OPTIMO: A 65nm 279GOPS/W 16b 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,* Arijit Raychowdhury, *Senior Member, IEEE*

***Abstract*—This paper presents OPTIMO, a 65nm, 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 we measure a peak energy-efficiency of 279 GOPS/W.**

***Index Terms*—Optimizations, Array-processing, Multi-cast network, Distributed, ADMM, Near-Memory Computing**

I. Introduction

The explosion of big-data problems arising in statistics, machine learning (ML), image processing, 5G systems and other related areas [1] have 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. Current generation of such hardware are mostly geared towards inference in neural networks (NNs). However, looking beyond the success of neural network (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 hyper-spectral) [10], resource-allocation in 5G massive MIMO networks [11] and solving inverse problems such as LDPC decoding [12]. Currently, most of these algorithms are solved in GPUs and CPUs; however, with the wide-spread proliferation of machine learning, embedded signal processing, computational imaging etc., there is a growing demand for solving such optimizations both at edge-nodes as well as the cloud.

Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, 30332 USA Email: mchang87, llin79@gatech.edu, jrom, arijit.raychowdhury@ece.gatech.edu

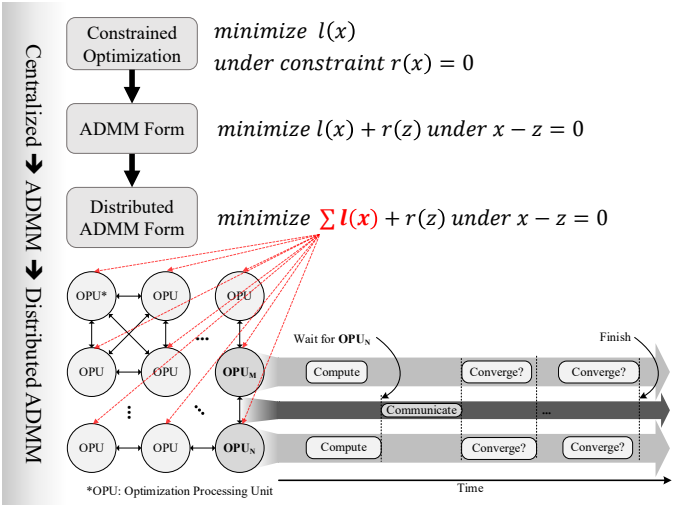


Fig. 1: OPTIMO: A spatial array processor for solving distributed optimizations via distributed ADMM.

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 dimensionaility of the task, namely, the size of the data-set or the feature-set. This requires innovative algorithms as well as hardware-algorithm co-design. Of increasing importance, are distributed optimization algorithms where different processing elements can work on different parts of the data or features; and then communicate their local solutions with each other to, 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 the next section; but

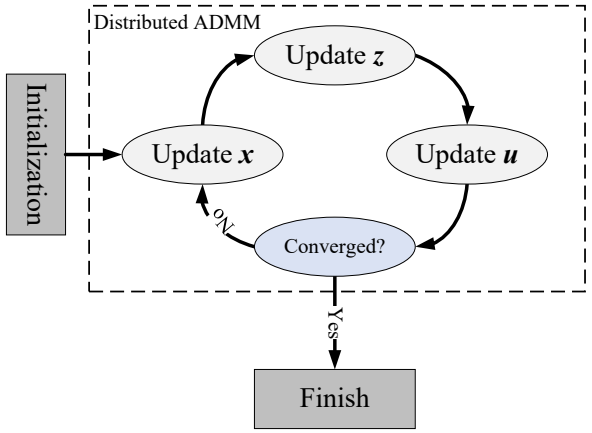


Fig. 2: Program flow for distributed ADMM.

it suffices to say that ADMM is one of the most common iterative algorithms for solving a large class of optimization problems and interested readers are pointed to [1] for a detailed survey. From a hardware perspective, ADMM, particularly in its distributed form is an 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 re-configurable 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 the next section, but the information flow is captured in this diagram.

