High-Throughput Training of Deep CNNs on ReRAM-Based Heterogeneous Architectures via Optimized Normalization Layers

Biresh Kumar Joardar, *Member, IEEE*, Aryan Deshwal, *Student Member, IEEE*, Janardhan Rao Doppa, *Member, IEEE*, Partha Pratim Pande, *Fellow, IEEE*, and Krishnendu Chakrabarty, *Fellow, IEEE*

***Abstract*—Resistive random-access memory (ReRAM)-based architectures can be used to accelerate convolutional neural network (CNN) training. However, existing architectures either do not support normalization at all or they support only a limited version of it. Moreover, it is common practice for CNNs to add normalization layers after every convolution layer. In this work, we show that while normalization layers are necessary to train deep CNNs, only a few such layers are sufficient for effective training. A large number of normalization layers do not improve prediction accuracy; it necessitates additional hardware and gives rise to performance bottlenecks. To address this problem, we propose *DeepTrain*, a heterogeneous architecture enabled by a Bayesian optimization (BO) methodology; together, they provide adequate hardware and software support for normalization operations. The proposed BO methodology determines the minimum number of normalization operations necessary for a given CNN. Experimental evaluation indicates that the BO-enabled *DeepTrain* architecture achieves up to 15× speedup compared to a conventional GPU for training CNNs with no accuracy loss while utilizing only a few normalization layers.**

***Index Terms*—3-D, convolutional neural networks (CNNs), GPU, normalization, resistive random-access memory (ReRAM).**

I. Introduction

CONVOLUTIONAL neural networks (CNNs) are used in a wide spectrum of domains, e.g., self-driving cars, medical diagnosis, image recognition, etc. However, CNN training and inference are both computationally expensive tasks that necessitate a high-performance and energy-efficient hardware support. Emerging resistive random-access memory (ReRAM) has demonstrated great potential for efficient CNN training and inference [1], [2]. ReRAM crossbars can efficiently perform matrix–vector multiplications, which form the backbone of most CNN computations [1]. Prior works, such as Pipelayer [1] and AccuReD [3], have shown that ReRAM-based architectures can outperform GPUs for training CNNs while consuming less energy. In addition, ReRAM-based systems are more area efficient compared to their GPU counterparts and do not require expensive off-chip memory access due to their "in-memory" nature of computation [2].

Despite these advantages, existing ReRAM-based architectures lack one important feature that is necessary to train deep CNNs (i.e., CNNs with many layers). Existing ReRAM-based architectures lack batch normalization (BN) support, which is necessary for training deep CNNs. Deep CNNs tend to suffer from vanishing/exploding gradient problems, which can be solved using a BN layer [4]. However, BN operations are prone to data overflows and hence 32-bit floating-point representation is recommended for implementing them [5]–[7]. ReRAM-only architectures are based on 16-bit fixed-point representation that is not suitable for BN [3]. Without BN, training deep CNNs on ReRAM-based architectures can result in: 1) no meaningful training or 2) significant loss of prediction accuracy as we show in this work.

Specialized initialization schemes, such as Xavier [8] or Kaiming [9] initializations have been proposed to train CNNs in the absence of BN. In some scenarios, these techniques can enable meaningful deep CNN training using only the low-precision ReRAMs. However, as we show later, non-BN methods (such as the use of Xavier initialization) require careful hyperparameter selection (i.e., expert domain knowledge) and yet, are not effective all the time. In addition, repeated experiments to tune the hyperparameters can be expensive, particularly for deep CNNs that take longer to train. For instance, VGG-19 takes ∼2× more time to train than VGG-11 for the CIFAR-10 dataset on an Nvidia Titan Xp GPU. Hence, repeated experiments to tune the hyperparameters for VGG-19 are expensive and should be avoided (minimized). The use of BN reduces the hyperparameter dependencies and achieves high accuracy in (almost) all cases (excluding extreme scenarios such as unrealistically high/low learning rate; same for

Manuscript received December 23, 2020; revised April 1, 2021; accepted May 12, 2021. Date of publication May 25, 2021; date of current version April 21, 2022. This work was supported in part by the U.S. National Science Foundation (NSF) under Grant CNS-1955353 and Grant CNS-1955196, and in part by the USA Army Research Office under Grant W911NF-17-1-0485. The work of Biresh Kumar Joardar was supported by NSF through the Computing Research Association for the CIFellows Project under Grant 2030859. This article was recommended by Associate Editor N. K. Jha. *(Corresponding author: Partha Pratim Pande.)*

Biresh Kumar Joardar and Krishnendu Chakrabarty are with the Department of Electrical and Computer Engineering, Duke University, Durham, NC 27708 USA (e-mail: bireshkumar.joardar@duke.edu; krish@duke.edu).

Aryan Deshwal and Partha Pratim Pande are with the School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA 99164 USA (e-mail: aryan.deshwal@wsu.edu; pande@wsu.edu).

Janardhan Rao Doppa is with the College of Engineering and Architecture and the School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA 99164 USA (e-mail: jana.doppa@wsu.edu).

Digital Object Identifier 10.1109/TCAD.2021.3083684

other hyperparameters). Hence, hardware support for BN on ReRAM-based architectures is necessary to efficiently train deep CNNs.

On the other hand, introducing BN support in existing ReRAM-based architectures requires additional hardware with 32-bit floating-point support. The hardware cost clearly needs to be minimized. It is a common practice for existing CNN implementations (e.g., the CNN models from the popular Pytorch package) to incorporate a BN layer after every convolution (Conv) layer [10]. Our analysis in this work demonstrates that having a large number of BN layers is not necessary and is often counterproductive. Too many BN layers can create performance bottlenecks, consume more energy, and limit ReRAM-based architectures from reaching their full potential. Overall, it is essential to provide the right balance between conflicting requirements: 1) avoid performance, hardware, and energy overheads introduced by a large number of BN layers and 2) prevent accuracy loss due to a small number (or no) BN layers. However, determining this optimal CNN configuration is an expensive task as CNN training is computationally challenging and takes a considerable amount of time. In this article, we formulate a novel sparsity-aware Bayesian optimization (BO) problem and propose an efficient algorithm to quickly determine the minimum number of BN layers necessary for a given CNN to achieve high-performance and energy-efficient training without accuracy loss.

Overall, the main contributions of this article are listed as follows.

