Lab
†Georgia Institute of Technology ‡University of California, San Diego
{jspark, hsharma, divya_mahajan, jkkim, prestonolds}@gatech.edu hadi@eng.ucsd.edu

ABSTRACT

The growing scale and complexity of Machine Learning (ML) algorithms has resulted in prevalent use of distributed general-purpose systems. In a rather disjoint effort, the community is focusing mostly on high performance single-node accelerators for learning. This work bridges these two paradigms and offers CoSMIC, a full computing stack constituting language, compiler, system software, template architecture, and circuit generators, that enable programmable acceleration of learning at scale. CoSMIC enables programmers to exploit scale-out acceleration using FPGAs and Programmable ASICs (P-ASICs) from a high-level and mathematical Domain-Specific Language (DSL). Nonetheless, CoSMIC does not require programmers to delve into the onerous task of system software development or hardware design. CoSMIC achieves three conflicting objectives of efficiency, automation, and programmability, by integrating a novel multi-threaded template accelerator architecture and a cohesive stack that generates the hardware and software code from its high-level DSL. CoSMIC can accelerate a wide range of learning algorithms that are most commonly trained using parallel variants of gradient descent. The key is to distribute partial gradient calculations of the learning algorithms across the accelerator-augmented nodes of the scale-out system. Additionally, CoSMIC leverages the parallelizability of the algorithms to offer multi-threaded acceleration within each node. Multi-threading allows CoSMIC to efficiently exploit the numerous resources that are becoming available on modern FPGAs/P-ASICs by striking a balance between multi-threaded parallelism and single-threaded performance. CoSMIC takes advantage of algorithmic properties of ML to offer a specialized system software that optimizes task allocation, role-assignment, thread management, and internode communication. We evaluate the versatility and efficiency of CoSMIC for 10 different machine learning applications from various domains. On average, a 16-node CoSMIC with UltraScale+ FPGAs offers 18.8× speedup over a 16-node Spark system with Xeon processors while the programmer only writes 22–55 lines of code. CoSMIC offers higher scalability compared to the state-of-the-art Spark; scaling from 4 to 16 nodes with CoSMIC yields 2.7× improvements whereas Spark offers 1.8×. These results confirm that the full-stack approach of CoSMIC takes an effective and vital step towards enabling scale-out acceleration for machine learning.

MICRO-50, October 14-18, 2017, Cambridge, MA, USA

ACM ISBN 978-1-4503-4952-9/17/10…$15.00
https://doi.org/10.1145/3123939.3123979

CCS CONCEPTS

• **Hardware** → **Hardware accelerators**; • **Computing methodologies** → **Machine learning**; • **Computer systems organization** → **Cloud computing**;

KEYWORDS

Accelerator, scale-out, distributed, cloud, machine learning

ACM Reference format:
Jongse Park, Hardik Sharma, Divya Mahajan, Joon Kyung Kim, Preston Olds, and Hadi Esmaeilzadeh. 2017. Scale-Out Acceleration for Machine Learning. In *Proceedings of The 50th Annual IEEE/ACM International Symposium on Microarchitecture, Cambridge, MA, USA, October 14-18, 2017 (MICRO-50),* 15 pages.
https://doi.org/10.1145/3123939.3123979

1 INTRODUCTION

Prevalence of interconnected compute platforms has transformed the IT industry, which is now rapidly moving towards scale-out solutions that extract insights from data. Following this trend, systems that enable distributed computing on general-purpose platforms are gaining eminence (e.g., Spark [1] and Hadoop [2]). In a concurrent yet disjoint effort, due to the diminishing benefits from general-purpose processing, the community is developing mostly single-node accelerators for a variety of applications, including machine learning [3–12]. However, there is a gap between scale-out systems and accelerators due to the lack of solutions that enable distributed acceleration at scale. Moreover, it is not enough to just design and integrate accelerators independent from algorithms and programming interfaces. We need a holistic approach that reworks the fundamental hardware-software abstractions and enables a broad community of programmers to seamlessly utilize accelerators at scale for a specific domain of applications. Reusing the traditional stack for scale-out acceleration is inadequate as the entire computing stack is designed and optimized merely for CPUs, which were the sole processing platform up until recently. To that end, this paper sets out to design a full and specialized computing stack, dubbed CoSMIC[1], for scale-out acceleration of learning.

CoSMIC offers the entire stack of layers to execute a wide range of learning algorithms on accelerator-augmented scale-out systems. These layers comprise a domain-specific language, a compiler, a specialized runtime system, and a multi-threaded template architecture for the accelerator. The template architecture can be automatically tailored for deployment on FPGAs or realization as custom Programmable ASICs (P-ASICs). FPGAs offer flexibility as well as efficiency and are becoming readily available in different markets [13–16], now even in Amazon Elastic Compute Cloud (EC2) [16]. Not only have FPGAs become a lower-cost alternative to ASICs, but also serve as prototypes for custom chip design. However, designing efficient accelerators is onerous even when targeting a single-node FPGA and requires extensive expertise in both hardware design and

[1] **CoSMIC**: **Co**mputing **S**tack for **M**L acceleration **I**n the **C**loud

application domain. This challenge is exacerbated in the scale-out setting due to the added complexity of task distribution and communication. Additionally, P-ASICs impose high non-recurring engineering costs over long design periods and usually need unintuitive or narrow programming interfaces. Furthermore, as technology is scaled, modern FPGAs and ASICs can harbor an ample amount of resources, whose effective utilization necessitates rethinking accelerator design paradigms. Therefore, to realize scale-out acceleration, we address the following triad of challenges when devising the CoSMIC full stack: (1) efficiently exploiting large number of on-chip resources, (2) enabling distributed acceleration using accelerator-augmented nodes, and (3) relieving programmers of distributed system coordination and the onus of hardware design. Furthermore, CoSMIC targets a wide class of learning algorithms and provides support for new learning models and algorithmic changes to the existing ones. To realize CoSMIC we were required to address the following research challenges.

(1) How to enable scale-out acceleration of many ML algorithms, yet disengage programmers from hardware design. To tackle this challenge, CoSMIC leverages a combination of two theoretical insights: (1) a wide range of learning algorithms are stochastic optimization problems, solved using a variant of gradient descent [12, 17, 18]; (2) differentiation is a linear mathematical operator, and thus the gradient over a set of data points can be calculated as an aggregated value over the partial gradients computed in parallel for each point [19–25]. A variety of learning algorithms can be parallelized using these two insights. Examples include, but are not limited to, recommender systems, Kalman filters, linear and nonlinear regression models, support vector machines, least square models, logistic regression, backpropagation, softmax functions, and conditional random fields. To implement these algorithms, one needs to have (1) the partial gradient calculation function, (2) the aggregation operator, and (3) the number of data points that are processed before each aggregation. The first layer of the CoSMIC stack exposes a high-level mathematical language to programmers to specify these three constructs, which capture the entirety of the learning algorithm. The next layer of the CoSMIC stack fully automates the scale-out acceleration. The CoSMIC compiler maps and schedules the operations on the distributed accelerators. The next layer, a specialized runtime system, assigns roles and tasks for the scale-out system components and orchestrates the distributed calculation of the partial gradients and their iterative aggregation. The final layer of the CoSMIC stack provides a novel multi-threaded template architecture for the accelerators. This layer can be automatically customized and tailored according to the high-level specification of the learning algorithm and the constraints of the system.

(2) How to design customizable accelerators that efficiently exploit the large capacity of advanced process technologies. Advanced manufacturing processes have made integration of compute and storage resources on the chip. As a result, even modern FPGAs offer large capacities—e.g. Intel Arria 10 [26] instances comprise 1,518 DSP slices with 6.6 MBytes of storage and Xilinx UltraScale+ in Amazon EC2 [16] includes 6,840 DSP slices and 43 MBytes of storage. A single instance of learning algorithm may not effectively exploit resources since it is limited by the fine-grained parallelism in its Dataflow Graph (DFG). Therefore, CoSMIC offers a novel Multiple-Instruction Multiple-Data (MIMD) multi-threaded template architecture that divides the resources across multiple instances of the learning algorithm as independent threads. The last layer of CoSMIC customizes this template and generates the final accelerator by striking a balance between the number of threads running on the chip and the resources assigned to each thread. The code generation differs for FPGAs and P-ASICs. For FPGAs, the generated core is tailored to one specific learning algorithm as the chip can be erased and reprogrammed for different applications. For P-ASICs, the generated accelerator is a programmable superset of the design that fits in the area and power budget of the chip. Any algorithm that can be expressed using the DSL can be compiled and accelerated on the generated P-ASIC. The generated code and template are in the form of Register-Transfer Level (RTL) Verilog code. The template architecture is designed, optimized, and implemented by experts once in Verilog, which ensures efficiency although CoSMIC generates the accelerators automatically. More specifically, the template is designed as a two-dimensional matrix of compute units to ensure data dependencies and within-thread communications do not curtail its scalability to rather large number of processing elements. We also designed a tree-like bus to connect the rows and allocated bidirectional communication across columns. Hence, the communication latency only grows by a logarithmic order with an increase in the number of compute units, improving on-chip scalability. Furthermore, CoSMIC's backend compiler minimizes data movement by mapping operations to where their operands are located. This hardware-software co-design that aims to maximize effective resource utilization ensures effective utilization of on-chip resources, especially when they are plentiful.

(3) How to devise the system software that is specialized for distributed multi-threaded acceleration of learning. To be inline with the recent industry trends in integrating accelerators in datacenters [14–16], CoSMIC targets commodity distributed systems in which accelerators sit on the high-speed expansion slots (e.g., PCIe). For generality, we assume no special connectivity between the accelerators although such connectivity will most likely improve the benefits of CoSMIC. CoSMIC aims to best utilize the system-wide resource on both CPUs and accelerators. CoSMIC achieves this objective by offering a lean and specialized system software layer that exclusively supports learning algorithms that can be trained using parallel variants of stochastic gradient descent. This specialized layer allows the CoSMIC stack to assign the partial gradient calculation onto the accelerators while the CPUs perform aggregation and networking. This task assignment alleviates the use of accelerator resources for TCP/IP communication, avoids data copies to accelerator boards for aggregation, and enables using commodity distributed systems with CoSMIC. Moreover, it maximizes system-wide resource utilization as well as portability to different accelerator boards. Within each node, the system software maintains an internal thread pool. These threads handle the communication with the remote peer nodes. Internally managing this thread pool avoids costly OS-level context switches. The system software layer also maintains another internal thread pool that asynchronously aggregates the partial gradients. In addition, this layer assigns roles to the nodes and orchestrates the exchange of partial gradients and their aggregation.

We evaluate the benefits of the CoSMIC stack using 10 different learning applications from various domains including medical diagnosis, computer vision, finance, audio processing, and recommender systems. We compare CoSMIC against Spark, a popular framework for scale-out computing using the optimized MLlib machine learning library [27]. On average, a 16-node CoSMIC with UltraScale+ VU9P FPGAs offers 18.8× speedup over a 16-node Spark system with Xeon E3 Skylake CPUs while the programmer only writes 22–55 lines of code. When scaling the nodes from 4 to 16, CoSMIC's performance improves by 2.7×, while Spark's performance scales only

by 1.8×. We also compare the CoSMIC system with the distributed GPU (NVIDIA Tesla K40c) implementation. We report the benefits of CoSMIC for two P-ASIC implementations that match the compute resources and off-chip bandwidth of the FPGA and the GPU. On average, these P-ASICs offer 1.2× and 2.3× higher system-wide performance, while the GPU delivers 1.5× speedup over FPGA system. While using custom chips can improve computation time by 11.4×, the system-wide performance benefits are limited to 2.3×. Finally, with CoSMIC's novel multi-threaded accelerator architecture, the FPGA and the two P-ASIC systems respectively achieve 4.2×, 6.9×, and 8.2× higher Performance-per-Watt than the GPU system. These results confirm that CoSMIC is an effective and vital initial step to enable acceleration of learning at scale. To this end, this work not only contributes the full stack of CoSMIC, but also defines a new multithreaded accelerator architecture, a novel communication-aware scheduling and mapping algorithm, and a lean and specialized system software for thread management and system orchestration.