In this paper, 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

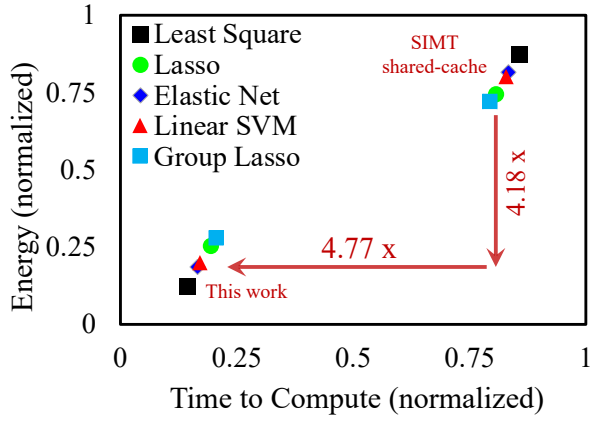


Fig. 3: This work (measured) shows 4.77x (4.18x) improvement in energy (performance) compared to a GPU-style SIMT machine (simulated).

[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 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 (Fig. 3), we note a 4.77x (4.18x) 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 machine learning models, computational imaging, signal processing etc.

II. An Overview of the Algorithm

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

A. Alternating Direction Method of Multipliers (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 2 subvectors, and its generalization to the case of multiple subvectors is straightforward and is omitted here.

The original form of ADMM with 2 subvectors denoted as $x \in \mathbf{R}^n$ and $z \in \mathbf{R}^m$ solves the problem expressed as

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

where $A \in \mathbf{R}^{p \times n}$, $B \in \mathbf{R}^{p \times m}$, and $c \in \mathbf{R}^p$. We assume 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:

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

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 which starts from arbitrary initial values $x^{(0)}, z^{(0)}$, and $y^{(0)}$, and update using the following updated rules:

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

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

B. Distributed ADMM

By splitting up a 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 parallely 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, machine learning, 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:

$$\begin{aligned} &\min_x \ l(x)+r(z) \\ &\text{subject to } x = Az-b, \end{aligned} \tag{5}$$

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 below.

1) Splitting across data: In most classical statistical estimation and machine learning 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,

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

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:

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

the optimization problem (4) becomes

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

or in the ADMM form

$$\begin{aligned} &\min_{x_1,x_2,\cdots,x_N} \ \sum_i^N l_i(A_ix_i-b_i)+r(z) \\ &\text{subject to } x_i = z \text{ for } i=1,\cdots,N. \end{aligned} \tag{6}$$

Following the ADMM algorithm described in the previous section, 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 machine learning. For example, when $l_i(A_ix-b_i) = ||A_ix-b_i||_2^2$ 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

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

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 $Ax = \sum_{i=1}^N A_ix_i$, *i.e.*, A_ix_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 isin 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:

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

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 above 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 machine learning community namely as Least Square optimization [17], Lasso [18], Group Lasso [21], Elastic Net [19], Support Vector Machines [20], and Distributed Averaging [22]. The table of loss functions and the regularization functions for each template problem are shown in Fig. 4.

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	$\|x_i\|^2$	$y_k(a_k^Tx_i+b)$	Classification
Distributed Averaging	$(1/2)\|Ax-b\|_2^2$	-	Large scale modeling and prediction

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

One thing to keep in mind is that even though all the six algorithms follow the same program flow as 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. Further, 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 paper, the hardware architecture and the programming model provides 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 the next section.

III. An Overview of the System Architecture

A. System Architecture

Fig. 5 illustrates the chip-architecture where 49 programmable 16b OPUs (optimization processing units) are capable of (1) computing locally and iteratively and (2) transmitting/receiving data from the neighbors. The chip boundary has communication interfaces to the PCB that contain: (1) Scan ports (2) System control ports (3) Clock ports. High level program-code and data-sets are translated to instruction and data, scanned in to 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 (1) after a fixed number of iterations, or (2) 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 16b 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 8b and we have determined that 16b precision yields the same results as floating point for the thousands of image and signal processing data-sets that we have analyzed. Further, ADMM is forgiving in terms of quantization error, and the system converges because of the iterative nature of the algorithm.

In the next section, The detail architecture of each OPU is described.

