OPTIMO: A 65-nm 279-GOPS/W 16-b Programmable Spatial-Array Processor with On-Chip Network for Solving Distributed Optimizations via the Alternating Direction Method of Multipliers

Muya Chang, *Student Member, IEEE*, Li-Hsiang Lin, Justin Romberg, *Fellow, IEEE*, and Arijit Raychowdhury, *Senior Member, IEEE*

***Abstract*— This article presents OPTIMO, a 65-nm, 16-b, fully programmable, spatial-array processor with 49 cores and a hierarchical multi-cast network for solving distributed optimizations via the alternating direction method of multipliers (ADMM). ADMM is a projection-based method for solving generic-constrained optimizations' problems. In essence, it relies upon decomposing the decision vector into subvectors, updating sequentially by minimizing an augmented Lagrangian function, and eventually updating the Lagrange multiplier. The ADMM algorithm has typically been used for solving problems in which the decision variable is decomposed into two or multiple subvectors. We demonstrate six template algorithms and their applications and measure a peak energy efficiency of 279 GOPS/W.**

***Index Terms*— Alternating direction method of multipliers (ADMM), array processing, distributed, multi-cast network, near-memory computing, optimizations.**

I. Introduction

THE EXPLOSION of big-data problems arising in statistics, machine learning (ML), image processing, 5G systems, and other related areas [1] has accelerated the development of hardware prototypes that rely on data-flow architectures and near-memory processing to address the memory bottleneck. As computational models that rely on a close coupling between data storage and computation become relevant, the importance of specialized hardware architectures that can provide breakthrough advances in energy efficiency and performance is also increasing. The current generation of such hardware is mostly geared toward inference in neural networks (NNs). However, looking beyond the success of NN accelerators for classification [2]–[5], we recognize a growing need for solving complex optimization problems, which arise in all areas of signal processing [6]–[9], such as ML model training, computational imaging (medical, optical, and hyperspectral) [10], resource allocation in 5G massive multiple-input and multiple-output (MIMO) networks [11], and solving inverse problems, such as low-density parity-check (LDPC) decoding [12]. Currently, most of these algorithms are solved in GPUs and CPUs; however, with the widespread proliferation of ML, embedded signal processing, computational imaging, and so on, there is a growing demand for solving such optimizations both at edge nodes as well as the cloud.

In spite of the diversity of applications, a common mathematical framework, namely solving constrained optimizations (i.e., minimize $l(x)$ under a constraint $r(x) = 0$ for a vector x and functions l and r) binds most optimization problems. A particularly challenging task in solving optimization problems is the dimensionality of the task, namely, the size of the data set or the feature set. This requires innovative algorithms as well as hardware-algorithm codesign. Of increasing importance are distributed optimization algorithms where different processing elements can work on different parts of the data or features, then communicate their local solutions with each other too, and finally, reach a global consensus. These algorithms have been discussed in the literature [1], [13]–[15]. Among these algorithms, alternating direction method of multipliers (ADMM) has been particularly successful for solving the constraint optimization problems for large-scale data sets [1]. In particular, ADMM provides excellent convergence for distributed data. In addition, it has been shown in [16] that quantization error does not lead to unbounded error for large problem classes. This allows us to use fixed-point arithmetic for solving ADMM on large data. A review of the ADMM algorithm is provided in Section II, but it suffices to say that ADMM is one of the most common iterative algorithms for solving a large class of optimization

Manuscript received June 26, 2019; revised September 8, 2019 and October 29, 2019; accepted November 2, 2019. Date of publication December 12, 2019; date of current version February 25, 2020. This article was approved by Guest Editor Qun Jane Gu. This work was supported in part by the Semiconductor Research Corporation through JUMP CBRIC Task under Grant 2777.006, in part by the National Science Foundation under Grant 1640081, and in part by Nanoelectronics Research Corporation (NERC), a wholly owned subsidiary of the Semiconductor Research Corporation (SRC), through Extremely Energy Efficient Collective Electronics (EXCEL), an SRC-NRI Nanoelectronics Research Initiative, under Grant 2698.001 and Grant 2698.002. *(Corresponding author: Muya Chang.)*

The authors are with the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, 30332 USA (e-mail: mchang87@gatech.edu; llin79@gatech.edu; jrom@ece.gatech.edu; arijit.raychowdhury@ece.gatech.edu).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JSSC.2019.2953831

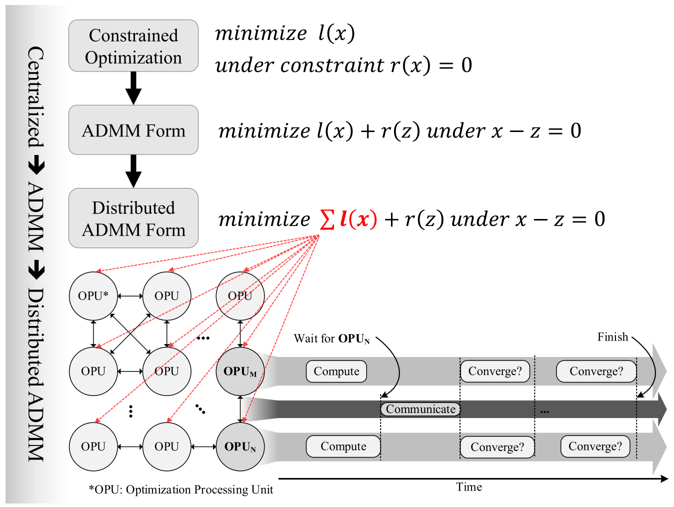

Centralized → ADMM → Distributed ADMM
Constrained Optimization
minimize $l(x)$ under constraint $r(x) = 0$
ADMM Form
minimize $l(x) + r(z)$ under $x - z = 0$
Distributed ADMM Form
minimize $\sum l(x) + r(z)$ under $x - z = 0$
OPU*
OPU
OPU$_M$
OPU$_N$
Wait for OPU$_N$
Finish
Compute
Converge?
Communicate
Time
*OPU: Optimization Processing Unit

Fig. 1. OPTIMO: A spatial-array processor for solving distributed optimizations via distributed ADMM.

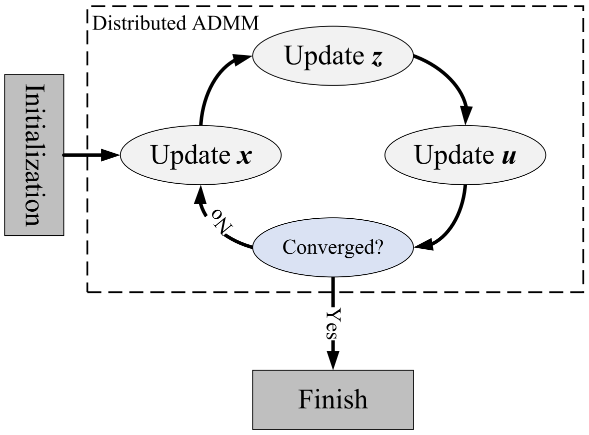


Fig. 2. Program flow for distributed ADMM.

problems and interested readers are pointed to [1] for a detailed survey. From a hardware perspective, ADMM, particularly in its distributed form is a universal and powerful computing model as it relies on local, iterative computing on a subset of the data (local memory) along with periodic exchange of information with near/distant neighbors (a programmable or reconfigurable data flow) to converge to a global solution (called consensus), as shown in Fig. 1. Also, Fig. 2 briefly shows how the program is initialized, executed, and terminated. Here x, z, and u are intermediate variables, which will be introduced in Section II, but the information flow is captured in this diagram.