2 DISTRIBUTED LEARNING

The CoSMIC stack empowers programmers to exploit accelerator-augmented distributed systems for a wide range of learning algorithms without requiring them to deal with the laborious task of hardware design and system software programming. Although providing higher performance drives this work, programmability and generality are its other two pillars. CoSMIC facilitates programming by exposing a math-oriented DSL to programmers to express various learning algorithms as stochastic optimization problems. The layers of the CoSMIC stack compile this high-level specification to generate the accelerator architecture, and offer the system software that orchestrates them for scale-out execution. This stack is not designed for a specific ML algorithm. Instead, it is adept at accelerating learning algorithms that can be trained using variants of gradient descent optimizer. This section provides the theoretical foundation of these type of algorithms.

2.1 Learning as Stochastic Optimization

CoSMIC targets a wide range of supervised machine learning algorithms. These algorithms have two phases: training and prediction (inference). We focus on training, as it is more complex and involves several passes of prediction-tuning over the training data. Since training involves prediction, CoSMIC can accelerate prediction as well.

Each machine learning algorithm is identified by a set of parameters (θ) and a transfer function (g), that maps an input vector (X_i) to a predicted output vector (Y_i). As Equation 1 illustrates, training is the process of finding θ such that the predicted output $Y_i = g(\theta, X_i)$ has a minimum difference from the expected output Y_i^* for all input-output pairs (X_i,Y_i^*) in the training dataset.

$$\text{Find } \theta \ni \{Loss = \sum_i f(\theta, X_i, Y_i^*) = \sum_i \langle g(\theta, X_i) - Y_i^* \rangle\} \text{ is Minimized} \tag{1}$$

This unique loss function ($\sum_i f(\theta, X, Y^*)$) defines each of the learning algorithms in our target class. A machine learning algorithm learns the model (θ) by solving an optimization problem that minimizes this loss function ($\sum_i \langle g(\theta, X_i) - Y^* \rangle$). To learn a model ($\theta$), optimization algorithms iterate over the training data and gradually reduce the loss by adjusting the model parameters. One of the most common [17, 18, 50] optimization algorithm is Stochastic Gradient Descent (SGD). SGD is based on the observation that a function decreases fastest in the negative direction of its gradient.

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

As Equation 2 shows, each iteration t of SGD calculates $\theta^{(t+1)}$ by updating $\theta^{(t)}$ in the negative direction of the gradient (∂f) with a learning rate (μ). The process is repeated until the loss is minimized. The gradient function varies with the learning algorithm, while the rest of the process is fixed. Hence, our stack requires programmers to specify the algorithm by expressing the gradient of its loss function ($\frac{\partial f}{\partial \theta}$).

2.2 Parallelizing Stochastic Optimization

SGD only consumes one input-output vector (X_i, Y_i) per iteration, traversing the entire data sequentially. Thus, basic SGD is impractical for scale-out acceleration, where the training data is large and dispersed across multiple nodes. To enable scale-out acceleration, we exploit the insight that gradient is a linear operator. Therefore, the gradient over a set of data points can be computed by aggregating partial gradients calculated over partitions of this set. Different parallel variants of SGD [19–25] have been developed, which differ in how they iterate over the partitions and aggregate the partial gradients. For instance, the batched gradient descent algorithm [21] uses summation for aggregation, whereas the parallelized SGD [20] uses averaging. Equation 3 shows the use of parallelized stochastic gradient descent algorithm [20], for distributed learning.

$$\underset{j:1\to n}{\textbf{Parallel}} \langle \theta_j^{(t+1)} = \textbf{SGD}\big(\{XY_1, ..., XY_b\}, \theta^{(t)}, f\big)\rangle \tag{3a}$$

$$\theta^{(t+1)} = \frac{\sum_j \theta_j^{(t+1)}}{n} \tag{3b}$$

As shown, each node independently performs the traditional stochastic gradient descent for b input-output pairs ($\{XY_1, ..., XY_b\}$) and calculates a set of partial updates, $\theta_j^{(t+1)}$. These partial updates are aggregated with averaging, which yields the overall update ($\theta^{(t+1)}$). Equation 3a and 3b are repeated until the loss function f is minimized and the model is trained. The meta parameter b, called the mini-batch size, is the amount of local data that is processed before each aggregation step. CoSMIC expects the programmer to provide the gradient ($\frac{\partial f}{\partial \theta}$), aggregation operator (σ), and mini-batch size (b). Using only this information, CoSMIC orchestrates the scale-out acceleration of the learning algorithm. The next section discusses the accelerated execution flow and the system software layer.

3 COSMIC SYSTEM SOFTWARE

CoSMIC targets scale-out systems with commodity nodes that use off-the-shelf CPUs. Each node hosts an accelerator board, identical across all the nodes and installed on a high-speed expansion slot such as PCIe. The nodes communicate through conventional TCP/IP stack via a Network Interface Card (NIC). We choose to use commodity host systems, networking hardware-software to alleviate dependency on a particular part. To understand the specialized system software layer of CoSMIC, we first need to delve into the overall execution flow across the nodes of the scale-out system.

Execution and acceleration flow. Figure 1 illustrates a single node of the system. Each node stores a partition (D_i) of the training dataset. We have devised a multi-threaded ML accelerator for the nodes, which will be discussed in Section 5. To utilize multi-threading in the accelerator, the node further divides its data into equally sized sub-partitions (D_{i1} , ..., D_{ij}, ... D_{im}). These data sub-partitions are simultaneously processed by the accelerator. In Figure 1, each accelerator Thread_{ij} calculates its own private partial gradient ($\theta_{ij}^{(t+1)}$) by consuming a sub-partition of the training data. After the partial gradient updates are calculated, the multi-threaded accelerator aggregates

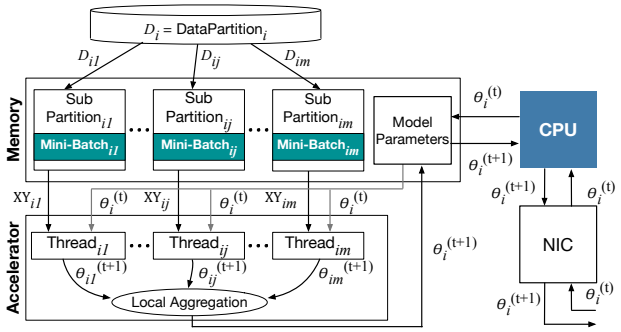


Figure 1: Execution and acceleration flow within each node.

them locally and produces the node's partial gradient update ($\theta_i^{(t+1)}$). The host CPU sends this locally-aggregated partial gradient update ($\theta_i^{(t+1)}$) to a special node that maintains the trained model parameters for a group of nodes. We refer to this special node as a Sigma node, while other nodes are called Delta nodes. The system software layer of CoSMIC performs the aggregation in a hierarchical manner to avoid overwhelming a single Sigma node. In the first level of the hierarchy, the group Sigma node calculates the group aggregate. In the next level of the hierarchy, a master Sigma node combines the aggregates. Besides aggregation, the Sigma nodes compute their own partial gradient updates, as they are also equipped with accelerators. After the aggregation, the Sigma nodes distribute the updated model parameters back to all the nodes and threads and invoke training for the next mini-batch.

Task assignment in the system software. CoSMIC offers a lean and scalable system software layer that amortizes the cost of OS-level context switches, networking, and general thread scheduling; avoids unnecessary data copies; and matches tasks to the system resources. To devise this layer, we leverage the observation that aggregation is significantly less compute intensive than partial gradient calculations. Hence, the system software layer assigns the partial gradient calculation to the accelerators, while the CPUs perform aggregation and networking. This task assignment alleviates the use of accelerator resources for TCP/IP communication, avoids data copies to accelerator boards for aggregation, and enables using commodity distributed systems. Moreover, it maximizes system-wide resource utilization and portability to different accelerator boards. To avoid extra data transfer with the memory and the host CPU, each accelerator internally aggregates the partial gradients for all its worker threads. Delta nodes send these partially aggregated gradients to their corresponding Sigma node. The system software workflow in the Sigma nodes is as follows.

Internal thread pools for networking and aggregation. Figure 2 illustrates the system software and its subroutines in the Sigma nodes. The main objective in devising these subroutines is to avoid the cost of generic thread management (creation, scheduling, and context switches) and networking by exploiting the specific execution flow of our class of learning algorithms. These subroutines need to open a socket for each communicating node. A naive approach would assign an active thread to handle each socket and spawn a thread to aggregate the received partial gradients. In contrast, the CoSMIC system software *internally* manages two thread pools, Networking Pool and Aggregation Pool as shown in Figure 2, limiting the number of active threads and reusing them as described below. When a Sigma node receives a partial update, our Incoming Network Handler catches the recv event using the Linux epoll system call. The epoll system call is effective since it does not require a linear scan on the list of monitored

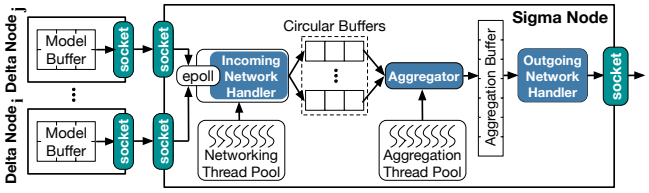


Figure 2: System software in a Sigma node.

sockets. The Incoming Network Handler assigns a thread from the Networking Pool to copy the received data from the socket buffer in the kernel space to a Circular Buffer for aggregation (Figure 2). We use Circular Buffers for concurrent networking and aggregation while each corresponding thread deals with smaller portions of data. As soon as the first chunk of data is copied, a thread from the Aggregation Pool starts processing the data and updates the Aggregation Buffer. This buffer holds the results of overall aggregation. The networking threads are data producers, while the aggregation threads are the consumers. Since Sigma nodes communicate with multiple other nodes, this approach uses the multi-threading capabilities of the CPUs to improve concurrency. The Circular Buffer reduces the memory required for aggregating partial results from multiple sources while enabling overlap between communication and computation. Our internally managed thread pools (1) alleviate the need to create an active thread for each connection, limiting the number of active threads; (2) reuse threads for different connections, mitigating the cost of context switching; and (3) use a producer-consumer semantics between the two thread pools, specializing their scheduling. These techniques avert the cost of generic thread management (creation, scheduling, and context switches), which is oblivious to the execution flow of machine learning.

4 THE COSMIC STACK

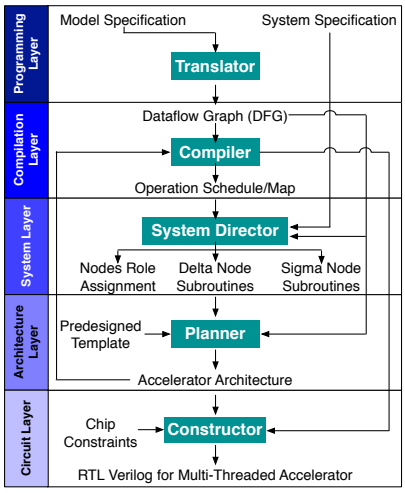
Figure 3 illustrates the layers of the CoSMIC stack and their interworking that orchestrates Sigma and Delta nodes and enable scale-out acceleration. This section discusses each layer briefly.