1) We demonstrate the importance of BN layers for training deep CNNs. We show that BN requires less extensive hyperparameter tuning than other well-known methods for successful training.
2) We propose a novel sparsity-aware BO algorithm that can quickly find the minimum number of BN layers necessary and their corresponding positions in a CNN for high-accuracy training.
3) We show that training these optimized CNNs using a 3-D ReRAM/GPU-based architecture (referred as *DeepTrain*) leads to better performance than sole GPU-based implementations.
4) We show that the knowledge learned by BO regarding the BN configurations from one dataset can be transferred to several other datasets as well, i.e., the BO optimization need not be repeated for every dataset.

The remainder of this article is organized as follows. In Section II, we present some of the relevant prior work and highlight our key contributions. In Section III, we discuss the challenges associated with training deep CNNs. Section IV introduces the salient features of the proposed *DeepTrain* architecture. The BO methodology for determining the most effective BN layers in a CNN is presented in Section V. In Section VI, we demonstrate the efficacy of the *DeepTrain* architecture. Finally, Section VII concludes this article by summarizing our key findings.

II. Related Prior Work

In this section, we present relevant prior work related to the training of deep CNNs and ReRAM-based architectures.

A. Training Deep CNNs

Vanishing and exploding gradients is a major challenge toward successful training of deep CNNs [4]. Special initialization techniques (e.g., Xavier and Kaiming initialization [8], [9]) and BN [4] can be used to address this problem. Among them, the use of BN layers is the most promising solution [11]. BN enables stable training with larger and diverse learning rates, thereby leading to faster convergence [11]. In addition, it reduces the necessity of hyperparameter tuning to improve accuracy as we show later. Traditionally, existing CNN implementations (e.g., the CNN models from the popular PyTorch package) incorporate a BN layer after every convolution (Conv) layer [10]. However, all BN layers do not contribute to successful CNN training. Having too many BN layers can adversely affect runtime and performance on ReRAM-based architectures as we demonstrate in this work. However, it is not trivial to determine which BN layers are necessary for a given CNN. In this work, we use BO over discrete space to address this problem. BO is used for optimization problems where the objective function is expensive [13]. However, existing BO algorithms are inefficient when the problem requires sparse solutions as in this work [14] (i.e., sparse in terms of the presence/absence of a BN layer after each Conv layer).

B. ReRAM-Based Architectures

Sole ReRAM-based architectures have been proposed for accelerating both inference [2], [15] and training [1], [3] for CNNs. However, these architectures do not implement BN layers due to the lack of a full-precision computing platform (most ReRAM-based architectures rely on 16-bit fixed-point representation [2]). As mentioned in [5] and [6], BN should be implemented using full precision to prevent data overflow. REGENT [16] and 3D-ReG [17] propose 3-D architectures consisting of ReRAM and GPUs for CNN training. However, the full-precision GPUs in these two architectures are used to preserve accuracy during the precision-sensitive backpropagation phase rather than for BN. The AccuReD architecture supports BN operations using the GPUs [3]. However, it uses BN after every Conv/FC layer which is wasteful and inefficient. ReGAN [18] implements a very limited BN operation on ReRAMs that works only if the divisor is of the form 2^n. However, our experiments indicate that this condition is rarely met (less than 2% of cases). Hence, for all practical purposes, ReGAN needs additional/external hardware support for implementing BN. In addition, ReGAN does not use full precision for BN and can experience overflows.

In this work, we advance the state of the art in ReRAM-based architectures for training deep CNNs by proposing a sparsity-aware BO-based methodology that can quickly determine the minimum number of BN layers necessary for a given CNN. These lightweight CNNs obtained using BO, are faster and more energy efficient than traditional NNs when trained using the *DeepTrain* architecture.

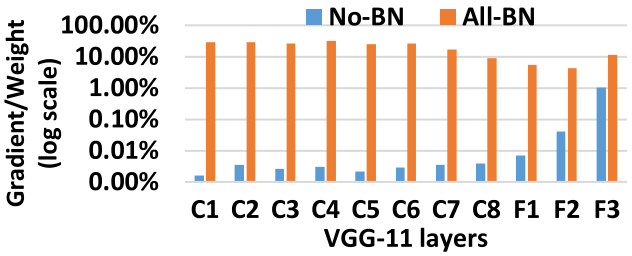

Fig. 1. Gradients normalized with respect to weights for each layer considering VGG-11with CIFAR-10 as an example.

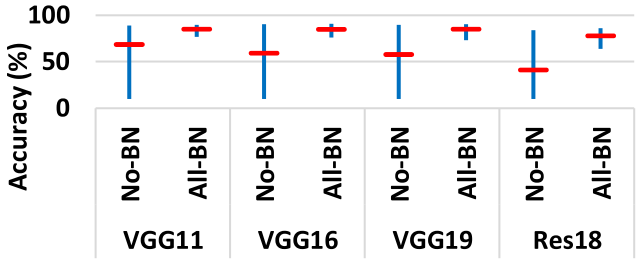

Fig. 2. Accuracy of No-BN and All-BN CNNs trained with various hyperparameter settings.

III. Deep CNN Training

In this section, we first highlight the difficulty of training deep CNNs due to the vanishing/exploding gradients problem. Next, we discuss the shortcomings of the Xavier and Kaiming initialization schemes as universal solutions and motivate the need for BN operations.

A. Exploding/Vanishing Gradients

To illustrate the impact of the vanishing/exploding gradients problem, we consider two cases: 1) *All-BN:* BN layers are placed after every Conv layer as is usually done in traditional CNNs [10] and 2) *No-BN:* BN layers are not used at all. Fig. 1 shows the average gradient values at each layer (normalized with respect to the average weight value at the same layer) for both these variants of VGG-11 [19] when trained using the CIFAR-10 dataset. For this experiment, we have used the default initialization scheme in Pytorch. Note that VGG-11 is considered as a representative example. Similar behavior is observed for the other CNN architectures considered in this work, namely, VGG-16, VGG-19 [19], and ResNet-18 [20]. As we can see from Fig. 1, there is a stark contrast in the gradient behavior between All-BN and No-BN. In the case of No-BN, we observe very small gradients (also known as vanishing gradients). The gradients are only 0.02%–0.04% of the weights in layers C1–C5. As a result, weight updates will be small, leading to poor accuracy. On the other hand, the gradients at the initial layers (C1–C5) are roughly 30% of the corresponding weights for All-BN ($\sim$1000$\times$ larger compared to No-BN). As a result, the error gradients will have a larger impact during the weight updates leading to faster training. The gradients in No-BN can also explode (i.e., too large gradients) in some cases leading to unstable training. Overall, this results in only 19.1% prediction accuracy after 50 epochs of training the No-BN VGG-11 configuration on the CIFAR-10 dataset. On the other hand, All-BN achieves 85.4% prediction accuracy under similar circumstances, indicating a successful training. Hence, it is important to address the exploding/vanishing gradients problem for training deep CNNs.