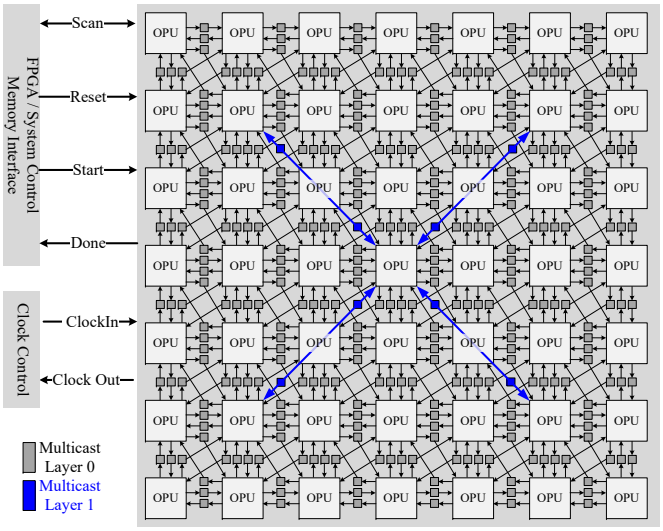


Fig. 5: System Architecture showing the 49 OPUs and a 2-layer multi-cast on-chip network.

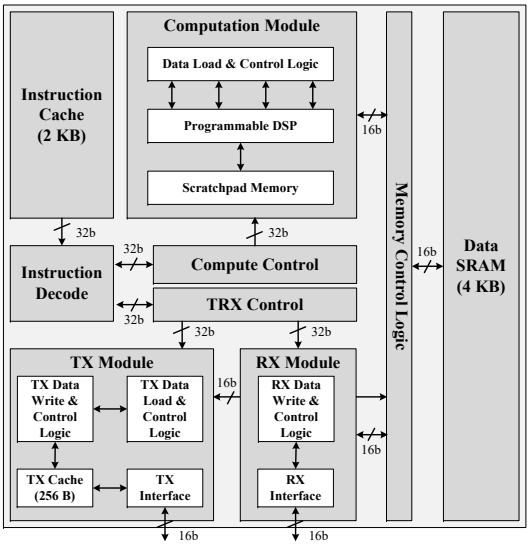


Fig. 6: Architecture of the Optimization Processing Unit (OPU) showing the principal modules.

B. Optimization Processing Unit

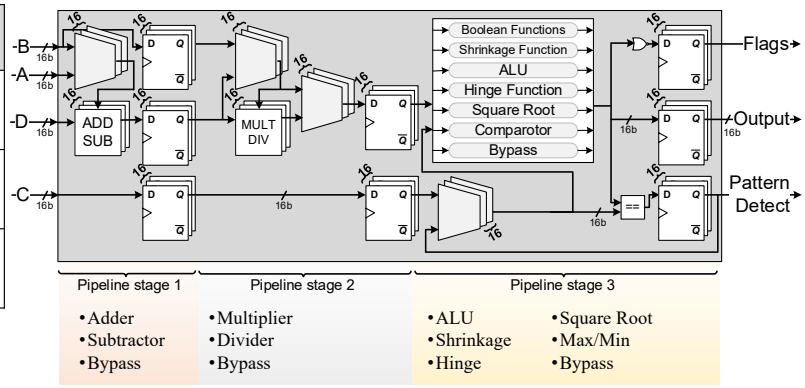
One of the important challenges for spatial-array architectures is scalability. In this design, we ensure that the IO pins for each OPU are placed in a symmetric fashion such that the OPUs can easily abut. Each OPU features (Fig. 6): (1) one computation module consisting of a programmable digital signal processor (P-DSP), a scratchpad memory and control logic, (2) 2KB of instruction cache, (4) 4KB of data memory (for local data R/W), and (5) a transceiver module for the gather and scatter processes. Programming is supported via 32b instructions, which will be described in detail in Section IV and each inter-OPU data-movement is supported on dedicated links. For this work, we mainly focused on how such architecture relates to iterative optimization problems; therefore we assume the weights and data can fit into each OPU. For more complex problems, the architecture can remain the same albeir with a more sophisticated 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 is 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 FIFOs enabling fast and parallel data exchange across clock-crossing boundaries [25]. Fig. 7 (a) further illustrates: (1) the number of instructions supported by each module, (2) the commonly-used macro functions and the number

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 instruction set architecture with a 32-b Instruction format and macro functions. (b) Programmable DSP Architecture showing a 3-stage pipeline.

of instructions per function, and (3) their usage in the six template algorithms.