4.1 Programming Layer

Our stack makes the accelerator-augmented scale-out systems programmable from a high-level DSL. With CoSMIC, programmers use our extension of the high-level language, developed in the prior work [12] that focuses on *single-FPGA* acceleration of learning. We chose to extend this DSL since it has a one-to-one mapping to mathematical formulations instead of providing linear algebra primitives as proposed in the past [51]. Moreover, it is open source and publicly available (http://act-lab.org/artifacts/tabla/). Using the extended language, programmers express the mathematical formula of the partial gradient and the aggregation operator in a textual format. Additionally, the programmer declares the mini-batch size. Figure 4(a) illustrates how a programmer uses our stack to accelerate the training of a binary classifier based on support vector machines. The first part of the code is the textual representation of Equation 4.

$$Gradient_i = \begin{cases} -y \times X_i, & \left(\left(\sum_i X_i \times W_i\right) \times y\right) > 1 \\ 0, & \left(\left(\sum_i X_i \times W_i\right) \times y\right) \leq 1 \end{cases} \quad (4)$$

The code has three segments: data declarations, gradient formulation, and aggregator specification. The DSL provides five data types: model_input, model_output, model, gradient, and iterator. These types denote the semantics of the variables in learning algorithms, and the statements represent the mathematical operations. For instance, the $\sum_i X_i \times W_i$ term in Equation 4 is implemented as sum[i](w[i] * x[i]), where x and w are declared as model_input and model, respectively.

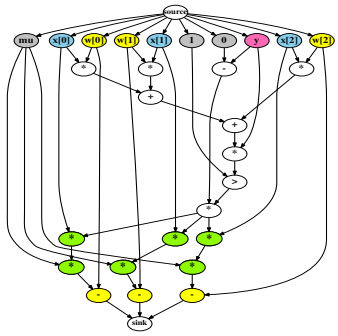


Programming Layer	Algorithmic Specification	Partial Gradient
		Aggregation Operator
		Mini-Batch Size
	System Specification	Number of Nodes
		Number of Groups
		Accelerator Type
Compilation Layer	Translator	Dataflow Graph (DFG)
	Compiler	Operation Schedule/Map
System Layer	System Director	Node Roles
	System Subroutines: Delta Nodes	Accelerator Invocation Module
		Module for Communication with Sigma Node
	System Subroutines: Sigma Nodes	Accelerator Invocation Module
		Networking Thread Pool for Communication with Delta Nodes
		Circular Buffer for Consumer-Producer Networking & Aggregation
		Aggregation Thread Pool
		Module for Communication with Next Level of Hierarchy Node
Architecture Layer	Hand-Optimized Template Design	RTL Verilog
	Performance Estimation Tool	Design Space of Possible Architectures
	Planner	Number of Threads
		Resources per Thread
		Accelerator Datapath
Circuit Layer	Constructor	RTL Verilog of the Multi-Threaded Accelerator

Figure 3: The full CoSMIC stack.

```
1  mu = 0.01; // learning rate
2  m = 3; // num of features
3  minibatch_size = 10000;
4
5  model_input x[m];
6  model_output y;
7  model w[m];
8  gradient g[m];
9  iterator i[0:m];
10
11 h = sum[i](w[i] * x[i]);
12 c = y * h;
13 g[i] = ((c > 1) * (0 - y)) * x[i];
14
15 n = 10; // number of nodes
16 aggregator(n) {
17     iterator j[0:n];
18     w[i] = (sum[j](w[i])) / n; }
```

(a)



(b)

Figure 4: (a) Programmer specifies the classification algorithm as its gradient and aggregation functions. (b) Translator outputs the DFG.

The iterator i represents the subscript in $\sum_i$. The aggregation function of the parallelized SGD, which averages the partial gradients, is specified by w[i] = sum[j](w[i]) / n. This high-level expression is then converted to a Dataflow Graph (DFG) by the Translator (Figure 4(b)).

4.2 Compilation Layer

In a conventional computing stack, the next natural step after translation would be compilation. However, in our specialized stack, the order of the steps is different since the architecture of the accelerator has not yet been solidified. First, the Planner (from the architecture layer) needs to produce the architectural plan of the accelerator. In the FPGA case, this plan even depends on the DFG of the learning algorithm. In the P-ASIC case, although this plan is not dependent on the DFG, it still changes according to the chip constraints. The back-edge from the architecture layer to the compilation layer in the left diagram of Figure 3 illustrates the dependence of Compiler to the Planner. Once the architecture is planned, the Compiler leverages our novel mapping/scheduling algorithm to statically map operations to the accelerator Processing Engines (PEs). This static mapping is converted to state machines and control units that are embedded in the accelerator code for FPGA realization. For P-ASIC, the mapping is converted to microcodes. This static scheduling strategy avoids the von Neumann overheads and significantly simplifies the hardware which is necessary for the efficiency of the accelerator. As detailed in Section 6, our mapping/scheduling algorithm also minimizes on-chip communication and alleviates the need for data preprocessing or marshaling. Compiler also generates the schedule for the template architecture's programmable memory interface that feeds a large number of PEs and streams data in without the need for PEs to request the data.

4.3 System Layer

Section 3 already detailed the system layer. The topmost component of this layer is the System Director that assigns roles (Sigma or Delta) to the nodes and then configures and initiates the corresponding system subroutines. This role assignment is based on the system specification, which includes the total number of nodes, the number of groups, and the accelerator type (Figure 3, right).

4.4 Architecture Layer

In the conventional stack, this layer defines the Instruction Set Architecture (ISA) of a microprocessor. In CoSMIC, this layer is responsible for planning the architecture of the accelerator in accordance with the constraints of the target platform. The plan is generated with respect to our novel multi-threaded template architecture, which is a parametric RTL Verilog of customizable design. This template architecture can accelerate multiple instances of the partial gradients simultaneously. However, it is not specific to a learning algorithm and can be shaped according to the constraints of the acceleration platform (e.g., area) and the DFG of the algorithm in the case of FPGA acceleration. Instead, it is a two-dimensional matrix of customizable PEs that this layer needs stretches or squeezes in either dimension to match the chip specifications. The main challenge is allocating the chip resources in such a way that strikes a balance between the single-threaded performance and multi-threaded parallelism. The Planner is responsible for this balanced plan by determining how many threads will be accelerated simultaneously; how many PEs will be allocated to each thread; and how the PE will be arranged in the 2D matrix of the accelerator. For P-ASICs, the Planner determines the largest number of PEs that fits in the area and power budget of the target chip. However, this metric depends on the PE buffer capacity that is decided according to a set of benchmarks. After determining the total number of PEs, the Planner steps are similar for P-ASICs and FPGAs. Thus, we only discuss the Planner in the context of FPGAs for brevity.

To determine these factors, the Planner takes in a high-level specification of the FPGAs, which includes the number of DSP units, the

off-chip memory bandwidth, the number of on-chip Block RAMs (BRAMs), and the size of each BRAM (Figure 3). The first step is determining the number of columns (=# PEs in a row) and rows. The Planner uses the off-chip memory bandwidth to first set the number of columns equal to the number of words that can be fetched in parallel from memory (=off-chip bandwidth). Having fewer columns would waste bandwidth, while more would increase pressure on the internal interconnection between the PEs. The Planner will then determine the maximum row count as $row_{max} = \frac{\#\,\text{DSPs}}{\#\,\text{of Columns}}$.

Next, the Planner determines the number of threads and their PE allocation through design space exploration. However, this design space is prohibitively large, due to the copious amount of resources in the modern FPGAs. We prune this design space through the following intuitive design decisions. The Planner first calculates the amount of required storage and area for accelerating one worker thread based on its DFG. The ratio of total on-chip storage and area to this thread's footprint will be the upper bound on the number of simultaneous threads. Then, we restrict the PE allocation to the row granularity, meaning each thread will have at least a row of PEs. Another parameter that affects the maximum number of threads is the programmer-provided mini-batch size, as it determines how many parallel threads can potentially be launched. The minimum of these parameters is the maximum number of possible threads ($t_{max} = min\left(\frac{\#\,\text{BRAMs} \times \text{BRAM Size}}{\text{DFG.storage()}}, row_{max}, \text{Mini-Batch Size}\right)$).

These design choices and the column/row arrangement restrict the design space from which the Planner needs to determine the optimal allocation of PEs to the threads. For instance, in UltraScale+, the design space is limited to 27 design points. However, the Planner still needs to explore this reduced design space. Instead of simulation, which will be intractable, we propose to equip the Planner with a performance estimation tool. The tool will use the static schedule of the operations for each design point to estimate its relative performance. This enables the Planner to choose the smallest, best-performing design point which strikes a balance between the number of cycles of data processing and off-chip data transfer. Performance estimation is viable, as the DFG does not change, there is no hardware managed cache, and the accelerator architecture is fixed during execution. Thus, there are no irregularities that can hinder estimation. As such, it takes less than five minutes to explore all the possible design points for UltraScale+. The result of this design space exploration is presented in Section 7. After this analysis, the Planner generates the Verilog code of the accelerator datapath from the template.

4.5 Circuit Layer

As Figure 3 depicts, the Constructor is the main module of the Circuit layer and generates the final Verilog code by adding the control logic. In the case of FPGAs, to generate the state machines and control units, the Constructor needs the Compiler to first statically map and schedule all the operations. In this case, the accelerator avoids the von Neumann overhead by bypassing instruction fetch and decode stages. Instead, the Constructor statically converts the execution schedule to state machines and control logic. In the case of P-ASICs, the Constructor adds a control logic that enables microcode execution on the PEs. Then, it inserts these control units within the datapath Verilog code generated by the Planner and produces the final synthesizable Verilog code of the accelerators. The Planner, the Constructor, and the Compiler work in tandem to make CoSMIC a cohesively co-designed stack that delivers high gains.

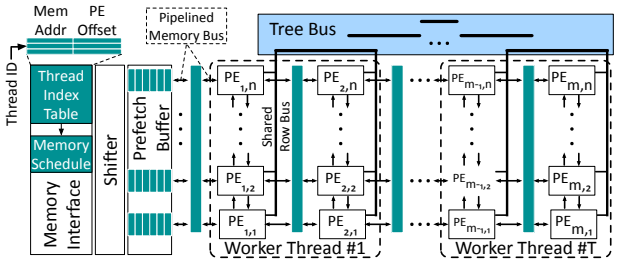


Figure 5: CoSMIC Multi-Threaded Template Architecture.

5 TEMPLATE ARCHITECTURE

A major challenge in acceleration is the generality across a wide range of algorithms and applications while supporting a variety of platforms (e.g., various FPGA chips). It is also crucial to offer a solution that can adapt to new algorithms and algorithmic changes. A fixed architecture cannot offer enough flexibility and is not deployable on different chips. Therefore, CoSMIC offers a template architecture to accelerate learning at scale. This template is predesigned, yet re-organizable, providing the capability to implement different gradient calculations and parallel variants of gradient descent aggregations and updates. The template offers reusability while delivering high performance, as it is hand-crafted by experts (e.g., our team). Our stack stretches and squeezes the template to best match the DFGs and the target chip. Hence, it is modular and scalable to maximally utilize the ample amount of resources in the server-grade FPGAs and P-ASICs.