In this article, we present OPTIMO, a spatial-array processor with near-memory processing, a hierarchical and multi-cast on-chip network, and full-programming support for solving distributed optimizations via ADMM. The motivation for OPTIMO is shown in Fig. 1. We demonstrate six template algorithms: 1) least-squares optimizations [17]; 2) least absolute shrinkage and selection operator (LASSO) [18]; 3) elastic net [19]; 4) linear support vector machine (SVM) [20]; 5) group-LASSO [21]; and 6) distributed averaging [22]. The algorithms use different objectives and constraints and represent a vast majority of statistical algorithms that are used on big data sets. To understand the importance

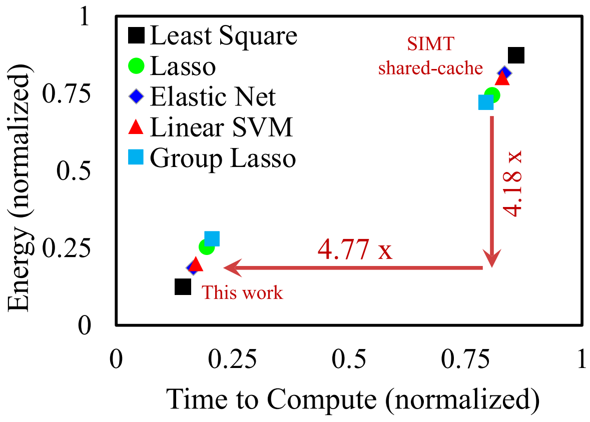


Fig. 3. This article (measured) shows 4.77× (4.18×) improvement in energy (performance) compared to a GPU-style SIMT machine (simulated).

of the data-flow architecture for solving such optimizations, we simulated a GPU style SIMT machine (with both local and shared cache) and compared it with OPTIMO, for iso-number of cores and clock frequency. From simulations (see Fig. 3), we note a 4.77× (4.18×) improvement in energy (performance). To the best of our knowledge, this is the first at-scale demonstration of a programmable array processor for solving a set of optimization problems that are used for various emerging applications, such as training of ML models, computational imaging, and signal processing.

II. Overview of the Algorithm

In this section, we provide an overview of ADMM as well as its distributed representation, namely distributed ADMM.

A. ADMM

The ADMM algorithm [1] is a projection-based method for solving generic problems of constrained optimizations. In essence, it relies upon decomposing the decision vector into subvectors, updating each subvector sequentially by minimizing an augmented Lagrangian function, and finally updating the Lagrange multiplier corresponding to the constraint that couples the subvectors using a dual subgradient method. The ADMM algorithm has typically been used for solving problems in which the decision variable is decomposed into two or multiple subvectors. For simplicity, we only review the form of ADMM with two subvectors, and its generalization to the case of multiple subvectors is straightforward and is omitted here.

The original form of ADMM with two subvectors denoted as $x \in \mathbf{R}^n$ and $z \in \mathbf{R}^m$ solves the problem expressed as

$$\begin{aligned} &\min \ l(x) + r(z) \\ &\text{s.t. } Ax + Bz = c \end{aligned} \tag{1}$$

where $A \in \mathbf{R}^{p\times n}$, $B \in \mathbf{R}^{p\times m}$, and $c \in \mathbf{R}^p$. We assume that both $l(x)$ and $r(z)$ are convex.

We solve (1) using ADMM by first deriving the augmented Lagrangian function of (1), and it is given by

$$\begin{aligned} L_\rho(x, z, y) = l(x) + r(z) + y^T(Ax + Bz - c) \\ +(\rho/2)||Ax + Bz - c||_2^2 \end{aligned} \tag{2}$$

where y is the Lagrange multiplier corresponding to the constraint $Ax + Bz = c$ and ρ is a positive scalar. Then, we perform an iterative algorithm that starts from arbitrary initial values $x^{(0)}, z^{(0)}$, and $y^{(0)}$, and update using the following updated rules:

$$\begin{aligned} x^{(k+1)} &:= \underset{x}{\operatorname{argmin}}\ L_\rho(x, z^{(k)}, y^{(k)}) \\ z^{(k+1)} &:= \underset{z}{\operatorname{argmin}}\ L_\rho(x^{(k+1)}, z, y^{(k)}) \\ y^{(k+1)} &:= y^k + \rho(Ax^{(k+1)} + Bz^{(k+1)} - c). \end{aligned} \tag{3}$$

Equation (3) is solved iteratively for $k \geq 0$ until convergence is achieved.

B. Distributed ADMM

By splitting up an objective function carefully, one can transform ADMM to solve a range of useful optimization programs in a distributed fashion, and this gives rise to distributed ADMM. In its distributed form, one can parallelly solve a large optimization problem over a large data set or a large vector over multiple cores with intermittent communication between the cores to achieve consensus. This makes solving many problems in image processing, signal recovery, ML, model prediction, and classification efficient and real time. To provide an overview of distributed ADMM, we consider the following problem:

$$\min_x \ l(Ax - b) + r(x) \tag{4}$$

or its ADMM form

$$\begin{aligned} &\min_x \ l(x) + r(z) \\ &\text{s.t. } x = Az - b \end{aligned} \tag{5}$$

which is a transformed representation of the original ADMM problem (1). There are two ways to solve (4) and (5) in a distributed manner: one is splitting across the data (or training examples in case of model fitting), and the other is by splitting across feature vectors. We explain the two splitting methods and provide their related examples and applications in the following.

1) Splitting Across Data: In most classical statistical estimation and ML problems, the number of features is modest, but the number of training examples can be very large. Thus, we can utilize the structure of the problem by letting each processor core handle a subset of the training data. This is useful in many scenarios, such as online social network data processing, wireless sensor networks, and many cloud computing applications. We partition A and b by rows

$$A = \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_N \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{bmatrix}$$

where $A_i \in \mathbf{R}^{m_i \times n}$, $b_i \in \mathbf{R}^{m_i}$, and $\sum_{i=1}^N m_i = m$. Thus, A_i and b_i represent the ith partition of the data handled by the ith processor. Over the partitions, if function f in (4) is separable in the sense

$$f(Ax - b) = \sum_{i=1}^N l_i(A_i x - b_i)$$

the optimization problem (4) becomes

$$\min_x \ \sum_i^N l_i(A_i x - b_i) + r(x)$$

or in the ADMM form

$$\begin{aligned} &\min_{x_1, x_2, \ldots, x_N} \ \sum_i^N l_i(A_i x_i - b_i) + r(z) \\ &\text{s.t. } x_i = z \quad \text{for } i = 1, \ldots, N. \end{aligned} \tag{6}$$

Following the ADMM algorithm described in Section II-A, we can solve (6).

Equation (6) is a generalized version of many problem formulations and the applications are referred to as penalized empirical risk minimization and structural risk minimization in ML. For example, when $l_i(A_i x - b_i) = ||A_i x - b_i||_2^2$ and $r(x) = \lambda ||x||_1$, problem (6) becomes the well-known LASSO problem [18] in its distributed form.