C. Programmable DSP

Fig 7 (b) illustrates 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 16b 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 detail of the 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, column) interact via a 2-layered multi-cast network with (1)*layer-0* establishing near-neighbor (neighborhood of 8) bi-directional connections and (2) *layer-1* connecting 4 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, and
4) a **final scatter** (step 2 and 3) process where the cluster-centers spreads 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 optmizations as the system computes locally, distributes information globally and iterates to reach concensus. We compare the proposed hierarchical multi-cast network with networks that allow 4 or 6 connections to the neighbors as is common in convolutional and deep neural networks [2]. It is intuitive to understand that instead of connections to all the 6 neighbors, consensus data can alo be transmitted by just connecting to the four near neighbors

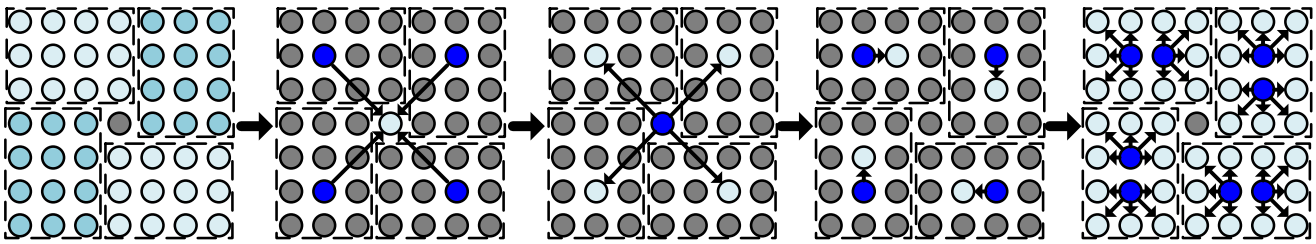


Fig. 8: Gather and scatter processes of communication enabled by a hierarchical on-chip network result in fast convergence to a global consensus.

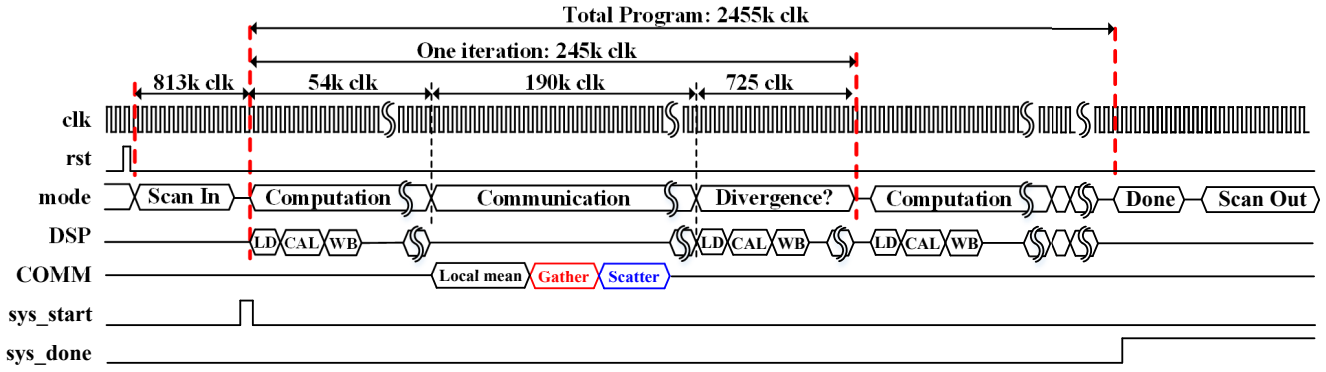


Fig. 9: Timing diagram of showing the various steps of a template optimization problem.

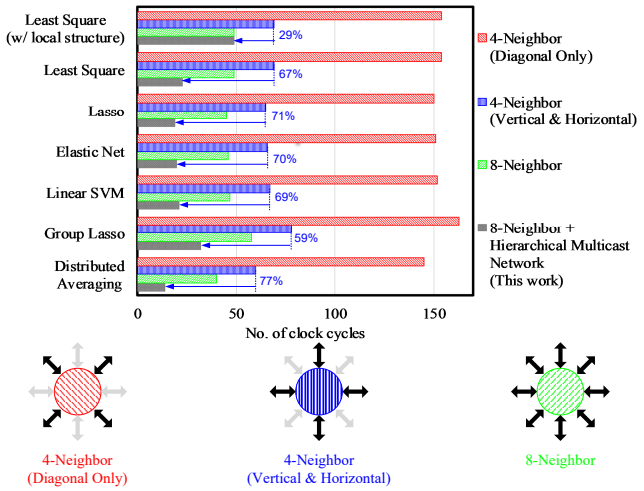


Fig. 10: Time for convergence for template algorithms as a function of their connectivity with their neighbors.