The need for multi-threading. A single instance of a learning algorithm cannot effectively exploit as much resources, since it is limited by the level of parallelism in its DFG. The DFG of the partial gradient update dictates the number and type of operations, along with data-dependencies. However, data-dependencies in the DFG limit the number of operations that the accelerator can execute in parallel. To increase the parallelism available to the accelerator, we use the insight that partial gradient updates generated by worker threads in parallel gradient descent algorithms are independent. As such, the CoSMIC template architecture executes multiple worker threads in the FPGA accelerator; each thread, using a subset of the accelerator resources, executes the entire DFG over the thread's data sub-partition to generate an independent partial gradient update. This multi-threading limits the data-communication within a worker thread to a subset of the accelerator's DSP slices, reducing communication overhead.

5.1 Accelerator Organization

As depicted in Figure 5, the template architecture constitutes: (1) the memory interface—to transfer data to and from external memory; (2) the shifter—to align the data coming from memory; (3) the prefetch buffer—to store the aligned data; and (4) the two-dimensional array of PEs—to compute partial gradient updates and locally aggregate them. We choose this 2D topology, because it enables the Planner to modularly add or remove PEs as columns or rows. As discussed, this organization also enables an efficient design space exploration by assigning PEs to the worker threads in the rows granularity.

Connectivity and bussing. As Figure 5 shows, the number of PEs in each row of the template matches the off-chip bandwidth so that the memory interface can feed all the PEs in a row every cycle, maximizing parallelism. Each row of PEs connects to the memory interface using a pipelined bus, as shown in Figure 5. Pipelining the bus is necessary for scalability since the bus is shared by all the rows in the accelerator. In addition to data transfer between external memory and

the PEs, connectivity between PEs is required to transfer intermediate results due to data-dependencies in the DFG. To facilitate the communication, PEs in a single row are connected to their adjacent PEs using bi-directional links and are also connected to the other PEs in the row via a shared bus. A hierarchical tree bus connects the shared bus for different rows. We specialize the interconnect between PEs in the template architecture for communication patterns typical for operations in stochastic gradient descent based learning algorithms. One such example of a common operation is a vector dot product, which involves element-wise multiplication followed by reduction ($\sum$). The result is then typically communicated to all PEs. While the PEs can execute the element-wise multiplication in parallel, the reduction and broadcast operations require significant communication between PEs, which can be a performance bottleneck. In order to alleviate the communication overhead and ensure high utilization of the accelerator's resources, PEs possess three distinct levels of connectivity. Figure 5 shows these three levels of connectivity for the template architecture with (n) PEs per row and (m) rows. At the first level, the n adjacent PEs within each row can communicate using bi-directional links. Next, a shared bus connects all of the n PEs within each row. Finally, we use a tree bus to connect the shared bus of m rows of the accelerator. To further aid the reduction operation, each node in the tree bus contains an ALU to perform $\sum$ and $\prod$ operations.

PE design. Figure 6 details a PE, the basic unit of the template architecture responsible for executing the operations of the DFG. The rows of PEs within a worker thread exploit fine-grained parallelism in the DFG, enabling the execution of multiple independent operations in parallel. A PE consists of separate buffers for storing training data, model parameters, and intermediate results. This partitioning of buffers is necessary to enable parallel accesses required for DFG operations. The buffers are composed of on-chip SRAMs and the size of each buffer can be configured by the Planner for a given DFG. CoSMIC's Compiler statically generates the schedule of operations for each PE. The PEs execute the scheduled operations using a five stage pipeline, orchestrated by a PE scheduler. The first pipeline stage reads the required data from PE's buffers, adjacent PE links, and shared bus links. This data is registered in the second stage. The third stage selects the input operands required by the scheduled operation. The fourth stage executes the scheduled operation using the PE's ALU. For FPGA implementation, the ALU uses DSPs blocks—the hardened on-chip arithmetic unit on the FPGA. The non-linear unit is a look-up table that implements expensive operations like sigmoid, gaussian, divide, and logarithm and is only instantiated in a PE if the Compiler schedules a non-linear operation for that PE. The output of the ALU unit is written back in the fifth and final stage of the PE pipeline. The PEs have a bypass path between the final stage and the ALU stage to forward the result of the previous operation. Figure 6 highlights the path taken by an add operation which reads from data and model buffers and writes back to the interim buffer.

Memory interface. Simplicity of the PEs and their highly pipelined design is vital for the efficiency of the accelerator. To further simplify the design, the template architecture prevents the PEs from initiating data requests to the memory. Instead, as illustrated in Figure 5, the design harbors a smart memory interface which feeds the PEs according to the schedule generated by the Compiler. This memory interface design is intended to alleviate the overhead of data marshaling, which would have been prohibitive since CoSMIC targets distributed learning with copious amounts of data. However, one issue that arises is that the vectors of data in the off-chip memory do not necessarily align

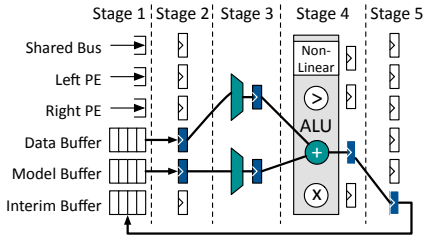


Figure 6: Pipelined PE. Black highlights an Add operation (InterimBuffer[i] = DataBuffer[j] + ModelBuffer[k]).

with the rows of the PEs. This can lead to under-utilization of off-chip bandwidth, which is often a performance bottleneck. To avoid the overhead of padding the data to align with the PEs, we propose to use an on-chip Shifter that aligns input data after fetching it, according to the data map generated by the Compiler. In addition to the Shifter, the memory interface will have a Prefetch Buffer. The size of the training data for each DFG is often large. Hence the time required for external memory access is significant. The Prefetch Buffer enables the accelerator to store the subsequent set of training data for the worker threads, thereby hiding the latency of memory accesses and enabling efficient MIMD execution. The memory interface can also perform broadcast writes to the PEs, as the same model needs to be sent to all the worker threads before they start calculating the new gradient updates.

5.2 Multi-Threaded Acceleration

The programmable memory interface plays a significant role in enabling multithreading in the accelerator without imposing significant hardware overhead. It harbors a Memory Schedule queue along with a Thread Index Table that stores thread-specific information as depicted in Figure 5. This information includes the memory address of each thread's data sub-partition and the base index of the first allocated PE row to the thread. In addition, each thread has its own dedicated pointer to the Memory Schedule queue. The data transfer schedule is the same for all the threads but it needs to start from different addresses and write to different PEs. The Thread Index Table enables correct and efficient data transfer from memory to all the threads while the schedule is shared. Each row of the table corresponds to one thread. The first field in each row is Mem Addr, which specifies the starting address of each thread's data sub-partition in the off-chip memory. The second field, PE Offset, specifies the index of the first PE of the thread. By walking through these rows, the memory interface controller uses the entries of the Memory Schedule and the Thread Index Table to generate memory accesses for each thread in a round-robin fashion. Each entry of the schedule stores a Base PE Index, RD/WR bit, Broadcast bit, and Size. The index of the target physical PE is (Base PE Index + PE Offset). The latter term in the addition comes from the Thread Index Table. The memory address is also obtained from the Thread Index Table, which is updated by the size of the transferred data after it finishes. Using this table, the memory interface has the necessary information to transfer each thread's data to its allocated PEs without the need for storing multiple copies of the memory schedule. The RD/WR bit of the memory schedule entry specifies whether the memory access is a read or a write. The Broadcast bit allows a memory read to be sent to all the worker threads via the memory interface bus. This bit is particularly useful when sending model parameters from memory to all worker threads. The Size specifies the size of the data transfer. The Compiler generates the memory schedule according to the Planner-provided architecture and the DFG. The following section discusses the Compiler in detail.

6 COSMIC COMPILATION

The Compiler is a critical layer of CoSMIC, since it statically determines a fine-grained map and schedule of all the data and operations, which significantly simplifies the hardware. This simplification is necessary for acceleration, particularly for FPGAs that incur lower frequency when design complexity grows. Furthermore, the Compiler minimizes on-chip and off-chip communication and avoids data preprocessing or marshaling. Avoiding data marshaling is crucial, since the accelerators process large amounts of data and any data transfer is costly. To this end, we propose an algorithm that minimizes data movements by statically mapping data elements to PEs before mapping the operations. Conventional mapping algorithms [12, 52] map operations before the data to find the lowest-latency schedule which adheres to the on-chip resource constraints. In contrast, we reverse the order of mapping, thereby minimizing data movement atop latency.

The Compiler takes as input the DFG of the gradient update, the architectural plan of the multi-threaded accelerator, and the data layout of the training dataset and model parameters in the memory. Using these inputs, the Compiler generates the following for each thread:

(1) Data map: assignment of inputs, outputs, model parameters, and intermediate values to the PEs.
(2) Operation map: assignment of all the DFG operations to PEs.
(3) Data transfer schedule: detailed schedule for memory interface and interconnection buses to send data to the appropriate PEs.

To generate the data map, the Compiler first segregates the DFG operands (graph edges) into DATA, MODEL, and INTERIM categories. These categories represent training data, model parameters, and intermediate operands, respectively. This semantic segregation enables the Compiler to provide an optimal data map without marshaling the data as follows. It starts by mapping each training data element (type DATA) to the PE that is connected to the memory interface column which brings in that element. The Compiler uses this data map to generate the schedule of data transfer from off-chip memory and embeds it into the memory interface. This map and schedule avoids marshaling by adhering to the layout of training data in the memory. Next, the Compiler generates the operation map and data map for the model parameters while minimizing the communication between PEs. We have designed Algorithm 1 for the Compiler to map the operations to the same PEs that hold their operands; hence minimizing inter-PE communication. This algorithm also maps the model parameters to the PEs that hold their corresponding operation. The intuition is to map the MODEL and INTERIM edges on to the same PE if a node operates on both of them. After determining the data map on the PEs, the algorithm traverses the DFG and map operations according to the location of their operands, minimizing data movement. During this pass, to reduce latency, the Compiler also prioritizes scheduling operations that have the longest dependence chain. The algorithm takes in the DFG (G) and the number of PEs per thread (n_{PE}) and goes through the following steps:

(1) **Initialize** the operation map ($O[n_{PE}]$) and the data map ($D[n_{PE}]$) to null and the *Graph* variable to the DFG (G). O and D are arrays of lists that hold the maps for each PE.
(2) **Select a vertex** (v) that is ready i.e. all its predecessors are mapped.
(3) **Check the operand type** for this vertex (v). If any of its operands (op_i) is of type DATA, then map v to the PE containing this data, else go to step (4). Check the type of the other operand (op_j). If the other operand (op_j) is of type MODEL, then map this model parameter to v's PE and go to step (5).

```
Input  : G: Dataflow graph (V,E)
         n_PE: Number of PEs per worker thread
Output : O: Operation map
         D: Data map
Initialize O[n_PE] ← ∅
Initialize D[n_PE] ← ∅
Initialize Graph ← G
PE_i = 0
while (graph ≠ ∅) do
    for (v ∈ Graph) do
        if (∀ p_i in v.parents = MAPPED) then
            if (∃ op_i in v.ops & op_i.type = DATA) then
                v.pe = op_i.pe
                if (∃ op_j in v.ops & op_j.type = MODEL) then
                    D[v.pe].append(v.op_j)
                Break
            else if (∃ op_i in v.ops & op_i.type = MODEL) then
                if (op_i.pe != NULL) then
                    v.pe = op_i.pe
                else
                    v.pe = w_i
                    D[v.pe].append(v.op_i)
                    PE_i = (PE_i + 1) % n_PE
                Break
            else if (∃ op_i in v.ops & op_i.type = INTERIM) then
                v.pe = op_i.pe
                Break
            O[v.pe].append(v)
            graph.remove(v)
    end
end
```

Algorithm 1: Minimum-Communication Data/Operation Mapping.