2) Splitting Across Features: For another set of applications such as natural language processing (NLP) [23] and bioinformatics [24], there are often a modest number of examples but a large number of features. In such situation, we would partition A by columns x by rows

$$A = \begin{bmatrix} A_1 & A_2 & \ldots & A_N \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} \tag{7}$$

where $A_i \in \mathbf{R}^{m \times n_i}$, $x \in \mathbf{R}^{n_i}$, and $\sum_{i=1}^N n_i = n$. This implies that $Ax = \sum_{i=1}^N A_i x_i$, i.e., $A_i x_i$ can be thought of as a "partial" prediction of b using only the features referenced in x_i. With this partitioning and under the assumption (which in most practical applications is true) that $r(x)$ is separable such that $r(x) = \sum_{i=1}^N g_i(x_i)$, the problem (4) in its ADMM form becomes

$$\begin{aligned} &\min_{x_1, x_2, \ldots, x_N} \ \sum_i^N l(z_i - b) + \sum_{i=1}^N r_i(x_i) \\ &\text{s.t. } Ax_i = z_i \quad \text{for } i = 1, \ldots, N. \end{aligned} \tag{8}$$

More examples of problems which can be formulated in the form of (8) can be found in [1], and interested readers are pointed to [1] for further reading.

C. Distributed Optimization as a Template Problem

What has been described earlier is an overview of the distributed ADMM problem formulation, the details of how it can be mapped to specialized hardware will be described in Section IV. We have chosen six popular algorithms from signal/image processing and ML community, namely as least-square optimization [17], Lasso [18], Group Lasso [21],

Algorithms	$f(x)$	$g(z)$	**Applications**
Least Square	$(1/2)\|Ax-b\|_2^2$	-	Modeling for Prediction
Lasso	$(1/2)\|Ax-b\|_2^2$	$\lambda\|x\|_1$	Variable selections Modeling for prediction
Elastic Net	$(1/2)\|Ax-b\|_2^2$	$\lambda_1\|x\|_1+\lambda_2\|x\|_2^2$	Variable selections Robust modeling for prediction
Group Lasso	$(1/2)\|Ax-b\|_2^2$	$\lambda\sum_{i=1}^{N}\|x_i\|_2$	Structure variable selection Modeling for prediction
Linear SVM	$\|x_i\|^2$	$y_k(a_k^T x_i+b)$	Classification
Distributed Averaging	$(1/2)\|Ax-b\|_2^2$	-	Large scale modeling and prediction

Fig. 4. Table for loss functions and regularization functions.

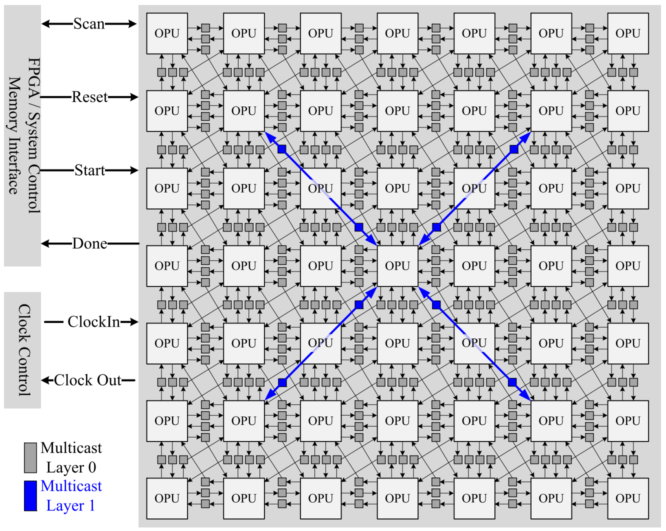


Fig. 5. System architecture showing the 49 OPUs and a two-layer multi-cast on-chip network.

elastic net [19], SVMs [20], and distributed averaging [22]. The table of loss functions and the regularization functions for each template problem are shown in Fig. 4.

One thing to keep in mind is that even though all the six algorithms follow the same program flow as in Fig. 2, how x and z are updated depends on the loss function and the regularization functions. This calls for hardware-level programmability which we describe next. Furthermore, the programming model ensures that a larger class of algorithms can be mapped to the hardware, and although we do not describe them in this article, the hardware architecture and the programming model provide a fundamental fabric for solving a very large class of distributed optimizations. Once a problem can be written in the form of (6) and (8), distributed ADMM can be efficiently mapped and executed on the proposed test chip, which we call OPTIMO. To introduce OPTIMO, we first present its architecture in Section III.

III. Overview of the System Architecture

A. System Architecture

Fig. 5 shows the chip architecture where 49 programmable 16-b optimization processing units (OPUs) are capable of: 1) computing locally and iteratively and 2) transmitting/receiving data from the neighbors. The chip boundary has

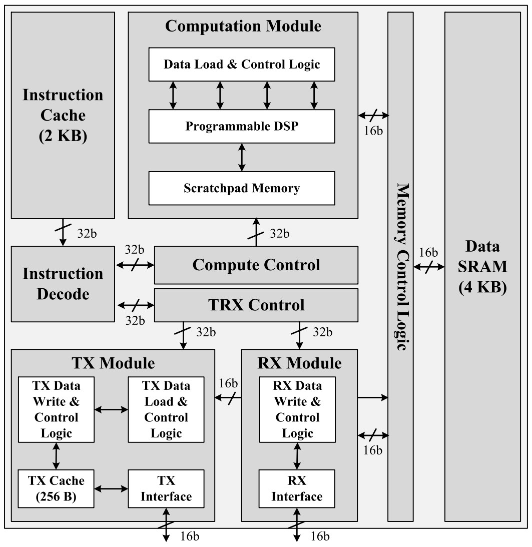


Fig. 6. Architecture of the OPU showing the principal modules.

communication interfaces to the PCB that contain: 1) scan ports; 2) system control ports; and 3) clock ports. High-level program code and data sets are translated to instruction and data, scanned into the chip through scan ports and then executed. The control ports are used to start/reset the system and the clock ports are used to either provide the clock externally and/or monitor the system clock frequency. Convergence is declared either after a fixed number of iterations or when the maximum cycle-to-cycle change of data in an OPU falls below a threshold. The system also contains two multicast layers, which will be described in Section III-D.

The choice of 16 b is driven by the data set and the applications. For signal and image processing applications, that are of interest to us, the raw data is 8 b and we have determined that 16-b precision yields the same results as floating point for the thousands of image and signal processing data sets that we have analyzed. Furthermore, ADMM is forgiving in terms of quantization error, and the system converges because of the iterative nature of the algorithm.

In Section IV, the detailed architecture of each OPU is described.

B. OPU

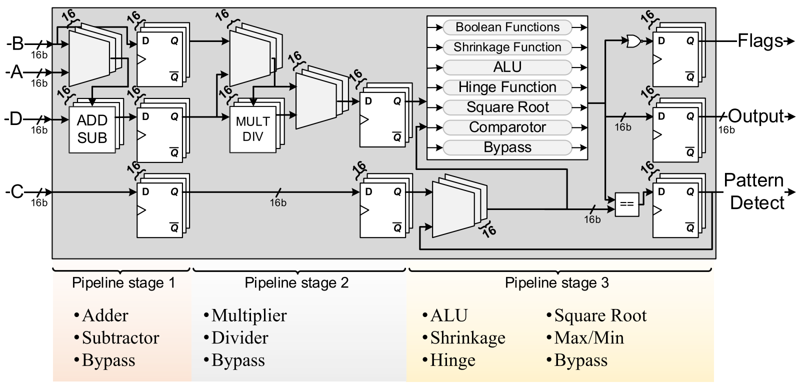
One of the important challenges for spatial-array architectures is scalability. In this design, we ensure that the IO pins for each OPU are placed in a symmetric fashion such that the OPUs can easily abut. Each of the OPU features is given as follows (see Fig. 6): 1) one computation module consisting of a programmable digital signal processor (P-DSP), a scratchpad memory, and control logic; 2) 2 kB of instruction cache; 3) 4 kB of data memory (for local data R/W); and 4) a transceiver module for the gather and scatter processes. Programming is supported via 32-b instructions, which will be described in detail in Section IV, and each inter-OPU data movement is supported on dedicated links. For this article, we mainly focused on how such architecture relates to iterative optimization problems; therefore, we assume that the weights and data can fit into each OPU. For more complex problems,

Macro Functions	No. of instr.	Algorithm Type					
		1*	2*	3*	4*	5*	6*
L_1 Norm	3		✓	✓			
$(L_2\ \text{Norm})^2$	3	✓	✓	✓	✓	✓	
L_2 Norm	6		✓				
L_{Inf} Norm	3		✓				
Shrinkage	3		✓	✓			
MAC	4	✓	✓	✓	✓	✓	✓
Hinge Loss	3				✓		
Distributed Averaging	10	✓	✓	✓	✓	✓	

Functional Blocks	No. of Instr.
Computation Controller	11
Communication Controller	8
Programmable DSP	768

1*. Least Square 4*. Elastic Net
2*. Lasso 5*. Linear SVM
3*. Group Lasso 6*. Distributed Averaging

(a)



(b)

Fig. 7. (a) Programming support is enabled via a custom ISA with a 32-b instruction format and macro functions. (b) P-DSP architecture showing a three-stage pipeline.