B. Xavier/Kaiming Initialization

As mentioned earlier, the use of specialized initialization schemes, such as Xavier [8] or Kaiming [9], are alternative ways to train deep CNNs in the absence of BN. By carefully setting the weights, these techniques can prevent exploding/vanishing gradients. However, as we show in Fig. 2, these methods require careful hyperparameter selection (i.e., expert domain knowledge) and yet, do not work all the time. On the contrary, the use of BN reduces the hyperparameter dependencies and achieves high accuracy in all cases considered here.

To demonstrate the sensitivity of these methods to the choice of hyperparameters, we varied five hyperparameters: 1) learning rate (LR); 2) the number of epochs; 3) LR schedule; 4) batch size; and 5) the initialization scheme (Xavier and Kaiming only; other initialization methods resulted in significantly low training accuracy in the absence of BN). The results are shown in Fig. 2. Note that these hyperparameters were chosen here as an example only. Overall, we performed CNN training with 150 different hyperparameter settings and noted the best accuracy achieved at the end of each training instance. Fig. 2 shows the range of observed accuracy (represented by the *blue line*, whose ends indicate the *minimum* and *maximum* accuracy; the *red line* represents the *average* accuracy) considering all 150 experiments with VGG-11/16/19 and ResNet-18.

Fig. 2 clearly shows that it is possible to train CNNs in the absence of BN *sometimes,* which is in line with previous work [8], [9]. However, it is not reliable and fails to train most of the time. For instance, the average accuracy of No-BN with Xavier/Kaiming initialization considering all 150 instances of training without BN is a mere 57.8% for VGG-19. This happens as multiple combinations of hyperparameter (among the 150 chosen here), either failed to train or resulted in unacceptable accuracies. This is problematic as an ML-practitioner (or user) will have to repeatedly train a CNN to find out the valid hyperparameter combination(s) for successful training. This process can be time consuming, particularly for deep CNNs and larger datasets, which require more time to train. On the other hand, CNN training with BN is more robust to the choice of hyperparameters and is effective in all cases considered here. Unlike No-BN, All-BN achieves an average accuracy (considering all 150 All-BN training instances) of 85.1% for VGG-19 indicating that all these 150 experiments succeeded. This demonstrates the shortcomings of Xavier/Kaiming initialization schemes as a universal solution to the problem of exploding/vanishing gradients. As shown in Fig. 2, all the other CNNs exhibit similar trends as well. Hence, hardware support for BN is important to train deep CNNs and should be incorporated in ReRAM-based architectures.

IV. DeepTrain Architecture

In this section, we discuss the proposed ReRAM-based architecture: *DeepTrain*, to train deep CNNs with BN support.

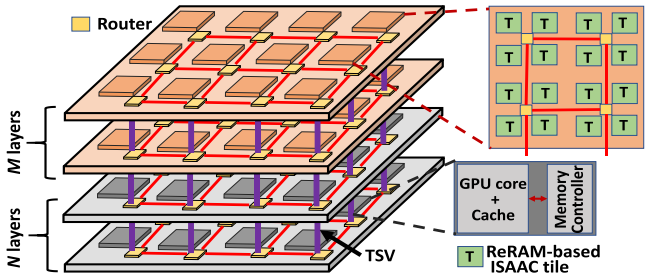

Fig. 3. Illustration of the *DeepTrain* hardware architecture.

A. Hardware Support for BN

Existing ReRAM-based designs, such as REGENT [16] and Pipelayer [1], primarily focus on accelerating the MAC-dominated Conv and FC layers to accelerate inference/training of CNNs. Unlike GPUs, which support 32-bit floating-point precision, these ReRAM-based architectures rely on 16-bit fixed-point precision for CNN training. The lower precision of data results in lightweight CNNs that require less computational and memory resources for successful training. However, this approach also presents a major roadblock toward implementing BN as it requires full-precision support [5], [6] (discussed earlier). Hence, a mixed-precision training strategy is needed, which is not possible with the existing ReRAM-only architectures. In addition, BN involves relatively more complex mathematical operations, such as division and square root [4]. To the best of our knowledge, there is no *ReRAM-only* architecture that can implement BN. ReGAN provides a partial solution to this problem that works only if the divisor is of the form 2^n [18]. As mentioned earlier, this condition is rarely satisfied (e.g., less than 2% cases in our experiments). Hence, for all practical purposes, the BN support in ReGAN is not sufficient.

Overall, a suitable hardware platform for BN must have full-precision support and should be capable of easily implementing more complex math operations such as division. In this article, we propose a 3-D architecture that consists of both ReRAMs and the full-precision computing unit to address this problem. Fig. 3 shows the envisioned architecture. In this work, we use GPUs as the choice of full-precision hardware for BN support. However, any other processing unit that can support full-precision computing (CPU, TPU, etc.) can also be used here. As shown in Fig. 3, the proposed architecture includes M ReRAM layers and N GPU layers, stacked vertically. Here, we use 3-D integration as it enables the use of disparate technologies [21] (GPUs and ReRAMs in this case). Note that fabricating heterogeneous components on a single layer of silicon is often not feasible due to manufacturing incompatibilities. We discuss each component of the proposed architecture in more detail in the next section.

It should be noted that the recently proposed AccuReD architecture uses a similar concept to include BN support in the ReRAM-based platform [3]. However, as we show later, the AccuReD architecture is suboptimal due to the use of too many BN layers (AccuReD uses the All-BN CNN configuration). The proposed architecture addresses this problem by judiciously using a limited number of BN layers, resulting in significantly more speedup for training deep CNNs compared to AccuReD.