(4) **If operand type of the vertex (v) is MODEL**, then map v to the PE where the model parameter resides, otherwise go to step (5). If the operand is not mapped, then map this vertex and the operand op_i to a new PE (PE_i). The PE_i variable is a counter, incremented after each round of successful mapping. Incremental assignment enables parallel execution of the operations in neighboring PEs.
(5) **If operand type of the vertex (v) is INTERIM**, map the vertex(v) to the PE in which the operand resides.
(6) **Reiterate steps 2 through 5** until all the vertices are mapped.

Given the data and operation map, the Compiler generates the execution schedule for all the components of the accelerator, including its programmable memory interface and PE interconnects. Recall that each thread performs the same gradient update rule but uses different training data. Therefore, the Compiler generates the map and schedule for one thread and use it for all of them. However, to overlap off-chip data transfer with computation, the accelerator is MIMD, not SIMD. Thus, threads can be at different computation stages since they start execution as soon as they receive an operand. To enable the MIMD execution, the Planner produces a PE Offset for each thread, which is the index of the first PE that is assigned to the thread. The PE Offset and the starting address of its training data is loaded into the Thread Index Table as discussed before (see Figure 5). The Compiler generates only one schedule for the memory interface since the destination PE can be calculated at runtime by adding each thread's PE Offset to the PE index that is in the schedule. Finally, the Compiler uses the map of the model parameters to generate the schedule for the aggregation stage that follows partial gradient calculations.

7 EVALUATION

We evaluate CoSMIC with 10 different machine learning benchmarks using various acceleration platforms, which consist of one FPGA (Xilinx UltraScale+ VU9P) and two P-ASICs. These accelerators

Table 1: Benchmarks, algorithms, application domains, and datasets.

Name	Algorithm	Domain	Description	# Features	Model Topology	Model Size (KB)	Lines of Code	# Input Vectors	Input Data Size (GB)
mnist	Backpropagation	Image Processing	Handwritten digit pattern recognition	784	784×784×10	2,432	55	60,000	0.4
acoustic		Audio processing	Hierarchical acoustic modeling for speech recognition	351	351×1,000×40	1,527	55	942,626	5.6
stock	Linear	Finance	Stock price prediction	8,000	8,000	31	23	130,503	14.7
texture	Regression	Image Processing	Image texture recognition	16,384	16,384	64	23	77,461	17.9
tumor	Logistic	Medical Diagnosis	Tumor classification using gene expression microarray	2,000	2,000	8	22	387,944	10.4
cancer1	Regression	Medical Diagnosis	Prostate cancer diagnosis based on the gene expressions	6,033	6,033	24	22	167,219	13.5
movielens	Collaborative	Recommender System	Movielens recommender system	30,101	301,010	1,176	42	24,404,096	0.6
netflix	Filtering	Recommender System	Netflix recommender system	73,066	730,660	2,854	42	100,498,287	2.0
face	Support Vector	Computer Vision	Human face detection	1,740	1,740	7	27	678,392	15.9
cancer2	Machine	Medical Diagnosis	Cancer diagnosis based on the gene expressions	7,129	7,129	28	27	208,444	20.0

are hosted in machines equipped with Intel Xeon E3 v5 processors. We first compare the scalability of the FPGA-accelerated CoSMIC systems to a popular distributed computing platform, Spark [1], while increasing the number of nodes from 4 to 8 to 16. For the scale-out experiments, we used Amazon EC2. We built a local three node system for the in-depth sensitivity studies. We also perform comparison with the distributed GPU (Nvidia K40c) implementation of the benchmarks. Table 2 details the specification of these platforms. Lastly, we compare the CoSMIC template architecture with TABLA [12], a single-node FPGA acceleration framework for ML.

7.1 Methodology

Benchmarks and training input datasets. Table 1 shows the list of 10 benchmarks—obtained from machine learning literature—that train two different models with each of the following five different algorithms: backpropagation, linear regression, logistic regression, collaborative filtering, and support vector machines. The benchmarks represent various application domains including image processing, audio processing, finance, medical diagnosis, recommendation systems, and computer vision. The mnist and acoustic benchmarks train Multi-Layer Perceptrons (MLPs) for handwritten digit [53, 54] and automatic speech recognition [55], respectively. The stock benchmark trains a linear regression model to predict stock prices using the tick-level data points [56]. The texture benchmark trains another linear regression model for texture recognition [57]. The tumor and cancer1 benchmarks train two different logistic regression models to detect tumors [58] and cancer [59] using the microarray gene expression data. The movielens and netflix benchmarks train recommender systems that employ the collaborative filtering algorithm on Movielens datasets [60, 61] and Netflix Prize Dataset [62]. The face benchmark trains a support vector machine for face recognition [63]. The cancer2 benchmark trains another support vector machine to detect cancer [63]. We train each benchmark for 100 epochs over its dataset. We repeat the experiments 10 times and use the average runtime. In Table 1, the "# of Features" column shows the number of elements in each training data vector and the "Model Topology" column denotes the model topology of each benchmark. The "Model Size" column shows the size of the model parameters. The "Lines of Code" column lists the number of lines of code that the programmer writes, which ranges from 22 to 55. Finally, the "# of Input Vectors" and "Input Data Size" columns show the number of the training vectors and the size of the training data. The model parameters for all the benchmarks fit in on-chip memory of the FPGA and the P-ASICs.

Scale-out system specification. Both CoSMIC and Spark systems are deployed on a cluster of machines, which are equipped with the high-performance quad-core Intel Xeon E3 Skylake processors with hyper-threading support that operates at 3.6 GHz. The detailed CPU specification is provided in Table 2. The machines run Ubuntu 16.04.1 LTS with the kernel version 4.4.0-47. The machines are connected through a TP-LINK 24-Port gigabit Ethernet switch (TL-SG1024) via TP-Link gigabit Ethernet network interface card (TG'-3468). The switch supports full duplex operation on all ports (2 Gbps per port) and a combined switching capacity of up to 48 Gbps.

Spark. We compare CoSMIC with Spark version 2.1.0. Spark is selected as the point of comparison since it supports efficient in-memory processing for iterative applications. Moreover, Spark provides the MLlib [27] machine learning library. The Spark MLlib library provides the baseline implementation for backpropagation, linear regression, logistic regression, collaborative filtering, and support vector machines [27]. To optimize the performance of MLlib, we build Spark with vectorized OpenBLAS library. For all the Spark results, we use the best-performing combination of machines and threads. The best number of threads is selected for each benchmark individually.

FPGA. As Table 2 shows, we use Xilinx Virtex UltraScale+ VU9P for the FPGA experiments. We use Xilinx Vivado 2017.2 to synthesize the generated accelerators at 150MHz. The synthesized accelerators are connected to the external DRAM using the AXI-4 IP.

GPU. For comparison with GPUs, we extend CoSMIC's runtime system to support GPUs since Spark does not. The alternative would have been integrating GPUs with Spark, which is on its own a line of ongoing research [64–67]. As such, we build a GPU-accelerated CoSMIC system. We had three Nvidia Tesla K40 GPUs at our disposal, which are used for this comparison (see Table 2 for hardware specification). For the GPU experiments, we developed highly optimized CUDA implementations using well-known libraries, including LibSVM-GPU [68] and Caffe2+cuDNN [69], as well as source code from related works [6, 12]. In all cases, we used the latest versions of each library (e.g., cuBLAS v8.0 [70] and cuDNN v7.0 [71]). We use WattsUp [72] to measure the system power following the same methodology in the prior work [73].

P-ASICs. We use Synopsys Design Compiler (L-2016.03-SP5) and TSMC 45-nm high-Vt standard cell libraries to synthesize the CoSMIC-generated architectures and obtain the area, frequency, and power results. We used CoSMIC to generate two P-ASIC designs: one with the PE count and off-chip bandwidth that match those of the

Table 2: CPU, GPU, FPGA, and P-ASICs.

	CPU	GPU
Chip	Xeon E3-1275 v5	Tesla K40c
Cores	4	2,880
Memory	32 GB	12 GB
TDP	80 W	235 W
Frequency	3.6 GHz	875 MHz
Technology	14 nm	28 nm

	FPGA
Chip	UltraScale+ VU9P
DSP Slices	6,840
BRAM	44,280 KB
TDP	42 W
LUTs	1,182 K
Flip Flops	2,364 K

	P-ASIC	P-ASIC
Chip	F	G
PEs	768	2,880
Area (mm^2)	29	105
Power	11 W	37 W
Frequency	1 GHz	1 GHz
Technology	45 nm	45 nm

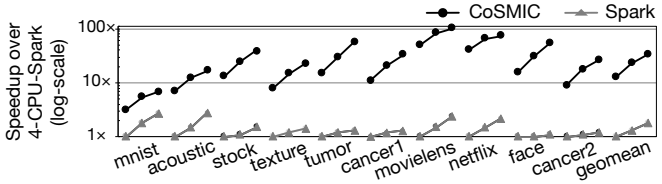
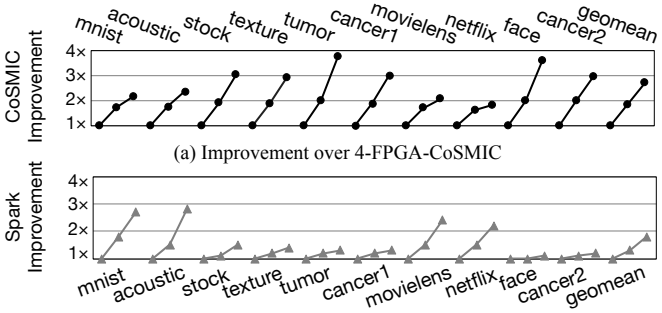


Figure 7: Speedup over Spark as the number of nodes increases from 4 to 8 to 16. Baseline: Spark system with 4 nodes (4-CPU-Spark).



(a) Improvement over 4-FPGA-CoSMIC

(b) Improvement over 4-CPU-Spark

Figure 8: Scalability comparison of CoSMIC and Spark as the number of nodes increases from 4 to 8 to 16.

FPGAs (P-ASIC-F), the other that match those of the GPUs (P-ASIC-G). Table 2 provides the details of these P-ASICs. We combine the system-level measurements with the synthesis and simulation/estimation results to evaluate these P-ASICs.

7.2 Experimental Results

Performance comparison. Figure 7 shows the result of performance comparison between CoSMIC and Spark using three system configurations: 4-Node, 8-Node, and 16-Node. The baseline is a 4-Node Spark system, referred to as 4-CPU-Spark. On average, the 4-FPGA-, 8-FPGA-, 16-FPGA-CoSMIC configurations deliver 12.6×, 23.1×, and 33.8× higher performance, respectively. Whereas, increasing the number of nodes with Spark from 4 to 16 only yields 1.8× performance improvement. The performance does not scale linearly as the number of nodes increases due to system management overhead in networking and aggregation. The performance gains for different benchmarks depend on their model topology, parallelism, and memory footprint. For example, movielens (collaborative filtering) sees the highest speedup (100.7×) since its DFG is significantly parallel that allows CoSMIC to utilize the FPGAs resources for higher performance. On the contrary, mnist and acoustic (backpropagation) achieve relatively smaller speedup (6.8× and 16.5×) since these benchmarks require significant on-chip communication, which bottlenecks performance. These results show that CoSMIC's full-stack approach, which comes with our multithreaded accelerators, is highly effective for the scale-out acceleration of these ML applications. Furthermore, these results show that CoSMIC better utilizes the added resources and is more scalable as the number of nodes increases.