(as found in Google's TPU). However, this comes at a cost of increases number of iterations. Architectural and network simulations of various optimization algorithms on more than 10000 random data-sets reveal a 29%-77% reduction in convergence time compared to a fixed, 4-neighbor connection (Fig. 10).

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) 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 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 asynchornous clocks per-OPU, the clock-crossing FIFO features a 64b buffer and operates on a 4-phase handshaking scheme, as shown in Fig. 11. The example timing diagram for executing ADMM is shown in Fig. 9.

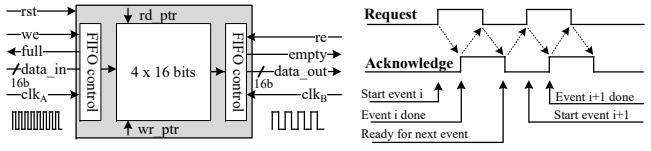


Fig. 11: FIFO Architecture and the corresponding timing diagram.

F. Die micrograph and chip characteristics

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

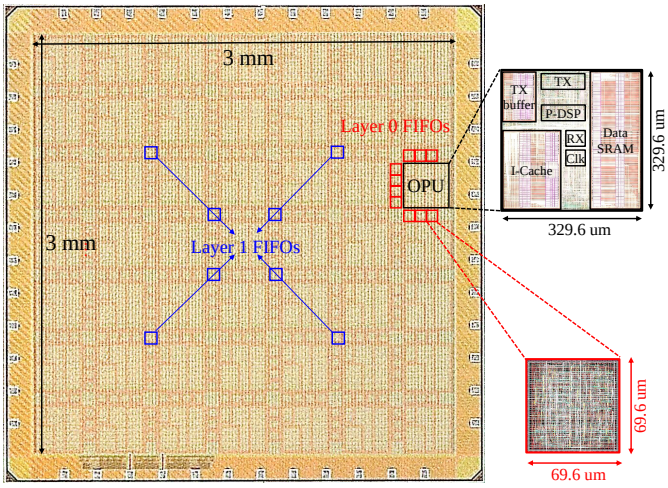


Fig. 12: Die micrograph

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.

IV. Instruction Set Architecture and Programming

As mentioned in section II, the complexity and details of the algorithmic steps to be followed to update x and z 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 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: (1) Computation Controller (2) Communication Controller (3) Programmable-DSP and (4) Branch Controller. As shown in Fig. 14, the instructions are 32 bits long and contain fields that include *destination*, *mode*, *operands* and other fields for detailed masks or configurations. A complete discussion of all the instructions in the ISA are not needed; suffice 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 is 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

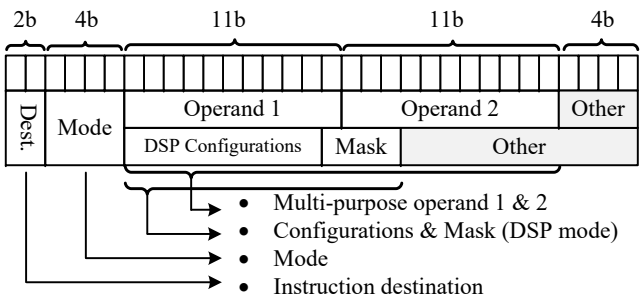


Fig. 14: 32-b Instruction format.

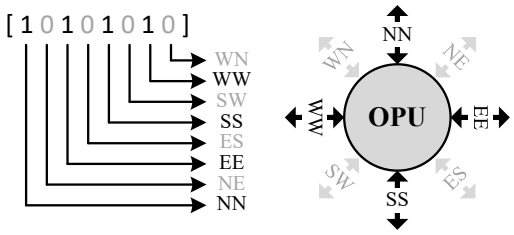


Fig. 15: Example of a 'mask' configuration.

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, or matrix multiplications...etc), 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 allow 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 to enable/disable up to 8 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 is broadcast to the links that are enabled in present iteration. Since the unbalanced masks between 'sender' and 'receiver' will result in dead locks, it relies on the compiler to guarantee the masks as well as the length of data sequence are properly configured and matched.