B. Overall Architecture

Each planar layer consists of multiple GPU (or ReRAM) tiles. The planar layers are connected using through-silicon vias (TSVs). The ReRAM layers consist of ReRAM crossbars that can be used for both storage and computation (ReRAMs perform in-memory computation). This reduces the amount of traffic (data movement) necessary compared to conventional CPU/GPU-based manycore architectures. Note that the ReRAM layers can be based on any existing ReRAM-only implementations, such as ISAAC or Pipelayer, etc. In this work, without loss of generality, we implement the ReRAM tiles following ISAAC [2]. Each tile in ISAAC includes the *in-situ* multiply–accumulate (IMA) units, which consist of multiple ReRAM crossbars, along with other peripheral circuitry. In addition, each ReRAM tile connects to a router for communication via the NoC.

However, unlike ISAAC [2], *DeepTrain* is equipped with the additional GPU layers to implement BN necessary for training deep CNNs. The GPU layer consists of conventional GPU tiles (with cache) to support the precision-critical BN operations. The GPU tiles include a GPU core, cache, and a memory controller (MC) that can access data from the ReRAM layers. In addition, each tile also includes a router for communication. The GPU tiles can access the ReRAM layers using TSVs.

The GPU and ReRAM tiles are arranged in a conventional grid fashion. As ReRAM tiles are smaller than their GPU counterparts, four ReRAM tiles are clustered together following [2]. The GPU tiles and ReRAM clusters are connected via a 3-D mesh NoC where the vertical connections are established using TSVs. Here, it should be noted that mesh NoCs are not suited for multihop long-range communication. However, CNN training involves data sharing between adjacent layers only. Hence, long-range communication can be avoided by appropriately mapping the CNN layers to different processing tiles [3]. As a result, a simple NoC topology such as mesh is sufficient as the communication backbone in *DeepTrain*. In addition, a mesh NoC is more amenable to multicast support, which is necessary for CNN training.

Overall, *DeepTrain* (Fig. 3) consists of two types of processing elements that have contrasting properties.

1) *GPUs:* They are relatively slow in implementing MAC operations but provide a full-precision computing platform (32-bit floating point).
2) *ReRAMs:* They can implement efficient MAC operations but can only support low-precision computing (16-bit fixed point).

Hence, the MAC-heavy, but less precision-sensitive, Conv and FC layers are mapped to the ReRAMs. On the other hand, the computationally inexpensive but precision-critical BN layers are executed on the GPUs. To ensure that communication does not become a bottleneck, we map the CNN computations using a multicast-enabled load-balancing scheme [3]. In a CNN, each neuron sends the same data to all its connected

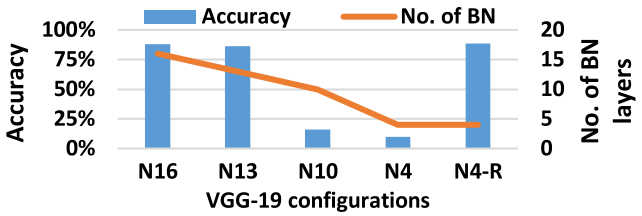

Fig. 4. Accuracy for different VGG-19 configurations with CIFAR-10.

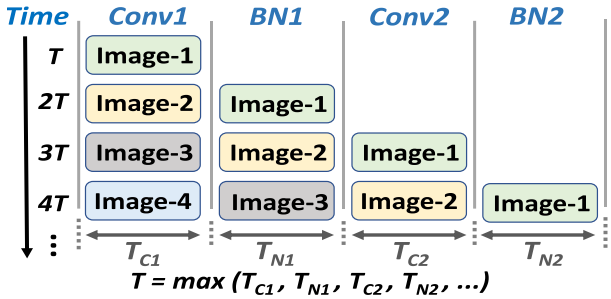

Fig. 5. Pipelined CNN training for ReRAM-based architectures following [1].

neurons in the next layer while also saving it (on another set of ReRAM cells) to use during the backpropagation phase. This results in a lot of redundant traffic as the same data needs to be transmitted to multiple destinations (ReRAM/GPU tiles). Hence, multicast-enabled mapping is necessary to avoid sending the same data repeatedly. This reduces the amount of traffic in the NoC, eliminating the possibility of hotspot creation. The use of mesh NoC in *DeepTrain* allows the incorporation of a simple *XYZ*-based tree multicast [31]. The *XYZ* tree multicast works as follows. Depending on the location of destination tiles, one or more copy of the same message is first sent along the *X*-dimension, followed by turns along the *Y*-dimension. Finally, the message packet(s) turn along the *Z*-dimension to reach all intended destinations. Overall, the mapping aims to reduce traffic hotspots in the NoC for efficient communication support for training deep CNNs. Next, we discuss the BO-based optimization algorithm that enables *DeepTrain* to achieve further performance benefits.

V. BO-Optimized CNNs

In this section, we first demonstrate the limitations of existing CNN architectures that use BN after every Conv/FC layer. Next, we present the proposed sparsity-aware BO methodology that can identify the most effective locations for BN layers in a CNN. These BO-enabled CNNs require significantly fewer BN layers than their traditional All-BN counterparts leading to better performance and energy efficiency using the proposed hardware architecture.

A. Challenges Using All-BN

From Figs. 1 and 2, we know that BN layers are important for training deep CNNs. Traditionally, BN layers are added after every Conv layer (referred as All-BN in this work) [10]. However, All-BN is suboptimal from the perspective of hardware and can adversely affect the performance of *DeepTrain* (shown later). Some of the BN layers in All-BN are redundant and do not contribute toward achieving better accuracy. Instead, these extra BN operations increase the required amount of computation, which can lead to an unnecessary performance bottleneck.

Fig. 4 shows the prediction accuracy with different VGG-19 configurations for CIFAR-10. The parameter N_i in Fig. 4 indicates i BN layers used after each of the *first* i Conv layers, i.e., after C1-Ci. Note that N16 is the same as All-BN. From Fig. 4, we note that it is possible to achieve All-BN level accuracy with N13. By using three fewer BN layers, N13 reduces the number of computations that need to be performed on the GPUs without significant accuracy loss. Note that N13 experiences a small accuracy drop of ∼1% compared to All-BN. However, both N10 and N4 fail to achieve acceptable accuracy. Interestingly, if we rearrange the placement of BN layers in N4 and add them after C5, C8, C11, and C16 layers of VGG-19 instead (referred as N4-R in Fig. 4), the trained model achieves All-BN level accuracy. Using four BN layers (instead of 16 in All-BN) introduces significant performance and energy benefits (shown later). Hence, having BN after each Conv layer (All-BN) is not necessary.