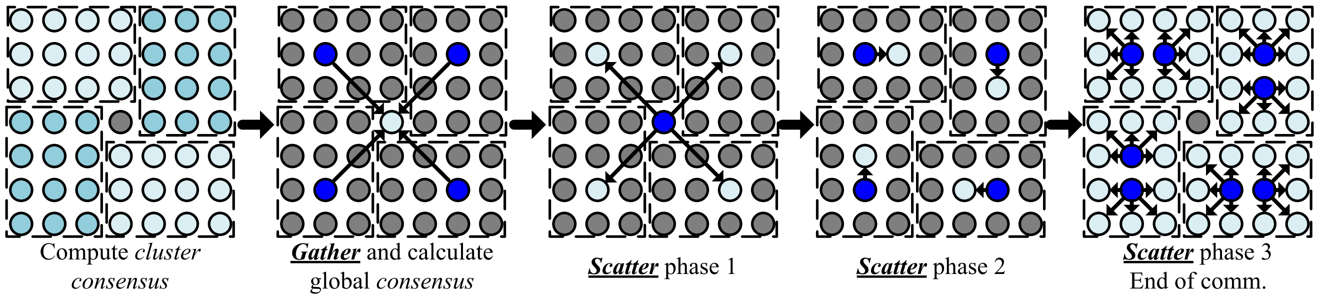

Compute *cluster consensus*
Gather and calculate global *consensus*
Scatter phase 1
Scatter phase 2
Scatter phase 3 End of comm.

Fig. 8. Gather and scatter processes of communication enabled by a hierarchical on-chip network result in fast convergence to a global consensus.

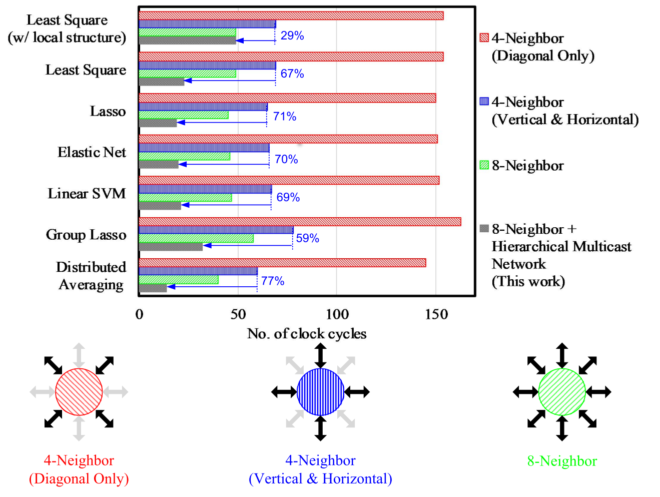


Fig. 9. Time for convergence for template algorithms as a function of their connectivity with their neighbors.

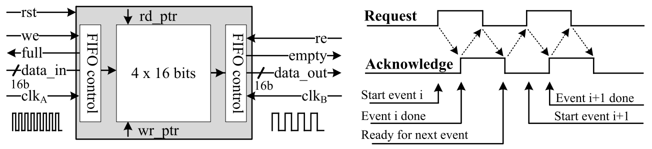


Fig. 10. FIFO architecture and the corresponding timing diagram.

the architecture can remain the same albeit with a more sophisticated network on chip (NoC) and higher bandwidth OPU to OPU bandwidth. Before data transmission, a transmit buffer temporarily stores the data and it is flushed out at the end of the transmission. Received data are not buffered; instead, the control logic directly writes the incoming traffic to the data cache, thereby reducing both latency and energy. The design supports synchronous, mesochronous, as well as asynchronous communication among OPUs with bidirectional first-in-first-outs (FIFOs) enabling fast and parallel data exchange across clock-crossing boundaries [25]. Fig. 7(a) further shows the number of instructions supported by each module, the commonly used macrofunctions and the number of instructions per function, and their usage in the six template algorithms.

C. P-DSP

Fig 7(b) shows the principal components of the P-DSP, which consists of three pipeline stages in an architecture designed to maximize energy efficiency. The first supports add/subtract (or bypass), the second stage supports multiply/divide (or bypass), and the final stage supports a class of fixed-function blocks, as shown in Fig. 7(b). The key fixed-function blocks are Boolean functions processors, shrinkage function unit, a 16-b ALU, a hinge function calculator, and a square root function evaluator. Instruction-level control of the pipeline and variable latency through the P-DSP is maintained via a program counter. The number of cycles required to execute an instruction will dynamically change depending on the type of instruction and the architecture configuration. The

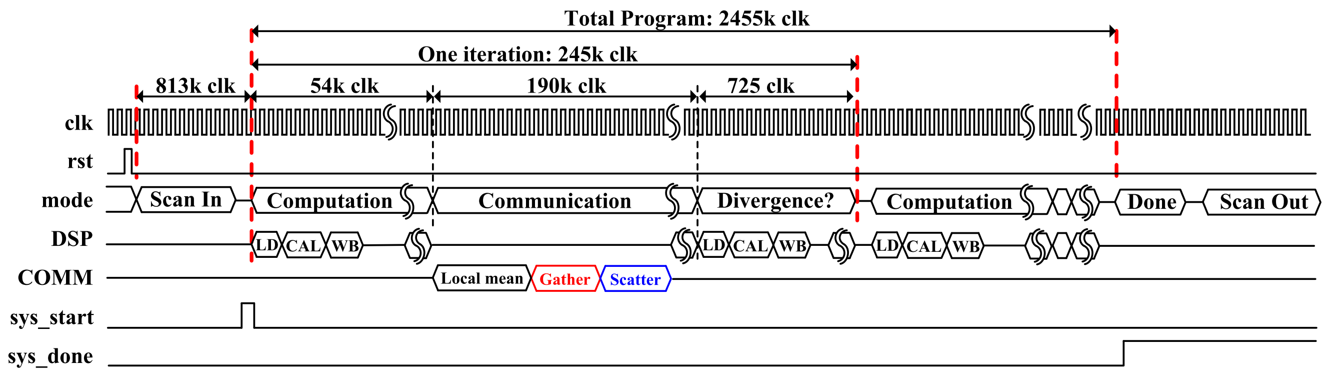


Fig. 11. Timing diagram of showing the various steps of a template optimization problem.