Scalability. To better compare the scalability of the two systems, Figure 8 shows the performance improvement over each system's own 4-Node configuration. Figure 8(a) shows the improvement with CoSMIC when the 4-FPGA-CoSMIC is the baseline and Figure 8(b) shows the improvement with Spark when 4-CPU-Spark is the baseline. On average, CoSMIC performs 1.8× and 2.7× faster when the system

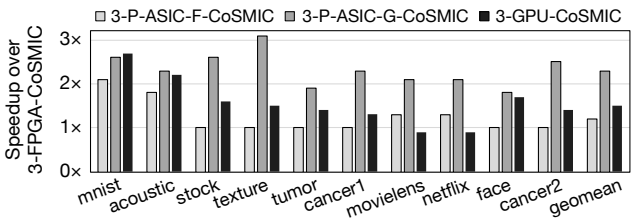


Figure 9: System-wide speedup over 3-FPGA-CoSMIC.

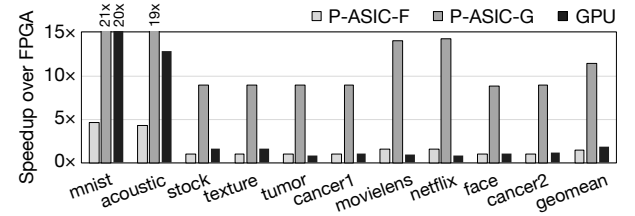


Figure 10: Computation speedup over FPGA.

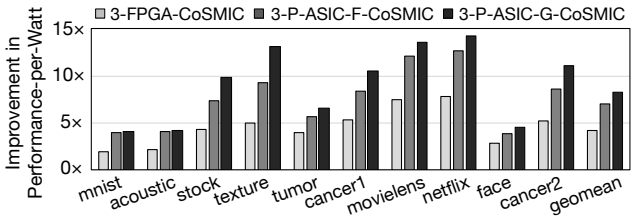
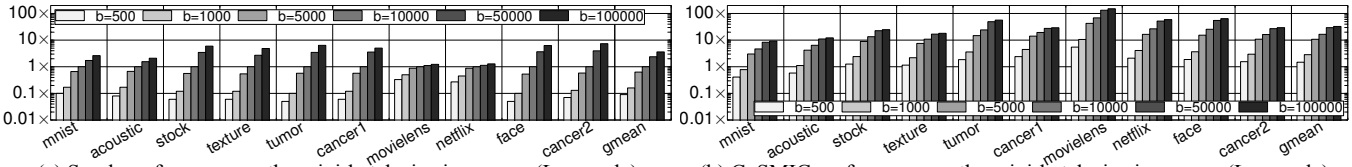


Figure 11: Performance-per-Watt, baseline: 3-GPU system.

is scaled up to 8 and 16 nodes, respectively. As a point of reference and comparison, Spark shows 1.3× and 1.8× speedup for the same increase in the number of nodes. The results from Figure 7 and Figure 8 show that CoSMIC scales well and better than Spark as the number of nodes increases. The improvement gap between Spark and CoSMIC is larger for the benchmarks that have higher ratio of communication to computation in the runtime (stock, texture, tumor, cancer1, face, and cancer2). For the other benchmarks, CoSMIC scales less steeply in comparison to Spark. These benchmarks are compute-bound and therefore acceleration is effective and adding accelerators reduces the computation time in the baseline 4-Node configuration. Since Spark does not utilize the accelerators, it benefits more from the added nodes as they bring in the necessary compute power that was missing in the 4-Node configuration. Therefore, adding more nodes helps but it is more effective for Spark. Nonetheless, as Figure 7 illustrates, CoSMIC significantly outperforms Spark across all the benchmarks. These results confirm that the specialization of the system software has been effective in enabling acceleration at scale.

Comparison of different acceleration platforms. Figure 9 compares the benefits of CoSMIC with FPGAs and P-ASICs to GPUs. The results are obtained from our three-node system configuration and the baseline is the 3-FPGA-CoSMIC. On average, the 3-P-ASIC-F-CoSMIC, 3-P-ASIC-G-CoSMIC, and 3-GPU-CoSMIC systems provide average 1.2×, 2.3×, and 1.5× higher performance than the 3-FPGA-CoSMIC system, respectively. Although as expected P-ASICs and the GPU outperform the FPGA, the benefits are relatively modest. To understand this trend, Figure 10 shows the improvement in compute time without considering the system software. On average, P-ASIC-F, P-ASIC-G, and GPU perform 1.5×, 11.4×, and 1.9× faster than FPGA, respectively. Except for mnist and acoustic benchmarks, which use the backpropagation algorithm, the benefits from P-ASIC-F and GPU are not overwhelming. GPU provides higher speedup on two specific benchmarks (20.3× for mnist and 12.8× for acoustic) as the



(a) Spark performance as the mini-batch size increases. (Log-scale)

(b) CoSMIC performance as the mini-batch size increases. (Log-scale)

Figure 12: Performance vs. mini-batch size as it is swept from 500 to 100,000; baseline: 3-node Spark when the mini-batch size is 10,000.

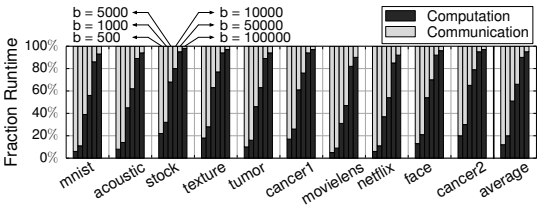


Figure 13: Fraction of 3-FPGA-CoSMIC runtime.

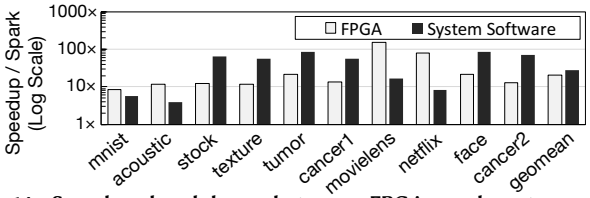


Figure 14: Speedup breakdown between FPGAs and system software (aggregation, networking, and management) for 3-FPGA-CoSMIC.

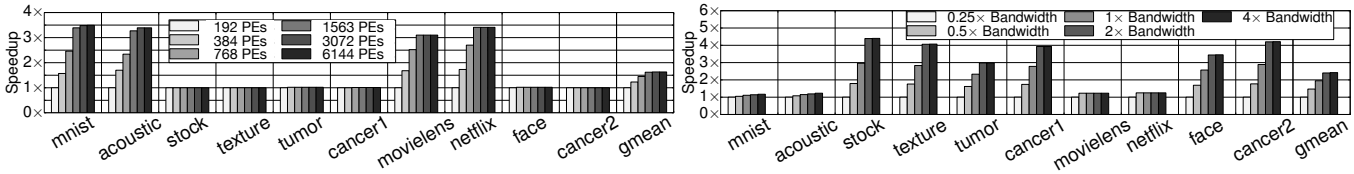
dominant part of their computation is relatively large matrix-matrix multiplication that GPUs can compute very efficiently. P-ASIC-F offers the same number of PEs and bandwidth compared to the FPGA but at higher frequency. These results show that just improvement in frequency does not translate to proportional speedup as long as the bandwidth remains unchanged. These results also show that the coalescence of CoSMIC's Planner, Compiler, and multi-threaded accelerator design has been effective in exploiting the FPGA resources. Across all benchmarks, P-ASIC-G shows significantly higher improvement as this design point combines more PEs, higher frequency, and higher bandwidth. The PE count and bandwidth of P-ASIC-G matches the GPU and its frequency is higher than the FPGA. However, as Figure 9 illustrates, even in the case of P-ASIC-G, the computation speedup does not translate to proportional system-wide improvement. These results confirm the importance of system software and CoSMIC-like full-stack approaches, as accelerators gain popularity.

The speedup of 3-GPU-CoSMIC comes from the GPU's higher frequency as well as massive parallelism; however, it also comes at an expense of higher power dissipation. Figure 11 highlights this power-performance tradeoff by depicting the improvement in Performance-per-Watt when comparing the FPGA- and P-ASIC-accelerated systems to the GPU-based system. The 3-FPGA-CoSMIC, 3-PASIC-F-CoSMIC, and 3-PASIC-G-CoSMIC systems achieve on average 4.2×, 6.9×, and 8.2× higher Performance-per-Watt than 3-GPU-CoSMIC, respectively. These results show that when the power-efficiency is the main concern, FPGAs or P-ASICs will be more desirable acceleration platforms than GPUs although GPUs provide higher performance than FPGAs and one of the P-ASICs, namely P-ASIC-F. Moreover, although P-ASICs provide both higher performance and power-efficiency, they impose a significant design and manufacturing cost. CoSMIC's template approach reduces the design time and cost as it offers a way to generate accelerator code. However, the cost of manufacturing may tip the scale towards FPGAs as they also offer significant benefits in both performance and power efficiency.

Sensitivity to mini-batch size. We use 10,000 as the default mini-batch size as used in the machine learning literature [74–76]. However, the optimal mini-batch size depends on several variables such as model, datasets, and training iterations. Larger mini-batch size reduces the rate of aggregation, which reduces the inter-node communication, leading to higher performance. Figure 13 illustrates this effect by segregating the fraction of runtime spent in computation and communication as the number of mini-batch size increases from b=500 to b=100,000 in the three-node runtime. On average, the computation with the mini-batch size 500 takes 12% of runtime but this increases to 95% when the mini-batch size is 100,000. However, reducing the aggregation rate can adversely affect training convergence [74–78]. To study the effect of mini-batch size on Spark and CoSMIC, we sweep the mini-batch size from 500 to 100,000 for three-node system configuration. Figure 12(a) and Figure 12(b) present the result of this sweep. For both figures, the baseline is the three-node Spark when mini-batch size is 10,000, our default setting. Comparing Figure 12(a) and Figure 12(b) shows that 3-FPGA-CoSMIC is faster across all combinations of benchmarks and mini-batch sizes. On average, with the same mini-batch size of b=500, CoSMIC is 16.8× faster. When the mini-batch size increases to b=100,000, CoSMIC is 9.1× faster. As the mini-batch size increases, Spark's overheads diminish. Nevertheless, CoSMIC outperforms Spark.

Sources of speedup. Figure 14 teases apart the benefits of FPGA acceleration from the benefits of the specialized system software over the three-node Spark. On average, the three FPGAs provide 20.7× speedup and the specialized system software–which also includes the aggregation part of the computation–is 28.4× faster than Spark's system software. As we discuss below, six of the benchmarks are more sensitive to data transfer and thus gain more benefits from the specialized system software compared to the benefits from FPGA. These benchmarks specifically benefit from the system software's task assignment that utilizes CPUs for both networking and aggregation of partial results from other nodes, thereby avoiding extra data transfer to the FPGAs. Nonetheless, all benchmarks gain from both FPGAs acceleration and specializing the system software.

Sensitivity to FPGA resources and bandwidth. CoSMIC can reshape and customize the template to match the resources of the target FPGAs or P-ASICs. The two main resources that affect performance are the number of PEs and the off-chip memory bandwidth. However, the DFG of the learning algorithm determines which resource is dominant. To study the interplay of algorithms and resources, we use a performance estimation tool that is validated against the hardware. Figure 15(a) illustrates the performance changes when the number of PEs varies from 192 to 6144 for a CoSMIC accelerator. The benchmarks that use the backpropagation (mnist and acoustic) and collaborative filtering algorithms (movielens and netflix) algorithms



(a) Speedup of CoSMIC accelerator with increasing number of PEs (Baseline: CoSMIC accelerator with 192 PEs)

(b) Speedup of CoSMIC accelerator as off-chip bandwidth changes (Baseline: CoSMIC accelerator using 25% of UltraScale+ bandwidth)

Figure 15: Speedup comparison with varying number of PEs and memory bandwidth for CoSMIC accelerators.