It is well known that Conv layer computations contribute to more than 80% of the execution time on GPUs [22]. Hence, energy and execution time savings by reducing the number of BN layers is relatively inconsequential for GPUs. However, ReRAM-based architectures that implement CNN training in a pipelined fashion can benefit from this methodology. It is well known that ReRAM crossbars can significantly speedup the MAC-dominated Conv layers. However, the amount of speedup will be limited if the other layers such as BN cannot keep up with it. Note that the slowest layer in a pipeline dominates the overall execution time. Fig. 5 illustrates a pipelined CNN training implementation using four layers (2 Conv and 2 BN layers) as an example. As shown in Fig. 5, the inputs (images belonging to the same batch) arrive sequentially and at any point of time (after the pipeline is full), e.g., at time 4T, processing elements responsible for all the CNN layers (ReRAMs and GPUs in this case) are active. Note that the conventional BN layer can be modified following [23] to work in a pipelined fashion (Fig. 5) without accuracy loss. As mentioned earlier, the overall performance will be dominated by the slowest layer. Therefore, the computations of each layer (Conv/BN) should be ideally accomplished in such a way that the time to execute different layers are similar, i.e., $T_C \approx T_N$; where T_C and T_N represent the time to execute the operations in a Conv and BN layer, respectively. However, executing too many BN layers (as in All-BN) on few GPUs can increase T_N disproportionately and slow down the entire pipeline (shown later). Hence, accelerating the Conv/FC layers on ReRAMs to improve T_C further will not be useful anymore. This can limit the achievable speedup in execution time using *DeepTrain*.

To address such a situation, we can increase the number of GPU tiles/layers in *DeepTrain* to reduce T_N and make the pipeline more balanced. However, the total number of logic layers that can be stacked vertically is often limited due to thermal limitations. For instance, Joardar *et al.* [3] have shown that up to four layers can be stacked before crossing 100 °C.

As a result, the amount of ReRAM and GPU tiles that can be incorporated in *DeepTrain* is limited. We cannot increase the number of GPU layers without sacrificing some of the ReRAM layers. As shown in [1], having more ReRAMs is beneficial as it enables us to parallelize the slower CNN layers leading to higher speedup. Hence, sacrificing ReRAM layers to add more GPUs is not an attractive option as it will reduce the amount of speedup that can be achieved by the proposed architecture. In addition, executing redundant BN operations on GPUs will result in needless energy consumption.

Therefore, All-BN is not optimal for high performance training on *DeepTrain*. Instead, we should remove some of the redundant BN layers to reduce the computation on GPUs. This would automatically reduce T_N (as the amount of work per GPU tile is lower) and prevent BN layers from creating a performance bottleneck. However, having too few BN layers can potentially lead to No-BN like behavior (i.e., poor accuracy as shown in Fig. 2). Therefore, our goal is to find the minimum number of BN operations necessary for a given CNN to achieve both high performance and All-BN level of accuracy. However, Fig. 4 clearly shows that both the number and location of BN layers affect the accuracy of the trained model. To the best of our knowledge, there exists no analytical model or mathematical formulation that can predict the final accuracy as a function of these two parameters. Hence, we need an algorithmic formulation to address this problem (discussed in the next section). Overall, this results in a symbiotic relationship between the DeepTrain architecture and the BO-based methodology. The DeepTrain architecture provides support for BN layers to improve the accuracy of the trained model but incurs more time and area overhead due to too many BN operations. The BO algorithm reduces the number of BN operations necessary for training leading to faster training, with fewer associated hardware resources and lower energy.

Here, it should be noted that the underlying 3-D architecture with All-BN is the same as the AccuReD architecture [3]. However, the major difference between DeepTrain and AccuReD is the overall configuration enabled by BO-N. As discussed earlier, only a finite number of planar layers can be stacked in a 3-D configuration before the temperature becomes unmanageable. AccuReD uses the All-BN configuration, which necessitates more GPU layers. Reducing the number of GPUs in AccuReD will result in more BN operations on fewer GPUs leading to a higher time to execute the BN layers (and hence a higher execution time). As a result, AccuReD cannot use systems with more ReRAM layers. On the other hand, the DeepTrain architecture uses the BO-N configuration. This reduces the number of operations that need to be executed on GPUs. Hence, we can reduce some GPU layers and add more ReRAM layers without affecting performance. As we show later via experiments, DeepTrain achieves significantly better performance compared to AccuReD.

B. Sparsity-Aware BO

In this section, we present the proposed sparsity-aware BO algorithm that can quickly determine the most effective BN layers in a given CNN.

Problem Definition: Given a deep CNN with C Conv layers, we can add a BN layer after any of these C layers. Therefore, the *combinatorial space* for candidate placements of BN layers is of size 2^C (exponential in the number of Conv layers). Evaluating each candidate choice is *computationally expensive* and involves training the CNN with the fixed placement of BN layers to compute the overall accuracy. It is well known that CNN training is time consuming. As a result, an exhaustive evaluation of all the 2^C ways to add BN layers in a CNN is not feasible, particularly for deep CNNs where C is large. For instance, we have 2^{16} choices of using BN in VGG-19 ($C = 16$). Hence, an exhaustive exploration is practically impossible. Therefore, we employ the BO framework and adapt it as needed to solve this problem in a computationally efficient manner, i.e., find a good solution in fewer iterations. Overall, our goal is to find a solution from the combinatorial space that exhibits the following properties: 1) uses the minimum number of BN layers (i.e., sparse solutions in terms of the presence/absence of a BN layer after each Conv layer) and 2) achieves similar accuracy as All-BN configuration. As we show later, this leads to better *DeepTrain* performance. This problem is an instance of the general problem of optimizing combinatorial spaces when the objective function is expensive to evaluate [28], which is relatively understudied when compared to its counterpart for continuous spaces.