C. Programmable-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.

D. Branch Controller

In order to support a dynamic program flow, the ability to execute different instructions on different register values is cru-

	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.

cial. 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 on board. It is programmed via serial scan through an external 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 illustrates the measured power-performance trade-off showing a peak F_{MAX} (in a synchronous setting) of 270 MHz (at 0.5V) and an operation down in 0.5V (with F_{MAX} = 10 MHz). Fig. 18 shows 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 dynamic and leakage power, is measured at 0.6V where we note a peak-efficiency of 0.279TOPS/W. Below 0.6V, the design is leakage dominated owing to the large (306.25 KB) on-die SRAM. It should also be noted that an operation here represents execution of a single pipeline-stage of the P-DSP and is computationally more demanding than MAC operations which are often considered as a benchmark for signal processing or neural network 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 asynchornous/mesochoronous designs over a long measurement window is identical.

The power-breakdown among computation, communication and storage at 0.6 V and 1.0 V is shown in Fig. 19. We see that the power consumed by all three components are almost equal at 1.0V and the system is dominated by SRAM power (mostly leakage) at 0.6V. 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.

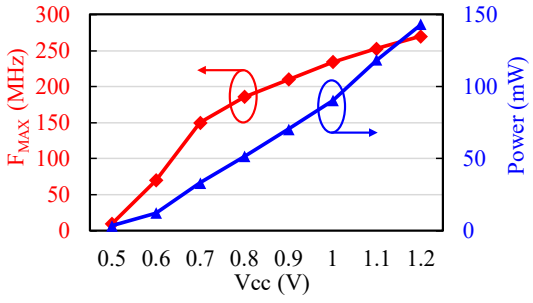


Fig. 17: Measured maximum frequency and power consumption.

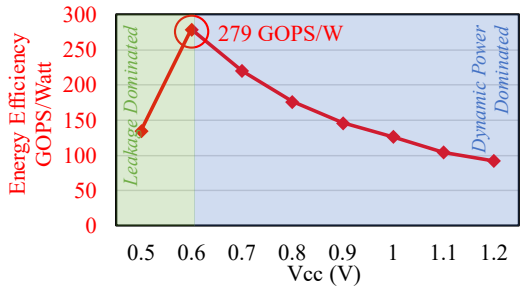


Fig. 18: Measured energy efficiency.

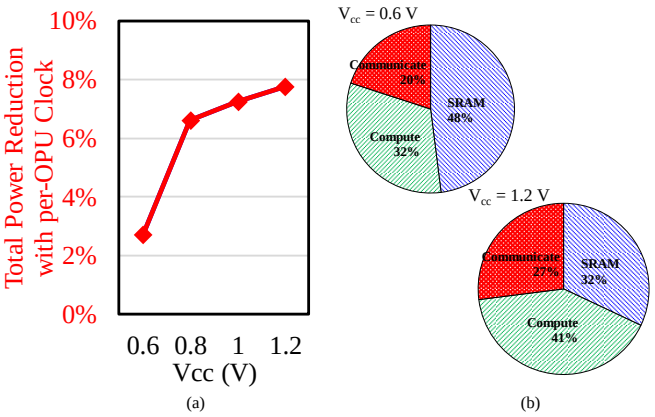


Fig. 19: (a) Total power reduction for asynchronous design (per-OPU clocking) compared to a purely sequential design, (b)Measured break-down of power consumption.

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 (Fig. 20 and Fig. 22). 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 neural networks. The on-chip network and the programmable P-DSPs allow flexibility to map such neural network based computing models, albeit with less energy-efficiency that fixed-function accelerators.

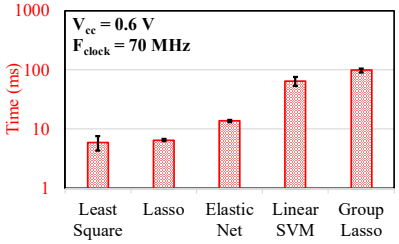


Fig. 20: Measured algorithm-level benchmarking showing the time to compute for six template algorithms. The errors bars show different problem instances that were characterized.

	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. 21: Comparison of the proposed array-processor with competitive spatial-array processors. The proposed design addresses distributed optimization which presents a more complex data-flow and compute than traditional CNN and DNN inference architectures.

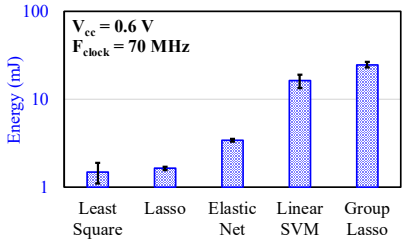


Fig. 22: Measured algorithm-level benchmarking showing the energy to compute for six template algorithms. The errors bars show different problem instances that were characterized.