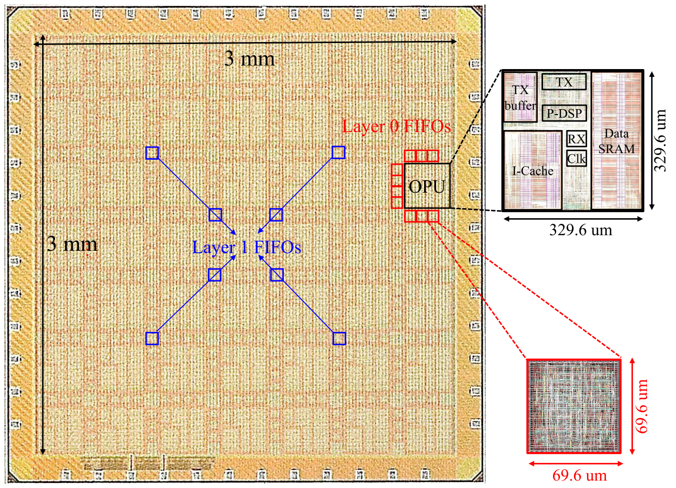


Fig. 12. Die micrograph.

Technology	TSMC 65nm GP 1P9M
Chip Size	3.41 mm x 3.41 mm
Core Area	3 mm x 3 mm
Package	QFN6x6-48
Pin Count	48
Gate Count (logic only)	2725 kGates (NAND2)
On-Chip SRAM	306.25 KB
Number of OPUs	49
No. of pipeline stages in programmable DSP	3
Core Supply Voltage	0.5-1.2 V
IO Supply Voltage	2.5 V
Clock Rate	10-270 MHz
Network	Asynchronous & Mesochronous
Peak Energy Efficiency	279 GOPS/W
Arithmetic Precision	16-bit fixed-point

Fig. 13. Chip characteristics.

detail of how to program the P-DSP block as well as the related instructions is described in Section IV.

D. On-Chip Network Design

The OPUs indexed as (row and column) interact via a two-layered multi-cast network with: 1) layer-0 establishing near-neighbor (neighborhood of 8) bi-directional connections and () layer-1 connecting four cluster center OPUs, i.e., (2, 2), (6, 2), (2, 6), and (6, 6) with the chip-center OPU, i.e., (4,4). Depending on the algorithm and structure of the data, optimization algorithms require complex data-flow patterns where both near-neighbor (layer-0 connections) as well as global information (layer-0 and layer-1 connections) are used. The 48 OPUs (excluding the chip center) are divided into four clusters as shown, with the OPU in the center as shown in Fig. 8. Global consensus is reached in each iteration via the following steps.

1) The four clusters reach cluster-level consensus (layer-0).
2) Gather process where the chip center obtains cluster-level consensus information from cluster centers (layer-1) and calculates the global data.
3) Scatter (step-1) process where the chip center scatters the global data back to the cluster centers.
4) Final scatter (steps 2 and 3) process where the cluster centers spread the data across all the OPUs

Once these processes terminate, the system will ready all the OPUs for the next iteration. The scatter and gather processes are intrinsic to distributed optimizations as the system computes locally, distributes information globally, and iterates to reach consensus. We compare the proposed hierarchical multi-cast network with networks that allow four or six connections to the neighbors-as is common in convolutional and deep NNs [2]. It is intuitive to understand that instead of connections to all the six neighbors, consensus data can also be transmitted by just connecting to the four near neighbors (as found in Google's TPU). However, this comes at a cost of increase number of iterations. Architectural and network simulations of various optimization algorithms on more than 10 000 random data sets reveal a 29%–77% reduction in convergence time compared to a fixed, four-neighbor connection (see Fig. 9).

E. Clocking

Clocking often becomes critical when scalability and power efficiency are considered. To overcome this issue, we implement two clocking options on the chip: 1) a single global clock (synchronous) either internal or external or 2) digital controlled oscillator (DCO)-based clock per-OPU enabling asynchronous/mesochorous links. The DCO-based local clocks have external control via scan for fine-tuning. The single global clock option acts as the baseline for us to compare with per-OPU-based clocking, where we show the comparison in Section V. The system runs at full capacity when all the OPUs are producing outputs at a fixed and equal rate—which requires synchronous communication. In such a scenario, no OPU has to wait for its neighbors to finish computation. However, per-OPU-based clocking removes design constraints

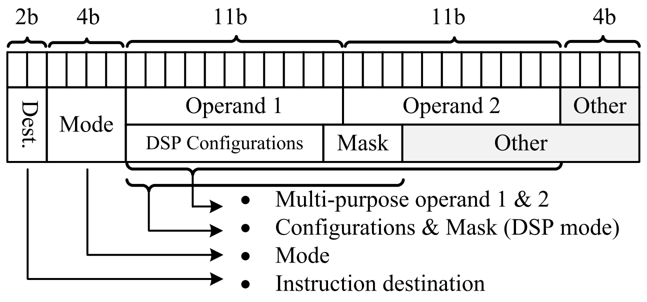


Fig. 14. 32-b Instruction format.

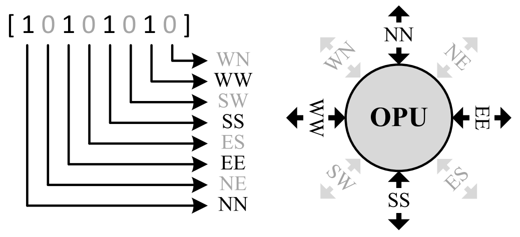


Fig. 15. Example of a “mask” configuration.

	Stage 1		**Stage 2**		**Stage 3**	
Number of cycles	Adder	1	Multiplier	18	Square Root	14
	Subtractor	1	Divider	27	Others	1
	Bypass	1	Bypass	1		

Fig. 16. Number of cycles required for each function.

on fine-grained control of clock skew. As a compromise, mesochronous clocks running at identical frequencies but mismatched phases maintain high throughput without requiring stringent skew requirements. To support mesochronous as well as asynchronous clocks per-OPU, the clock-crossing FIFO features a 64-b buffer and operates on a four-phase handshaking scheme, as shown in Fig. 10. The example timing diagram for executing ADMM is shown in Fig. 11.

F. Die Micrograph and Chip Characteristics

The test chip is fabricated in a TSMC 65-nm GP CMOS process and occupies a total area of 12mm^2, as shown in Fig. 12. It features 306.25 kB of on-die memory distributed across 49 cores. The chip characteristics are shown in Fig. 13.

IV. ISA AND PROGRAMMING

As mentioned in Section II, the complexity and details of the algorithmic steps to be followed to update x and z that depends on the combinations of the loss function and the regularization functions. For example, both Lasso and Group Lasso use L_1 norm as the penalty function; however, as previously shown in Fig. 7, the functions required for each algorithm are very different, therefore resulting in a very different sequence of instructions. To enable programmability, we develop a customized instruction set architecture (ISA) to support the possible set of arithmetic functions that are required. The instructions can be categorized by targets into four kinds as follows: 1) computation controller; 2) communication controller; 3) P-DSP; and 4) branch controller. As shown in Fig. 14, the instructions are 32 bits long and

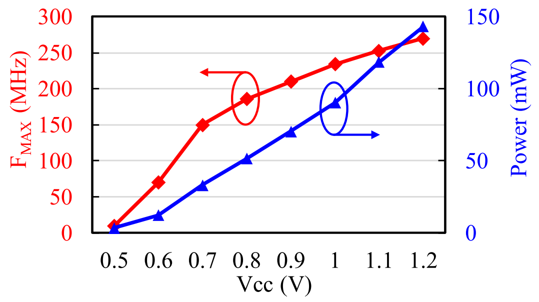


Fig. 17. Measured maximum frequency and power consumption.