Problem Formulation: We can formulate the above problem as *sparsity-aware BO over combinatorial spaces*. Let $D = \{0, 1\}^C$ denote the combinatorial space over a vector $x = (x_1, x_2, \ldots, x_c)$ of C binary variables, where x_i stands for the presence or absence of a BN layer after the ith convolution layer and C represents the number of Conv layers in the given CNN. Let $F(x)$ denote the evaluation score (prediction accuracy) of a candidate BN placement vector x. In this case, the black box evaluation function F involves training the CNN (with BN layers added following x) and is *expensive* (CNN training is a time-consuming task). Our goal is to find $x^* \in D$ quickly (within few iterations), to reduce the number of times the expensive evaluation function F is invoked. The optimal solution $x^* \in D$ should maximize the evaluation score $F(x)$ (accuracy) and must be sparse in terms of the L_1 norm ($|x|_1$), i.e., fewer BN layers should be used.

BO is an effective framework to solve black-box optimization problems with expensive evaluation functions [13]. There are three key elements in the BO framework.

1) The statistical model (SM) of the true function $F(x)$ that imposes a prior over the space of functions. This model should be able to make predictions for unknown inputs and also quantify its uncertainty. Bayesian models are typically used due to their flexibility and uncertainty quantification ability.
2) The acquisition function (AF) to score the utility of evaluating a candidate input $x \in D$ based on the SM. AF needs to tradeoff exploitation (selecting inputs with high prediction value) and exploration (selecting inputs with high uncertainty) using the learned SM.
3) AF optimization (AFO) to select the best scoring candidate input x_{next} according to AF depending on the SM. Next, we discuss few key aspects of this algorithm.

1) Second-Order Polynomial for SM: The choice of the SM will impact the effectiveness and computational complexity of the BO solution: complex models are expressive, but will require more training data to be useful (i.e., less effective for BO) and result in high computational complexity to solve the AFO problem for selecting inputs for evaluation. In this work, we employ a second-order Bayesian linear model as the surrogate function ($f_\theta(x)$) [14]. A sparsity inducing Horseshoe prior on the parameters along with an L1 norm-based penalty is added to the objective to capture/prioritize the condition that we require sparse solutions (also known as fewer BNs). The second-order model provides a good tradeoff between expressivity and accuracy. The model is described below with θ as the *learnable* model parameters: (linear terms) θ_i is the weight of x_i and (quadratic terms) θ_{ij} is the weight of $x_i.x_j$, where x_i and x_j are binary variables belonging to the vector x, i.e., $x = (x_1 x_2, \ldots, x_C)$

$$f_\theta(x) = \theta_0 + \sum_{i=1}^{C} \theta_i x_i + \sum_{i=1}^{C} \sum_{j=i}^{C} \theta_{ij} x_i.x_j - \lambda |x|_1. \quad (1)$$

The model parameters capture the uncertainty in the model and help in guiding the search-space exploration. Here, λ is a user-defined parameter that provides additional control over the sparsity. A larger value of λ favors sparser solutions due to lower L_1norm and is used in this work. Note that the hyperparameters used in Bayesian models are usually estimated by maximizing the marginal likelihood. However, our specific problem: sparse Bayesian linear regressor learning, is much simpler in this regard as the sparsity parameter has a well-defined semantics (i.e., the resulting solutions do not vary for small changes in the value of lambda). Hence, we performed a simple grid search over few coarse values of lambda to determine the most suitable one.

2) Thompson Sampling as AF: Any AF is defined as the expectation of a utility function under the posterior predictive distribution $P(y \mid x, D) = \int P(y \mid \theta) * P(\theta | x,\ D)$ [32], [33], where x is the input, y is the output of evaluating objective function at input x, and D is the training set of input–output pairs. In this work, we use Thompson sampling as the AF. Thompson sampling approximates this posterior by a single sample $\theta^* \sim P(\theta | x,\ d)$ which inherently enforces exploration due to the variance of this Monte-Carlo approximation and exploitation is the maximization with respect to the selected sample [32]–[34]. However, approximating the posterior using a single sample inherently enforces exploration due to the variance of this approximation. We optimize the sampled function to select the next candidate input for the expensive function evaluation (intractable combinatorial optimization in general). In our approach, the surrogate model is a Bayesian linear regressor. Hence, we sample the posterior parameters θ and we need to select the input that maximizes the inexpensive objective $f_\theta(x)$ for the next evaluation. This problem is formulated as a binary quadratic program (BQP) and solved using semidefinite program (SDP) solvers.

Here, it should be noted that other AFs, such as expected improvement (EI), upper confidence bound (UCB), and probability of improvement (PI), have also been used in prior works [13]. However, we choose Thompson sampling in this work for the following reasons: 1) Thompson sampling is a parameter-free AF unlike UCB, EI, and PI. In UCB, we need to set the tradeoff between exploration (variance) and exploitation (mean) in each BO iteration. In EI and PI, we need to set the threshold parameter and 2) in our problem setting, the input space is discrete/combinatorial, which is much more challenging than continuous spaces studied in most of the existing BO research. The key challenge is to find the next binary structure for evaluation from this large combinatorial space (2^d, where d is the number of binary variables). In its general form, this is an NP-Hard combinatorial optimization problem. With standard AFs, such as UCB, EI, and PI, we cannot get a tractable optimization solver to address this challenge. Thompson sampling allows us to reduce this problem to a BQP, which can be solved using state-of-the-art semidefinite programming solvers.

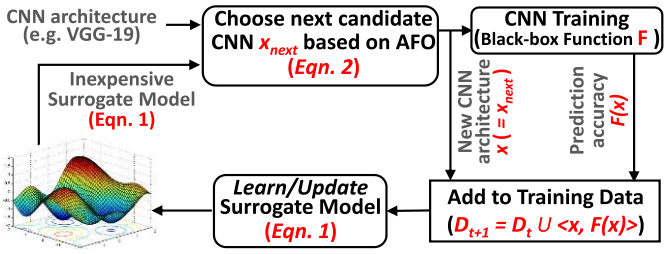

Fig. 6. Illustration of the proposed BO algorithm.

3) Semidefinite Program for AFO: Efficient and accurate AF optimization is a critical element for fast convergence of the BO procedure. When Thompson sampling is utilized as the AF, the AFO problem becomes a BQP as stated in

$$\min_{x \in \{0,1\}^C} \{x' A x + b' x\} \quad (2)$$

where x' and b' denote the transpose of vectors x and b, respectively, A denotes the matrix with all second-order parameters, i.e., $A_{ij} = \theta_{ij}$ and b denotes the vector with first-order parameters, i.e., $b_i = \theta_i + \lambda$ where θ is sampled from the posterior of the SM. This allows us to use a semidefinite programming (SDP)-based approach for solving the BQP.