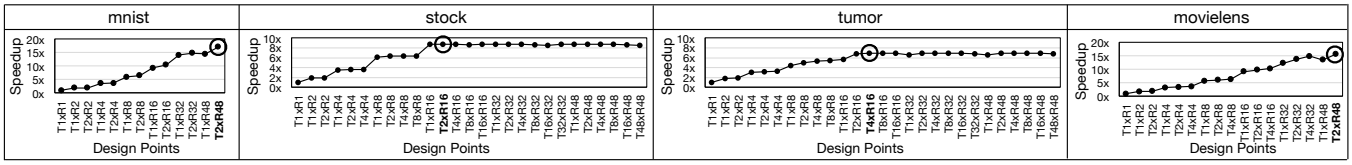


Figure 16: Design space exploration; T*x*×R*y*, *x* represents the number of threads and *y* represents the number of rows; baseline: T1×R1.

Table 3: Number of threads and FPGA resource utilization.

Name	# Threads per FPGA	LUTs (Total: 1,182,240)		Flip Flops (Total: 2,364,480)		BRAM (Bytes) (Total: 9720 KB)		DSP Slices (Total: 6840)	
		Used	Util	Used	Util	Used	Util	Used	Util
mnist	2	851,276	72.0%	772,029	32.7%	8,640	88.9%	4,070	59.5%
acoustic	2	851,276	72.0%	772,029	32.7%	8,128	83.6%	4,070	59.5%
stock	8	278,838	23.6%	249,907	10.6%	8,640	88.9%	1,320	19.3%
texture	1	283,535	24.0%	257,005	10.9%	8,640	88.9%	1,355	19.8%
tumor	4	281,522	23.8%	253,963	10.7%	8,640	88.9%	1,340	19.6%
cancer1	2	282,864	23.9%	255,991	10.8%	8,640	88.9%	1,350	19.7%
movielens	2	851,276	72.0%	772,029	32.7%	8,128	83.6%	4,070	59.5%
netflix	1	851,947	72.1%	773,043	32.7%	8,128	83.6%	4,075	59.6%
face	4	281,522	23.8%	253,963	10.7%	8,640	88.9%	1,340	19.6%
cancer2	2	282,864	23.9%	255,991	10.8%	8,640	88.9%	1,350	19.7%

show performance benefits as the number of PEs increases, since they are compute-bound. The rest of the benchmarks–linear regression, logistic regression, and support vector machines do not see any performance gains when the number of DSPs increases. Although these benchmarks are offered more PEs, the limited bandwidth curtails their performance. Figure 15(b), which sweeps bandwidth, suggests the same categorization (bandwidth-bound vs. compute-bound) for our algorithms. These results show that a single fixed design is not the most optimal for all the algorithms. Therefore, there is a need for template architectures and solutions, such as CoSMIC, that customize the accelerator design according to the algorithm. These results also suggest that modern accelerators need to strike a balance on allocating resource to off-chip communication and on-chip computation to maximize benefits for all benchmarks. Nonetheless, CoSMIC finds an optimal accelerator design considering both compute and bandwidth resources available on the FPGA.

Design space exploration. The Planner determines the number of PEs per thread and the number of threads in the accelerator. The Planner allocates PEs to each thread at the granularity of one row. This allocation strategy limits the design space that the Planner explores to find the optimal number of threads and rows-per-thread. In the case of UltraScale+ VU9P FPGA, the maximum number of possible design points is 27. Also, recall that the number of threads is also limited by the size of the model and not all the design points are possible. Figure 16 illustrates the result of this design space exploration for four different benchmarks. The performance of each design point is normalized to the design point which runs 1 thread using 1 row (T1xR1) of PEs. We sweep the number of rows from 1 to 48, which is the maximum number of rows in UltraScale+ while the maximum number of threads varies for every benchmark. The optimal design points are highlighted with a concentric circle in the graphs. Benchmarks mnist and movielens see the highest speedup when they use all the 48 rows since they are compute-bound. In contrast, the performance for stock and tumor saturates beyond 16 rows. This result is commensurate with Figure 15(a), which shows that mnist and movielens benefit significantly with an increase in the FPGA's computational resources (PEs), while stock and tumor do not. The rest of the benchmarks show trends similar to the ones in Figure 16. Further, the figure shows that for a fixed number of PE rows, increasing the number of threads improves performance, which confirms the importance of multi-threading. Table 3 shows the resource utilization and the optimal number of threads-per-FPGA for all the benchmarks corresponding to the optimal design point chosen by the Planner. The resource utilization is highest for benchmarks that are compute-bound and lowest for the benchmarks that are bandwidth-bound. Moreover, the results show the benefits of our template-based approach that enables optimal utilization of the limited resources in the FPGA's reconfigurable fabric.

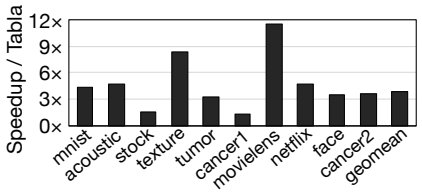


Figure 17: Speedup of CoSMIC's template architecture over Tabla's.

Comparison with Tabla. Prior work in TABLA [12] has explored single-node acceleration using a low-power FPGA (Zynq ZC702 with 220 DSPs). Our work, on the other hand, explores scale-out acceleration using modern high-power FPGAs (UltraScale+ with 6,840 DSPs). To provide a head-to-head comparison, we use the open-source TABLA framework [79] to generate accelerators for UltraScale+. We modify the templates for UltraScale+ and perform design space exploration to present the best results with TABLA. Figure 17 shows the speedup of CoSMIC compared to TABLA on UltraScale+ when using the same number of PEs. On average, CoSMIC performs 3.9× faster than TABLA. While both CoSMIC and TABLA use the same number of FPGA compute resources, the gap in performance shows that CoSMIC uses the compute resources more efficiently. The bottleneck for performance in TABLA is the communication of intermediate results due to data dependencies. As the number of DSPs in the TABLA architecture grows, the communication overhead

grows significantly. To reduce the communication overhead, CoSMIC architecture uses a scalable tree-bus across rows of our 2-D PE architecture, and a bidirectional link between columns of PEs. Moreover, TABLA's compiler does not consider the overhead of data communication, which is particularly important when the number of PEs is large. CoSMIC compiler (Section 6) maps the operations of the learning algorithm according to the location of data in order to reduce communication overhead. The combination of CoSMIC's scalable architecture, along with compiler optimization ensures that the FPGA's computational resources are used effectively.

8 RELATED WORK

Multi-node accelerators for machine learning. DaDianNao [4] provides a multi-chip ASIC accelerator for DNNs. Other works use multiple FPGAs for accelerating one specific task [80–82]. Farabet et al. [80] and Donninger et al. [81] use multiple FPGAs to accelerate DNNs [80] and a chess game [81], respectively. Walters et al. [82] propose a multi-FPGA accelerator for the Hidden Markov Models [82]. Putnam et al. [14] provide an FPGA fabric for accelerating Bing's ranking algorithms [14]. Microsoft [15] also provides an infrastructure for deploying FPGAs in datacenters, which is also used for the inference phase of DNNs. This release does not deal with training nor does it offer a framework for programming. CoSMIC provides the necessary framework to utilize and program such an infrastructure [15] for machine learning algorithms without involving programmers in hardware design. Recently, Microsoft also unveiled Brainwave [83] that uses multiple FPGAs for DNN inference. In contrast, CoSMIC is a full stack to accelerate training at scale. Google's TPU [84] is a systolic array for acceleration of matrix multiplication, which is prevalent operation in ML. TPU is also programmable from Tensorflow [85] that recently supports distributed execution. In contrast, CoSMIC enables the use of FPGAs for scale-out acceleration and comes with its own template architecture.

Template-based acceleration. TABLA [12] is a single-node accelerator generator for machine learning, which also uses a template-based architecture. As discussed in Section 7, TABLA, developed for a low-power FPGA (Zynq), does not effectively utilize the resources of a modern server-scale FPGA (UltraScale+). Furthermore, TABLA generates single-node FPGA accelerators which are inherently limited by the fine-grained parallelism available in the single-thread of stochastic gradient descent. In contrast, this paper framework not only generates *scalable* accelerators for distributed systems using a novel *multi-threaded* template architecture, but also provides the necessary system software stack for scale-out acceleration. Moreover, the compilation algorithm of this work differs from TABLA. Our algorithm reduces the data communication by mapping data first. In contrast, TABLA's algorithm maps operations first to reduce the single-threaded latency. Additionally, our algorithm optimizes the mapping of operation to the FPGA's resources according to the location of data to avoid data marshaling. DNNWEAVER [11] is another template-based accelerator generator that only generates accelerators for prediction with Deep Neural Networks (DNNs). DNNWEAVER does not deal with training, multiple FPGAs, or algorithms besides DNNs. Cheng, et al. [86] propose predesigned data flow templates as the intermediate point for HLS from general C/C++ workloads. LINQits [87] provides a template architecture for accelerating database queries. The last two works [86, 87] do not focus on learning algorithms nor do they deal with scale-out systems.

Single-node accelerators for machine learning. There is a large body of work on single-node accelerator design for ML [3, 5–10, 28–40, 40, 41, 41–49]. These works mostly focus on accelerating one or a fixed number of learning algorithms. CoSMIC, on the other hand, is a full stack that targets scale-out acceleration of learning.

HLS for FPGAs. Many related works (e.g., [49, 86, 89, 90]) explore HLS for FPGAs. HLS targets general applications while CoSMIC focuses specifically on machine learning. Therefore, HLS does not leverage any domain-specific knowledge or algorithmic insights. Using algorithmic commonalities for a range of machine learning algorithms is fundamental to our work and enables further benefits from hardware acceleration. Acceleration with HLS still requires hardware expertise. For instance, DNNWEAVER [11] reports that hardware design to optimize a Vivado HLS implementation of a deep neural network for FPGA took one month. The resulting implementation was an order of magnitude slower than a template-based accelerator for the same FPGA. A more recent work [86] uses dataflow templates as intermediate compilation target for C/C++ programs and delivers 9× higher performance than state-of-the-art HLS tools. CoSMIC takes a template-based approach that is driven by the theory of machine learning and targets distributed FPGA acceleration of training from a high-level domain-specific language.

System software for distributed FPGA acceleration. Another inspiring work [91] provides the mechanisms to integrate predesigned FPGA accelerator with Spark [1]. Melia [92] uses Altera's OpenCL-based HLS to offer a MapReduce-based framework for utilizing FPGAs in distributed systems. Another work [93] provides the framework for using Xilinx Vivado HLS tool for MapReduce [94] applications. CoSMIC does not rely on pre-developed FPGA accelerators or HLS for distributed FPGA acceleration, or generic system software.

9 CONCLUSION

While accelerators gain traction, their integration in the system stack is not well understood. CoSMIC takes an initial step toward such an integration for an important class of applications while providing generality and a high-level programming interface. The evaluations confirm that a full-stack approach is necessary and just designing efficient accelerators does not yield proportional benefits without a co-design of the entire system stack. The traditional approaches of profiling and offloading hot-regions of code lack the flexibility to support ever-changing algorithms and the emerging scale and heterogeneity in the systems. It is clear that a full-stack design is non-trivial but deeply understanding algorithmic properties of the application domain can significantly facilitate such approaches. CoSMIC takes advantage of the algorithmic understanding to simplify the layers of its stack by specializing them and offers a cohesive hardware-software solution. The encouraging results show that this paradigm is effective but the multifaceted nature of the cross-stack approach promises an exciting yet challenging road ahead.