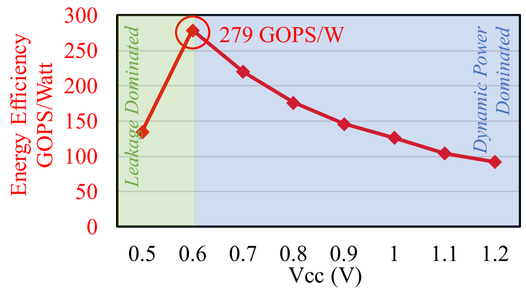


Fig. 18. Measured energy efficiency.

contain fields that include destination, mode, and operands and other fields for detailed masks or configurations. A complete discussion of all the instructions in the ISA is not needed; this is to say that the programmability provides us with the software stack that enables a large class of distributed optimizations to be efficiently executed. During the initialization phase mentioned in Section II, the instructions and the initial data are scanned into the instruction cache and data cache, respectively. Once the initialization is done, a “start” signal is broadcast to all the cores and the iterations begin until the convergence criteria are fulfilled.

A. Computation Controller

As mentioned in Section III, P-DSP takes up to four data inputs concurrently; thus fetching the data and keeping it ready for the P-DSP to access becomes critical. Furthermore, in order to support various operations with limited instruction cache, setting the corresponding reading pattern and reading size is also important. Thus, by setting the initial address, the desired size, and the desired operation, the computation controller automatically determines the corresponding target address and the pattern of reading a chunk of data from the local memory depending on the kind of operation (i.e., scalar operations, vector operations, matrix multiplications, and so on), then fetches the next elements in order and buffers them for the P-DSP to access. Thus, a combination of the instruction cache and fixed-function instruction decoders allows minimization of the overall gate count.

B. Communication Controller

As described in Section III-D, gather and scatter mechanisms (shown in Fig. 8) play an essential role in OPTIMO, and to make it efficient, a “mask” is associated with each corresponding “send” or “receive” instruction, which is used

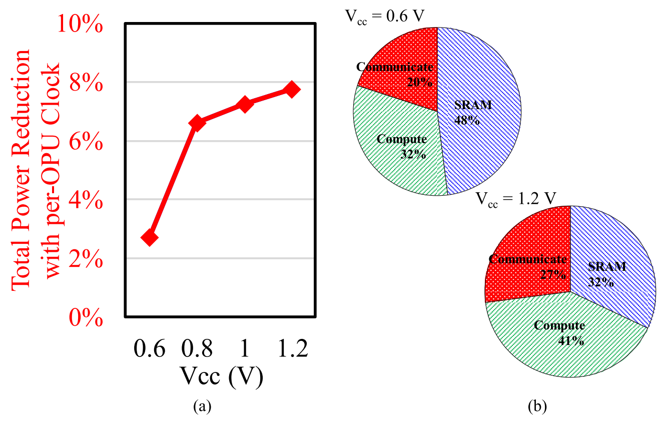


Fig. 19. (a) Total power reduction for asynchronous design (per-OPU clocking) compared to a purely sequential design. (b) Measured breakdown of power consumption.

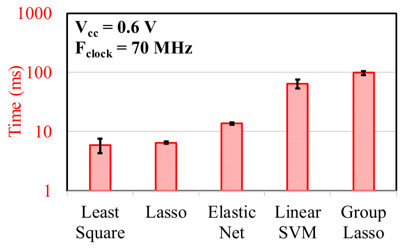


Fig. 20. Measured algorithm-level benchmarking showing the time to compute for six template algorithms. The errors bars show different problem instances that were characterized.

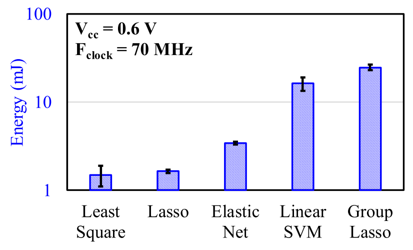


Fig. 21. Measured algorithm-level benchmarking showing the energy to compute for six template algorithms. The errors bars show different problem instances that were characterized.

to enable/disable up to eight neighborhood links depending on the operation. An example of mask enabling only the horizontal and vertical links is shown in Fig. 15.

From the "sender's" side, the sequence of data that it wishes to send is first buffered in the TX buffer. Then, along with the proper mask, the data are broadcast to the links that are enabled in the present iteration. Since the unbalanced masks between "sender" and "receiver" will result in deadlocks, it relies on the compiler to guarantee that the masks as well as the length of data sequence are properly configured and matched.

C. P-DSP Controller

A total of 9 bits of configuration are aggregated in the instruction and will be decoded inside the P-DSP. Since the P-DSP supports up to four concurrent inputs and is composed of three pipeline stages (shown in Fig. 7) (b), by changing and permuting the configurations, it supports up to 768 different kinds of computations. In addition to that, the number of cycles required is also varied depending on the instruction and the configuration, which is shown in Fig. 16.

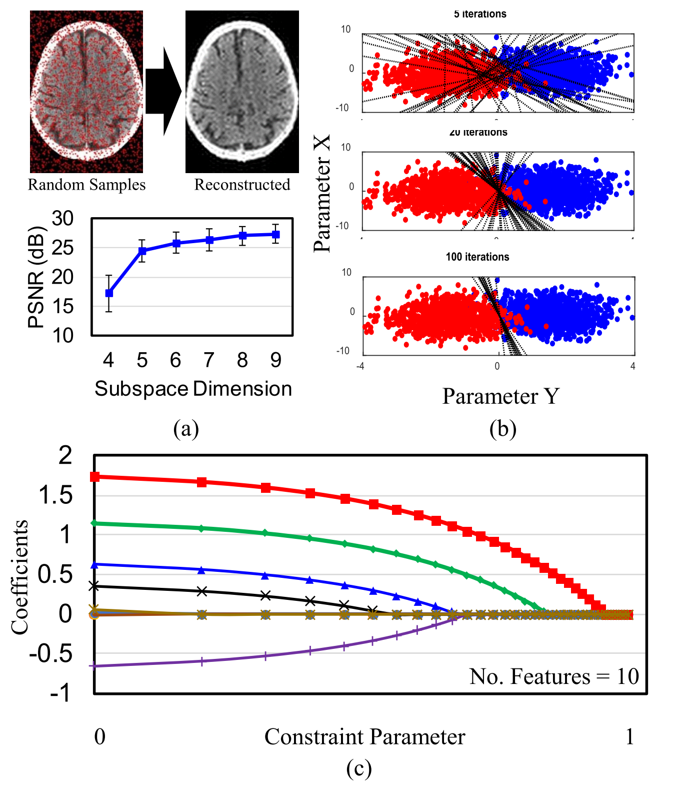


Fig. 22. Application of OPTIMO in (a) MRI image reconstruction, (b) binary SVM, and (c) Lasso feature extraction for sample problems.