Overall, Fig. 6 shows a high-level illustration of the BO algorithm. We first initialize the SM using a few random samples, i.e., $<x, F(x)>$ pairs where x indicates where the BN layers are added in a CNN and $F(x)$ represents the accuracy achieved using this configuration. Next, in each iteration, the BO algorithm invokes the AFO to select the next candidate BN layer configuration x_{next} following (2) (as shown in Fig. 6). This new configuration $x(= x_{\text{next}})$ is then evaluated using the expensive black box function F to compute the accuracy ($F(x)$) after training the CNN. Note that the BO algorithm reduces the number of times F needs to be invoked. The $<x, F(x)>$ pair is then added to the training data (D_t) already available from past function evaluations (Fig. 6). The new training data is then used to update/improve the *SM* for the next iteration. At the end of MAX iterations, the algorithm returns the best BN layer configuration $x(= x_{\text{best}})$ uncovered during the search process.

TABLE I
RELEVANT PARAMETERS FOR THE DEEPTRAIN ARCHITECTURE

Each GPU layer: 16-GPU tiles	
GPUs	Fermi architecture, 700 MHz, Private 64kB L1
Each ReRAM layer: 36-clusters, 4-tiles per cluster	
ReRAM tile	128x96-DACs (1-bit), 96-crossbars, 128x128 crossbar size, 330 mW power, 10MHz [2]

Each iteration of the BO algorithm involves solving the AFO problem (time complexity is polynomial in C) and training the CNN to get accuracy (time complexity is linear in the number of epochs and number of parameters) and updating the Bayesian linear regressor using each new training example (time complexity is quadratic in C). In practice, C is a small number (between 10 and 100) making our overall solution scalable. In our experiments, we observed that the BO algorithm converges in less than 20 iterations always to find near-optimal solutions from the combinatorial space of size 2^C. Here, it should be noted that the BO-based CNN optimization is dataset-agnostic. Our experiments indicate that CNNs optimized for one dataset can also be used for training using other datasets without any significant loss of accuracy. Hence, this optimization is a one-time process for any given CNN, which can then be used repeatedly; thereby amortizing the optimization cost (time and energy spent).

VI. EXPERIMENTAL RESULTS

In this section, we first discuss the experimental setup to evaluate the performance of the *DeepTrain* architecture. Next, we present a performance analysis to motivate the need for fewer BN layers. Finally, we present a performance comparison of *DeepTrain* with respect to a conventional GPU-based implementation.

A. Experimental Setup

We employ GPGPU-Sim [25] to simulate the GPU layers to obtain execution time and power information for BN layers. The GPU cores are based on the NVIDIA Fermi architecture. Here, note that newer GPU architectures (e.g., Volta) can also be used. However, to the best of our knowledge, detailed power models for newer architectures are not available in GPGPU-Sim (hence, we use the Fermi architecture). The ReRAM crossbar and tile configurations/area are based on the information available in [2]. CNN execution on ReRAM tiles exhibits deterministic behavior without any run-time dependencies. Moreover, ReRAM arrays execute instructions in order and instruction latencies are deterministic. Therefore, deterministic execution models suffice to reliably capture ReRAM performance parameters, e.g., execution time, on-chip traffic, etc. In this work, we use the ReRAM execution models from [2]. We omit the discussion of these models in this article for the sake of brevity. The ReRAM execution model and GPGPU-Sim are used together to simulate the entire CNN. Some of the important parameters for the *DeepTrain* architecture are listed in Table I.

The vertical connections in *DeepTrain* are established using TSVs. We assume a four-layer 3-D architecture in this work for simplicity. However, note that the proposed methodology is not limited to any specific 3-D technology or a particular choice of the number of planar layers. Furthermore, in this work, we assume that GPUs and ReRAMs cannot be fabricated on the same layer due to manufacturing incompatibilities (e.g., different process technology nodes for GPU and ReRAM). Hence, to implement both Conv/FC and BN layers, we need to have at least one ReRAM and one GPU layer in *DeepTrain*. Therefore, there can be three possible *DeepTrain* configurations: 1) 1R3G; 2) 2R2G; and 3) 3R1G. Here, *MRNG* denotes M layers of ReRAM tiles and N layers of GPU tiles stacked vertically (Fig. 3). However, our analysis indicates that a single layer of ReRAMs in *DeepTrain* (1R3G) does not have enough ReRAM crossbars to store all weights and intermediate data on-chip, particularly for larger CNNs, e.g., VGG-19. Hence, in this work, we explore the 2R2G and 3R1G configurations only.

Here, it should be noted that the 2R2G configuration with All-BN CNNs is essentially the same as the AccuReD architecture [3]. Hence, we use AccuReD as a baseline to evaluate the efficacy of the proposed methodology. Prior solutions, such as REGENT [16] and 3D-ReG [17], have also proposed similar architectures that use 3-D integration to incorporate both ReRAMs and GPUs for CNN training. However, these architectures are fundamentally different in terms of how the CNN layers are executed and the incorporation of BN layers. In DeepTrain, we use the ReRAMs to perform all the MAC operations (both forward and backward phases) while BN is implemented using the GPUs. However, REGENT and 3D-ReG use ReRAMs for the forward phase only while performing the backward phase computations on GPUs. As mentioned earlier, BN layers (both forward and backward phases) require high-precision support [5], [6]. Due to their mapping policy, both REGENT and 3D-ReG cannot support BN operations during the forward phase. Moreover, executing the MAC-heavy backward phase on relatively slower GPUs will bottleneck performance and any gains made by accelerating the forward phase on ReRAMs will be reduced. Due to these two reasons: 1) lack of BN support (during the forward phase) and 2) slower pipeline due to involvement of GPUs in the backward phase, we do not consider REGENT and 3D-ReG as appropriate baselines in this article.

The CNNs were implemented using PyTorch (Python) and trained on an NVIDIA Titan Xp GPU with 12 GB of memory for 50 epochs. The BO algorithm is also implemented using Python to easily integrate with the PyTorch-based CNN implementations. We run the BO algorithm for up to 100 iterations for all the CNNs and found that it always converges in less than 50 iterations.