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 (Fig. 23 (a)) to reconstruct MRI images with high PSNR in less than 8ms. Similarly binary SVMs (Fig. 23 (b)), a popular choice in ML classification problems shows convergence with increasing number of iterations (multi-class SVM records 91% accuracy on the MNIST data-set, which is the state-of-the-art). Further, feature extraction with LASSO (L1 regularization) used in ML, is shown in Fig. 23 (c). In Fig. 21, a comparison with the state-of-the-art shows a (1) a highly-programmable, iterative optimization solver with peak efficiency of 279GOPS/W (2) a hierarchical multi-cast network for program-specific data-movement and (3) competitive energy-efficiency and voltage-scalability.

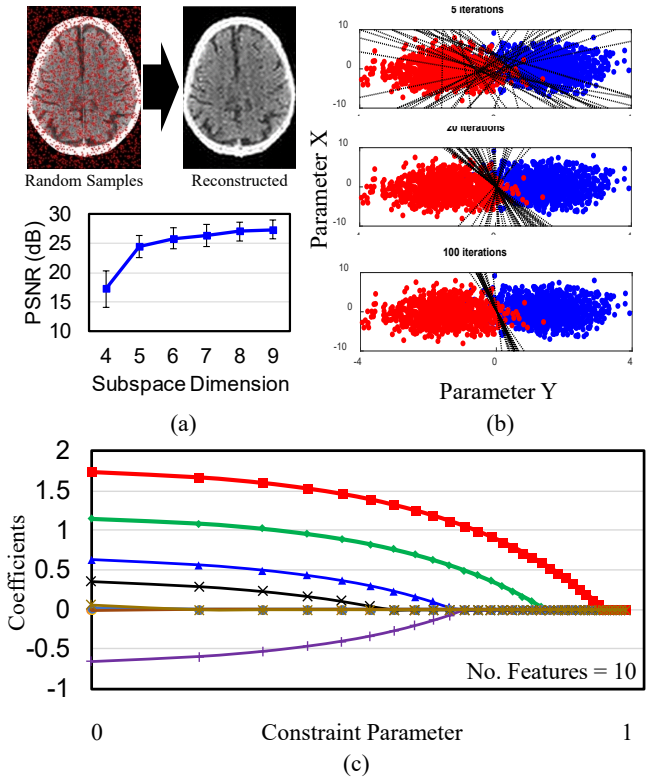


Fig. 23: Application of OPTIMO in (a) MRI image reconstruction (b) Binary SVM (c) Lasso feature extraction for sample problems.

VII. Conclusions

In this paper 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 which 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 270MHz and peak energy-efficiency of 279 GOPS/W.

VIII. Acknowledgements

This project was partly supported by the Semiconductor Research Corporation under grant JUMP CBRIC task ID 2777.006. It was also supported by the National Science Foundation under grant 1640081, and the 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 Research Task IDs 2698.001 and 2698.002

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*. now, 2011.

[2] Y. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, 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 *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 246–247, Feb 2017.

[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 *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 264–265, Jan 2016.

[5] S. Choi, J. Lee, K. Lee, and H. Yoo, "A 9.02mw cnn-stereo-based real-time 3d hand-gesture recognition processor for smart mobile devices," in *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, pp. 220–222, Feb 2018.

[6] L. L. Scharf, *Statistical signal processing*, vol. 98. Addison-Wesley Reading, MA, 1991.

[7] T. K. Moon and W. C. Stirling, *Mathematical methods and algorithms for signal processing*. 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 *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, pp. 222–224, Feb 2019.

[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 *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1493–1497, May 2019.

[11] V. Jungnickel, K. Manolakis, W. Zirwas, B. Panzner, V. Braun, M. Lossow, M. Sternad, R. Apelfrojd, and T. Svensson, "The role of small cells, coordinated multipoint, and massive mimo in 5g," *IEEE Communications Magazine*, vol. 52, no. 5, pp. 44–51, 2014.

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

[13] N. A. Lynch, *Distributed algorithms*. Elsevier, 1996.

[14] D. P. Bertsekas, "Incremental gradient, subgradient, and proximal methods for convex optimization: A survey," *Optimization for Machine Learning*, vol. 2010, no. 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 Transactions on Automatic control*, vol. 57, no. 3, pp. 592–606, 2011.

[16] S. Zhu and B. Chen, "Distributed average consensus with deterministic quantization: An admm approach," in *2015 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pp. 692–696, IEEE, 2015.