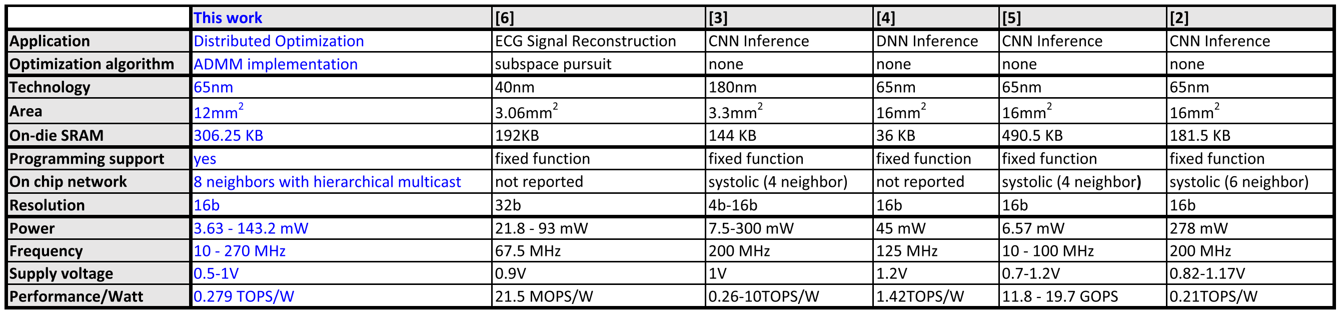
D. Branch Controller

In order to support a dynamic program flow, the ability to execute different instructions on different register values is crucial. By comparing the expected value with the target register value, we can jump to the instruction with the resulting offset and execute a desired branching operation. The current design provides full branch control and allows effective programming of a large class of algorithms.

V. Measured Results

The test chip is packaged in a QFN package and integrated on a PCB with the necessary passives and connectors onboard. It is programmed via serial scan through an external field-programmable gate array (FPGA). Before the system starts, the instructions for each OPU are scanned in and followed by a "start" signal, the FPGA then waits for the "done" signal of the system. Measured electrical performance and algorithm-level benchmarking are presented here.

Fig. 17 shows the measured power–performance tradeoff showing a peak F_{MAX} (in a synchronous setting) of 270 MHz (at 0.5 V) and an operation down in 0.5 V (with $F_{\text{MAX}} = 10$ MHz). Fig. 18 shows that as the operating voltage is reduced, the dynamic energy scales as V^2, whereas the time to complete the computation increases, thereby increasing active leakage power. The peak energy efficiency, considering both



	This work	[6]	[3]	[4]	[5]	[2]
Application	Distributed Optimization	ECG Signal Reconstruction	CNN Inference	DNN Inference	CNN Inference	CNN Inference
Optimization algorithm	ADMM implementation	subspace pursuit	none	none	none	none
Technology	65nm	40nm	180nm	65nm	65nm	65nm
Area	12mm^2	3.06mm^2	3.3mm^2	16mm^2	16mm^2	16mm^2
On-die SRAM	306.25 KB	192KB	144 KB	36 KB	490.5 KB	181.5 KB
Programming support	yes	fixed function	fixed function	fixed function	fixed function	fixed function
On chip network	8 neighbors with hierarchical multicast	not reported	systolic (4 neighbor)	not reported	systolic (4 neighbor)	systolic (6 neighbor)
Resolution	16b	32b	4b-16b	16b	16b	16b
Power	3.63 - 143.2 mW	21.8 - 93 mW	7.5-300 mW	45 mW	6.57 mW	278 mW
Frequency	10 - 270 MHz	67.5 MHz	200 MHz	125 MHz	10 - 100 MHz	200 MHz
Supply voltage	0.5-1V	0.9V	1V	1.2V	0.7-1.2V	0.82-1.17V
Performance/Watt	0.279 TOPS/W	21.5 MOPS/W	0.26-10TOPS/W	1.42TOPS/W	11.8 - 19.7 GOPS	0.21TOPS/W

Fig. 23. Comparison of the proposed array processor with competitive spatial-array processors. The proposed design addresses distributed optimization that presents a more complex data flow and computes faster than traditional CNN and DNN inference architectures.

dynamic and leakage power, is measured at 0.6 V where we note a peak efficiency of 0.279 TOPS/W. Below 0.6 V, the design is leakage dominated due to the large (306.25 kB) on-die SRAM. It should also be noted that an operation here represents the execution of a single pipeline stage of the P-DSP and is computationally more demanding than MAC operations that are often considered as a benchmark for signal processing or NN accelerators. Per-OPU DCO-based clocking reduces the overhead of routing a global clock. We measure 2.7%–7.75% power savings compared to a fully synchronous global clocking strategy. This is measured at iso-performance by ensuring that the system throughput for both the synchronous and asynchronous/mesochronous designs over a long measurement window is identical.

The power breakdown among computation, communication, and storage at 0.6 and 1.0 V is shown in Fig. 19. We see that the power consumed by all three components is almost equal at 1.0 V and the system is dominated by SRAM power (mostly leakage) at 0.6 V. Thus, distributed optimization, as presented here, shows an interesting class of algorithms where computation, communication, and storage are almost equally important in terms of power consumption.

We use the hardware prototype to execute template algorithms across multiple data sets and plot the time-to-compute and energy at 0.6 V (see Figs. 20 and 21). The data sets are generated at random, and MATLAB-based simulations are used to ensure correct functionality and convergence. The error bars indicate the range of energy and performance required for different data values in the data sets. We also note that group LASSO and linear SVM require the most number of iterations and energy–which is as expected, given the complexity of these algorithms. Although we demonstrate the capability of this near-memory spatial architecture in solving distributed optimizations, the proposed hardware and programming model can also support a variety of other array processing tasks as well, including inference in deep and convolutional NNs. The on-chip network and the P-DSPs allow flexibility to map such NN-based computing models, albeit with less energy efficiency that fixed-function accelerators.

VI. Applications

The programmable and iterative optimization solver is capable of addressing multiple applications. MRI image reconstruction from non-uniformly sampled data points is computationally challenging and requires patients to lie in the machine for a long time. Our solution uses iterative least-squares optimization [see Fig. 22(a)] to reconstruct MRI images with high peak signal-to-noise ratio (PSNR) in less than 8ms. Similarly, binary SVMs [see Fig. 22(b)], a popular choice in ML classification problems shows convergence with an increasing number of iterations (multi-class SVM records 91% accuracy on the MNIST data set, which is the state of the art). Furthermore, feature extraction with LASSO ($L1$ regularization) used in ML is shown in Fig. 22(c). In Fig. 23, a comparison with the state of the art shows: (1) a highly programmable, iterative optimization solver with peak efficiency of 279 GOPS/W; 2) a hierarchical multi-cast network for program-specific data movement; and 3) competitive energy efficiency and voltage scalability.

VII. Conclusion

In this article, we present a 49-core fully programmable spatial-array processor for solving distributed optimizations with support for a large class of algorithms and applications. We present a full-stack solution that enables full programmability, a key requirement for future high-performance systems that need to solve a large class of similar problems. We note a peak performance of 270-MHz and peak energy efficiency of 279 GOPS/W.

References

[1] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, *Distributed Optimization and Statistical Learning Via the Alternating Direction Method of Multipliers*. New York, NY, USA: Now, 2011.

[2] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze, “Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks,” *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.

[3] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst, “14.5 Envision: A 0.26-to-10TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm FDSOI,” in *IEEE ISSCC Dig. Tech. Papers*, Feb. 2017, pp. 246–247.

[4] J. Sim, J. Park, M. Kim, D. Bae, Y. Choi, and L. Kim, “14.6 A 1.42TOPS/W deep convolutional neural network recognition processor for intelligent IoE systems,” in *IEEE ISSCC Dig. Tech. Papers*, Jan./Feb. 2016, pp. 264–265.

[5] S. Choi, J. Lee, K. Lee, and H.-J. Yoo, “A 9.02mW CNN-stereo-based real-time 3D hand-gesture recognition processor for smart mobile devices,” in *IEEE ISSCC Dig. Tech. Papers*, Feb. 2018, pp. 220–222.

[6] L. L. Scharf, *Statistical Signal Processing*, vol. 98. Reading, MA, USA: Addison-Wesley, 1991.

[7] T. K. Moon and W. C. Stirling, *Mathematical Methods and Algorithms for Signal Processing*. London, U.K.: Pearson, 2000.