10 ACKNOWLEDGMENTS

We thank Ali Jalali for insightful discussions and Amir Yazdanbakhsh for help with the synthesis results. We also thank Sridhar Krishnamurthy and Xilinx for donating the FPGA boards. This work was in part supported by NSF awards CNS#1703812, ECCS#1609823, CCF#1553192, Air Force Office of Scientific Research (AFOSR), Young Investigator Program (YIP) award #FA9550-17-1-0274, and gifts from Google, Microsoft, and Qualcomm.

REFERENCES

[1] Apache Spark, 2017. URL https://spark.apache.org/.

[2] Apache Hadoop, 2017. URL http://hadoop.apache.org/.

[3] 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.

[4] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. DaDianNao: A machine-learning supercomputer. In *MICRO*, 2014.

[5] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. ShiDianNao: shifting vision processing closer to the sensor. In *ISCA*, 2015.

[6] 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.

[7] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, Jose Miguel Hernandez-Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *ISCA*, 2016.

[8] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. EIE: efficient inference engine on compressed deep neural network. *CoRR*, abs/1602.01528, 2016. URL http://arxiv.org/abs/1602.01528.

[9] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *ISCA*, 2016.

[10] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *ISCA*, 2016.

[11] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Misra, and Hadi Esmaeilzadeh. From high-level deep neural models to fpgas. In *MICRO*, October 2016.

[12] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. TABLA: A unified template-based framework for accelerating statistical machine learning. In *HPCA*, 2016.

[13] Snickerdoodle: Affordable FPGA platform for powering everything robots, drones, and computer vision, 2017. URL http://krtkl.com/.

[14] 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.

[15] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. In *MICRO*, 2016.

[16] Amazon EC2 F1 instances: Run custom FPGAs in the AWS cloud, 2017. URL https://aws.amazon.com/ec2/instance-types/f1/.

[17] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. Towards a unified architecture for in-rdbms analytics. In *SIGMOD*.

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

[19] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. On parallelizability of stochastic gradient descent for speech dnns. In *ICASSP*, 2014.

[20] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In *NIPS*, 2010.

[21] Ofer Dekel, Ran Gilad-Bachrach, Ohad Shamir, and Lin Xiao. Optimal distributed online prediction using mini-batches. *Journal of Machine Learning Research*, 13 (Jan):165–202, 2012.

[22] J. Langford, A.J. Smola, and M. Zinkevich. Slow learners are fast. In *NIPS*, 2009.

[23] Gideon Mann, Ryan McDonald, Mehryar Mohri, Nathan Silberman, and Daniel D. Walker. Efficient large-scale distributed training of conditional maximum entropy models. In *NIPS*, 2009.

[24] Dipankar Das, Sasikanth Avancha, Dheevatsa Mudigere, Karthikeyan Vaidynathan, Srinivas Sridharan, Dhiraj Kalamkar, Bharat Kaul, and Pradeep Dubey. Distributed deep learning using synchronous stochastic gradient descent. *arXiv:1602.06709 [cs]*, 2016.

[25] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous SGD. In *ICLR Workshop Track*, 2016.

[26] Intel Altera. Arria 10 architecture, 2017. URL https://www.altera.com/products/fpga/arria-serqies/arria-10/features.html.

[27] Spark MLlib: Apache spark's scalable machine learning library. URL http://spark.apache.org/mllib/.

[28] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *MICRO*, 2016.

[29] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M. Aamodt, and Andreas Moshovos. Stripes: Bit-serial deep neural network computing. In *MICRO*, 2016.

[30] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-x: An accelerator for sparse neural networks. In *MICRO*, 2016.

[31] Yu Ji, Youhui Zhang, Shuangchen Li, and Ping Chi. NEUTRAMS: Neural network transformation and co-design under neuromorphic hardware constraints. In *MICRO*, 2016.

[32] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer cnn accelerators. In *MICRO*, 2016.

[33] Ioannis Stamoulias and Elias S. Manolakos. Parallel architectures for the knn classifier – design of soft ip cores and fpga implementations. *ACM Transactions on Embedded Computer Systems*, 13(2):22:1–22:21, September 2013.

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

[35] 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.

[36] E.S. Manolakos and I. Stamoulias. Ip-cores design for the knn classifier. In *ISCAS*, 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*, 2011.

[38] Tsutomu Maruyama. Real-time k-means clustering for color images on reconfigurable hardware. In *ICPR*, 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*, 2003.

[40] M. Papadonikolakis and C. Bouganis. A heterogeneous fpga architecture for support vector machine training. In *FCCM*, 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*, 2009.

[42] A. Majumdar, S. Cadambi, and S.T. Chakradhar. An energy-efficient heterogeneous system for embedded learning and classification. *IEEE Embedded Systems Letters*, 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 Transactions on Architecture and Code Optimization*, 9(1):6:1–6:30, Marcg 2012.

[44] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun. Neuflow: A runtime reconfigurable dataflow processor for vision. In *CVPRW*, 2011.

[45] Antonio Roldao and George A. Constantinides. A high throughput fpga-based floating point conjugate gradient implementation for dense matrices. *ACM Transactions on Reconfigurable Technology System*, 3(1), January 2010.

[46] 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*, 2006.

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

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

[49] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *FPGA*, 2015.

[50] 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:1408.5093*, 2014.

[51] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embed. Comput. Syst.*, 13(4s):134:1–134:25, April 2014.

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

[53] Yann LeCun and Corinna Cortes. MNIST handwritten digit database, 2010. URL http://yann.lecun.com/exdb/mnist/.

[54] A variant of mnist dataset with 8 millions records. URL https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass.html#mnist8m.

[55] Joel Praveen Pinto. *Multilayer Perceptron Based Hierarchical Acoustic Modeling for Automatic Speech Recognition*. PhD thesis, EPFL, 2010.

[56] Bin Zhou. High-frequency data and volatility in foreign-exchange rates. *Journal of Business & Economic Statistics*, 14(1), 2008.

[57] S Dhanya and Roshni VS Kumari. Comparison of various texture classification methods using multiresolution analysis and linear regression modelling. *Springerplus*, 5(54), 2016.

[58] MR Segal, KD Dahlquist, and BR Conklin. Regression approaches for microarray data analysis. *Journal of Computational Biology*, 10(6), 2003.

[59] D Singh, P Febbo, K Ross, D Jackson, J Manola, C Ladd, P Tamayo, A Renshaw, Amico A D, J Richie, E Lander, M Loda, P Kantoff, T Golub, and W Sellers. Gene expression correlates of clinical prostate cancer behavior. *Cancer Cell*, 1(2), 2002.

[60] Iván Cantador, Peter Brusilovsky, and Tsvi Kuflik. Movielens dataset. In *HetRec*, 2011.

[61] Grouplens. Movielens dataset, 2017. URL http://grouplens.org/datasets/movielens/.

[62] Netflix Prize Data Set. URL http://www.netflixprize.com/.

[63] J. Weston, S. Mukherjee, O. Chapelle, M. Pontil, T. Poggiott, and V. Vapnik. Feature selection for svms. In *NIPS*, 2000.

[64] Spark gpu and simd support. URL https://github.com/kiszk/spark-gpu.

[65] Rajesh Bordawekar. Accelerating spark workloads using gpus. URL https://www.oreilly.com/learning/accelerating-spark-workloads-using-gpus.

[66] GPUEnabler. URL https://github.com/ibmsparkgpu/gpuenabler.

[67] CUDA-MLlib. URL https://github.com/ibmsparkgpu/cuda-mllib.

[68] 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.

[69] Caffe2, 2017. URL https://github.com/caffe2/caffe2.

[70] CUDA v8.0, 2017. URL https://developer.nvidia.com/cuda-toolkit.

[71] cuDNN v7.0, 2017. URL https://developer.nvidia.com/cudnn.

[72] Wattsup .net meter., 2017. URL http://www.wattsupmeters.com/.

[73] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *ASPLOS*, 2011.

[74] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J. Smola. Efficient mini-batch training for stochastic optimization. In *KDD*, 2014.

[75] Andrew Cotter, Ohad Shamir, Nathan Srebro, and Karthik Sridharan. Better mini-batch algorithms via accelerated gradient methods. In *NIPS*, 2011.

[76] Martin Takáč, Avleen Bijral, Peter Richtárik, and Nathan Srebro. Mini-batch primal and dual methods for svms. In *ICML*, 2013.

[77] Ofer Dekel, Ohad Shamir, and Lin Xiao. Optimal distributed online prediction using mini-batches. *Journal of Machine Learning Research*, 13(1):165–202, 2012.

[78] Richard H. Byrd, Gillian M. Chin, Jorge Nocedal, and Yuchen Wu. Sample size selection in optimization methods for machine learning. *Mathematical Programming*, 134(1), 2012.

[79] TABLA source code. URL http://www.act-lab.org/artifacts/tabla/.

[80] Clément Farabet, Yann LeCun, Koray Kavukcuoglu, Eugenio Culurciello, Berin Martini, Polina Akselrod, and Selcuk Talay. Large-scale fpga-based convolutional networks. *Machine Learning on Very Large Data Sets*, 2011.

[81] Chrilly Donninger, Alex Kure, and Ulf Lorenz. Parallel brutus: The first distributed, fpga accelerated chess program. In *IPDPS*, 2004.

[82] John Paul Walters, Xiandong Meng, Vipin Chaudhary, Tim Oliver, Leow Yuan Yeow, Darran Nathan, Bertil Schmidt, and Joseph Landman. Mpi-hmmer-boost: Distributed fpga acceleration. *Journal of VLSI Signal Processing*, 48(3):223–238, 2007.

[83] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengil, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Christian Boehn, Oren Firestein, Alessandro Forin, Kang Su Gatlin, Mahdi Ghandi, Stephen Heil, Kyle Holohan, Tamas Juhasz, Ratna Kumar Kovvuri, Sitaram Lanka, Friedel van Megen, Dima Mukhortov, Prerak Patel, Steve Reinhardt, Adam Sapek, Raja Seera, Balaji Sridharan, Lisa Woods, Phillip Yi-Xiao, Ritchie Zhao, and Doug Burger. Accelerating persistent neural networks at datacenter scale. In *HotChips*, 2017.

[84] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *ISCA*, 2017.

[85] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. *arXiv:1603.04467 [cs]*, 2016.

[86] Shaoyi Cheng and John Wawrzynek. High Level Synthesis with a Dataflow Architectural Template. In *OLAF*, June 2016.

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

[88] 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.

[89] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[90] I. Ouaiss, S. Govindarajan, V. Srinivasan, and R. Vemuri. An integrated partitioning and synthesis system for dynamically reconfigurable multi-fpga architectures. *Lecture Notes in Computer Science*, 1385-1388:31–36, 1999.

[91] Muhuan Huang, Di Wu, Cody Hao Yu, Zhenman Fang, Matteo Interlandi, Tyson Condie, and Jason Cong. Programming and runtime support to blaze fpga accelerator deployment at datacenter scale. In *SoCC*, 2016.

[92] Zeke Wang, Shuhao Zhang, Bingsheng He, and Wei Zhang. Melia: A mapreduce framework on opencl-based fpgas. *IEEE Transactions on Parallel and Distributed Systems*, 27(12):3547–3560, 2016.

[93] Dionysios Diamantopoulos and Christoforos Kachris. High-level synthesizable dataflow mapreduce accelerator for fpga-coupled data centers. In *SAMOS*, 2015.

[94] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*, 2004.