[17] N. R. Draper and H. Smith, *Applied regression analysis*, vol. 326. John Wiley & Sons, 1998.

[18] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 58, no. 1, pp. 267–288, 1996.

[19] H. Zou and T. Hastie, "Regularization and variable selection via the elastic net," *Journal of the royal statistical society: series B (statistical methodology)*, vol. 67, no. 2, pp. 301–320, 2005.

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

[21] M. Yuan and Y. Lin, "Model selection and estimation in regression with grouped variables," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 68, no. 1, pp. 49–67, 2006.

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

[23] C. D. Manning, C. D. Manning, and H. Schütze, *Foundations of statistical natural language processing*. MIT press, 1999.

[24] W. J. Ewens and G. R. Grant, *Statistical methods in bioinformatics: an introduction*. Springer Science & Business Media, 2006.

[25] J. Ax, N. Kucza, M. Vohrmann, T. Jungeblut, M. Porrmann, and U. Rckert, "Comparing synchronous, mesochronous and asynchronous nocs for gals based mpsocs," in *2017 IEEE 11th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pp. 45–51, Sep. 2017.

Muya Chang is a dual degree graduate student at Georgia Tech, where he is pursuing his Ph.D. in Electrical and Computer Engineering (ECE) and M.S. in Computer Science. He is a member of the Integrated Circuits and Systems Research Lab and is advised by ECE Associate Professor Arijit Raychowdhury. His research interests include energy-efficient hardware design for distributed optimizations.

Li-Hsiang Lin is a PhD candidate in Industrial Engineering at H. Milton Stewart School of Industrial and Systems Engineering at Georgia Tech. 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 electronical engineering and biomechanical engineering.

Justin Romberg Dr. Justin Romberg is a professor in the School of Electrical and Computer Engineering at the Georgia Institute of Technology. Dr. Romberg received the B.S.E.E. (1997), M.S. (1999) and Ph.D. (2004) degrees from Rice University in Houston, Texas. From fall 2003 until fall 2006, he was a Postdoctoral Scholar in Applied and Computational Mathematics at the California Institute of Technology. He spent the summer of 2000 as a researcher at Xerox PARC, the fall of 2003 as a visitor at the Laboratoire Jacques-Louis Lions in Paris, and the fall of 2004 as a Fellow at UCLA's Institute for Pure and Applied Mathematics. In fall 2006, he joined the ECE faculty as a member of the Center for Signal and Image Processing. In 2008, he received an ONR Young Investigator Award, in 2009 he received a PECASE award and a Packard Fellowship, and in 2010 he was named a Rice University Outstanding Young Engineering Alumnus. In 2006-2007, he was a consultant for the TV show "Numb3rs," and from 2008-2011, he was an Associate Editor for the IEEE Transactions on Information Theory.

Arijit Raychowdhury Arijit Raychowdhury is currently a Professor in the School of Electrical and Computer Engineering at the Georgia Institute of Technology where he joined in January 2013. He is currently the co-director of the Georgia Tech Quantum Alliance. From 2013 to July 2019 he was an Associate Professor and held the ON Semiconductor Junior Professorship in the department. He received his Ph.D. degree in Electrical and Computer Engineering from Purdue University (2007) and his B.E. in Electrical and Telecommunication Engineering from Jadavpur University, India (2001). His industry experience includes five years as a Staff Scientist in the Circuits Research Lab, Intel Corporation, and a year as an Analog Circuit Researcher with Texas Instruments Inc. His research interests include low power digital and mixed-signal circuit design, design of power converters, sensors and exploring interactions of circuits with device technologies. Dr. Raychowdhury holds more than 25 U.S. and international patents and has published over 170 articles in journals and refereed conferences. multiple IEEE and ACM journals. He is the winner of IEEE/ACM Innovator under 40 award; the NSF CISE Research Initiation Initiative Award (CRII), 2015; Intel Labs Technical Contribution Award, 2011; Dimitris N. Chorafas Award for outstanding doctoral research, 2007; the Best Thesis Award, College of Engineering, Purdue University, 2007; SRC Technical Excellence Award, 2005; Intel Foundation Fellowship, 2006; NASA INAC Fellowship, 2004; the Meissner Fellowship 2002. He and his students have won eleven best paper awards over the years. Dr. Raychowdhury is a Senior Member of the IEEE.