[8] M. Vetterli, J. Kovačević, and V. K. Goyal, *Foundations of Signal Processing*. Cambridge University Press, 2014.

[9] N. Cao, M. Chang, and A. Raychowdhury, "14.1 A 65nm 1.1-to-9.1TOPS/W hybrid-digital-mixed-signal computing platform for accelerating model-based and model-free swarm robotics," in *IEEE ISSCC Dig. Tech. Papers*, Feb. 2019, pp. 222–224.

[10] M. Chang, S. Gangopadhyay, T. Hamam, J. Romberg, and A. Raychowdhury, "Efficient signal reconstruction via distributed least square optimization on a systolic FPGA architecture," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, May 2019, pp. 1493–1497.

[11] V. Jungnickel *et al.*, "The role of small cells, coordinated multipoint, and massive MIMO in 5G," *IEEE Commun. Mag.*, vol. 52, no. 5, pp. 44–51, May 2014.

[12] E. Sharon, S. Litsyn, and J. Goldberger, "Efficient serial message-passing schedules for LDPC decoding," *IEEE Trans. Inf. Theory*, vol. 53, no. 11, pp. 4076–4091, Nov. 2007.

[13] N. A. Lynch, *Distributed Algorithms*. Amsterdam, The Netherlands: Elsevier, 1996.

[14] D. P. Bertsekas, "Incremental gradient, subgradient, and proximal methods for convex optimization: A survey," *Optim. Mach. Learn.*, vol. 2010, nos. 1–38, p. 3, 2011.

[15] J. C. Duchi, A. Agarwal, and M. J. Wainwright, "Dual averaging for distributed optimization: Convergence analysis and network scaling," *IEEE Trans. Autom. Control*, vol. 57, no. 3, pp. 592–606, Mar. 2012.

[16] S. Zhu and B. Chen, "Distributed average consensus with deterministic quantization: An ADMM approach," in *Proc. IEEE Global Conf. Signal Inf. Process. (GlobalSIP)*, Dec. 2015, pp. 692–696.

[17] N. R. Draper and H. Smith, *Applied Regression Analysis*, vol. 326. Hoboken, NJ, USA: Wiley, 1998.

[18] R. Tibshirani, "Regression shrinkage and selection via the lasso," *J. Roy. Statist. Soc., B (Methodol.)*, vol. 58, no. 1, pp. 267–288, 1996.

[19] H. Zou and T. Hastie, "Regularization and variable selection via the elastic net," *J. Roy. Statist. Soc., B (Stat. Methodol.)*, vol. 67, no. 2, pp. 301–320, 2005.

[20] J. A. K. Suykens and J. Vandewalle, "Least squares support vector machine classifiers," *Neural Process. Lett.*, vol. 9, no. 3, pp. 293–300, Jun. 1999.

[21] M. Yuan and Y. Lin, "Model selection and estimation in regression with grouped variables," *J. Roy. Statist. Soc., B (Statist. Methodol.)*, vol. 68, no. 1, pp. 49–67, 2006.

[22] L. Xiao and S. Boyd, "Fast linear iterations for distributed averaging," *Syst. Control Lett.*, vol. 53, no. 1, pp. 65–78, 2004.

[23] C. D. Manning, and H. Schütze, *Foundations of Statistical Natural Language Processing*. Cambridge, MA, USA: MIT Press, 1999.

[24] W. J. Ewens and G. R. Grant, *Statistical Methods in Bioinformatics: An Introduction*. New York, NY, USA: Springer, 2006.

[25] J. Ax, N. Kucza, M. Vohrmann, T. Jungeblut, M. Porrmann, and U. Rückert, "Comparing synchronous, mesochronous and asynchronous NoCs for GALS based MPSoCs," in *Proc. IEEE 11th Int. Symp. Embedded Multicore/Many-core Syst.–Chip (MCSoC)*, Sep. 2017, pp. 45–51.

Muya Chang (S'16) is a Dual-Degree Graduate Student with the Georgia Institute of Technology (Georgia Tech), Atlanta, GA, USA, where he is currently pursuing the M.S. degree in computer science and the Ph.D. degree in electrical and computer engineering (ECE).

He is a member of the Integrated Circuits and Systems Research Laboratory, Georgia Tech, and is advised by A. Raychowdhury, the ECE Associate Professor. His research interests include energy-efficient hardware design for distributed optimizations.

Li-Hsiang Lin is currently pursuing the Ph.D. degree in industrial engineering with the H. Milton Stewart School of Industrial and Systems Engineering, Georgia Institute of Technology (Georgia Tech), Atlanta, GA, USA. His specialization is statistics, with a minor in machine learning and operations research.

His research interests include computer experiments, nonparametric modeling, and developing new statistical methodologies in engineering applications, especially in electronics engineering and biomechanical engineering.

Justin Romberg (F'18) received the B.S.E.E., M.S., and Ph.D. degrees from Rice University, Houston, TX, USA, in 1997, 1999, and 2004, respectively.

From fall 2003 to fall 2006, he was a Post-Doctoral Scholar in applied and computational mathematics with the California Institute of Technology, Pasadena, CA, USA. He spent the summer of 2000 as a Researcher at Xerox PARC, Palo Alto, CA, USA, the fall of 2003 as a Visitor at the Laboratoire Jacques-Louis Lions, Paris, France, and the fall of 2004 as a Fellow at the Institute for Pure and Applied Mathematics, University of California at Los Angeles, Los Angeles, CA, USA. In fall 2006, he joined the ECE Faculty, Center for Signal and Image Processing, Atlanta, GA, USA, as a member. He is currently a Professor with the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta.

Dr. Romberg received the ONR Young Investigator Award in 2008 and the PECASE Award and the Packard Fellowship in 2009 and was named the Rice University Outstanding Young Engineering Alumnus in 2010. From 2006 to 2007, he was a Consultant for the TV show "Numb3rs." From 2008 to 2011, he was an Associate Editor for the IEEE TRANSACTIONS ON INFORMATION THEORY.

Arijit Raychowdhury (SM'13) received the B.E. degree in electrical and telecommunication engineering from Jadavpur University, Kolkata, India, in 2001, and the Ph.D. degree in electrical and computer engineering from Purdue University, West Lafayette, IN, USA, in 2007.

His industry experience includes five years as a Staff Scientist at the Circuits Research Lab, Intel Corporation, Santa Clara, CA, USA, and a year as an Analog Circuit Researcher at Texas Instruments Inc., Santa Clara. In January 2013, he joined the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, USA, where he is currently a Professor. From 2013 to July 2019, he was an Associate Professor and held the ON Semiconductor Junior Professorship with the Georgia Institute of Technology. He is currently the Co-Director of the Georgia Tech Quantum Alliance, Atlanta. He has published over 170 articles in journals and refereed conferences and multiple IEEE and ACM journals. He holds more than 25 U.S. and international patents. His research interests include low-power digital and mixed-signal circuit design, design of power converters, and sensors and exploring interactions of circuits with device technologies.

Dr. Raychowdhury is the winner of IEEE/ACM Innovator Under 40 Award, the NSF CISE Research Initiation Initiative Award (CRII) in 2015, the Intel Labs Technical Contribution Award in 2011, the Dimitris N. Chorafas Award for Outstanding Doctoral Research in 2007, the Best Thesis Award from the College of Engineering, Purdue University, in 2007, the SRC Technical Excellence Award in 2005, the Intel Foundation Fellowship in 2006, the NASA INAC Fellowship in 2004, and the Meissner Fellowship in 2002. He and his students have won eleven best paper awards over the years.