B. Impact of All-BN on Performance

Fig. 7 shows the execution time (normalized) of the slowest Conv/FC layer on ReRAMs and the slowest BN layer executed on GPUs for all possible *DeepTrain* architectures with All-BN CNNs. As mentioned earlier, the CNN training on ReRAMs

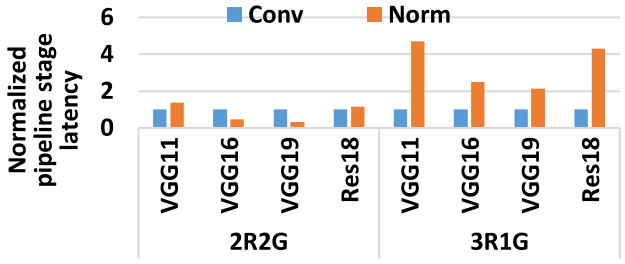

Fig. 7. Worst pipeline stage latency on GPUs (*BN*) and ReRAMs (*Conv/FC*) when All-BN CNNs are executed on different *DeepTrain* configurations (normalized with respect to *Conv/FC* in each case).

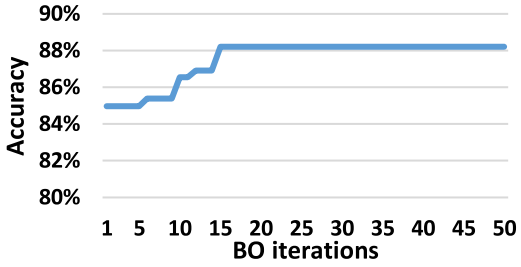

Fig. 8. Best model accuracy achieved after different BO iterations.

is implemented in a pipelined manner (Fig. 5), where each CNN layer constitutes a stage of the execution pipeline [1]. In other words, Conv/FC in Fig. 7 represents $\max(T_{Ci})$ while BN represents $\max(T_{Nj})$. Here, T_{Ci} and T_{Nj} represent the time to execute the ith Conv/FC layer on ReRAMs and the jth BN layer on GPUs, respectively (Fig. 5). The overall execution time (the parameter T in Fig. 5) depends on the slowest stage, i.e., $T = \max\{T_{Ci}, T_{Nj}\}$. As an example, for training VGG-11 on the 2R2G configuration (in Fig. 7), the BN layer execution on GPU requires slightly more time than the Conv/FC computation on ReRAMs. Hence, due to the pipeline, even if the computations associated with the Conv/FC layers are finished early, the ReRAMs must remain idle and wait for the GPUs to finish the computation. Therefore, the overall runtime will be determined by the time to execute BN in this case.

From Fig. 7, we note that the worst-case times needed to execute Conv/FC layers on the ReRAMs and BN layers on GPUs are relatively balanced in 2R2G. Adding more ReRAMs (as in 3R1G) accelerates the Conv/FC layers even further. As mentioned earlier, having more ReRAMs (as in 3R1G) is desirable as it enables us to accelerate the slower CNN layers by replicating the weights on the extra ReRAMs following [1]. However, as Fig. 7 clearly shows, the time to execute the BN layers skyrockets in this case. This happens because GPUs are limited in 3R1G (compared to 2R2G) and the amount of computation per GPU tile for All-BN CNNs is significantly higher. Note that the total number of layers that can be stacked vertically is limited due to thermal concerns [3]. As a result, due to the pipelined implementation of CNN training, the overall execution time will be dominated by the BN layers and any gains made by accelerating the Conv/FC layers on ReRAMs is not useful. This prevents *DeepTrain* from reaching its maximum achievable performance and thereby realizing its full potential.

Overall, there is no benefit in simply adding more ReRAMs when we use an All-BN configuration; the BN layers tend to limit any performance benefits eventually. To address this problem and achieve better performance, we should reduce the number of BN operations necessary to train the CNN without losing accuracy. As we have shown in Fig. 4, by using the BN layers judiciously, it is possible to achieve high accuracy with significantly fewer BN operations than an All-BN configuration. For instance, if we place only four BN layers after C5, C8, C11, and C16 layers of VGG-19 (referred as N4-R in Fig. 4), we can achieve All-BN (16 BN layers) level accuracy. This reduces the amount of computation necessary, which can lead to better performance as we show next.

C. Sparsity-Aware BO Algorithm

First, we present the efficacy of the proposed sparsity-aware BO algorithm in quickly finding a CNN configuration with few BN layers (referred as BO-N hereafter). Fig. 8 shows the quality of the best BO-N configuration uncovered (in terms of the accuracy of trained models) with respect to the number of BO iterations, for ResNet-18 using the CIFAR-10 dataset. Note that ResNet-18 has the highest number of BN layers among the four CNNs considered in this work. Hence, the search space of possible solutions is the largest here. We run the BO algorithm for 50 iterations with different random seeds and different initialization data for a rigorous analysis. As shown in Fig. 8, the sparsity-aware BO algorithm found the best solution (BO-N configuration with similar training accuracy as All-BN) in significantly fewer iterations (15 iterations in Fig. 8). Among all our experiments, the BO algorithm found the best solution within at most 20 iterations. In many cases, it found the best solution in less than ten iterations as well. Similar behavior was observed for the other CNNs considered in this work. Fig. 8 clearly shows that the proposed sparsity-aware BO algorithm converges very fast in practice.

Each experiment with BO often generates more than one valid solution. A solution is deemed valid if and only if it has an accuracy drop of less than 1% compared to All-BN. Hence, the output of BO is a set of such valid solutions D. Each solution $d \epsilon D$ is part of a Pareto set and has a different accuracy–performance–energy tradeoff. We choose the optimal solution $d^* \epsilon D$ that results in the best performance and energy consumption (measured using GPGPU-Sim simulations). In this work, we prioritize performance and energy while deciding d^*. However, note that $d^* \epsilon D$ is a valid solution, i.e., its accuracy is guaranteed to be within 1% of All-BN. Here, our assumption is that less than 1% accuracy loss is insignificant for most applications in exchange for significant performance and energy gains compared to All-BN as we show next. Other solutions from the Pareto set can also be chosen if accuracy is prioritized. However, such a solution can have relatively inferior performance and energy tradeoff compared to the d^* considered here.

D. All-BN Versus BO-N

Fig. 9(a) compares the prediction accuracy of the CNN obtained using the BO algorithm discussed in Section V and

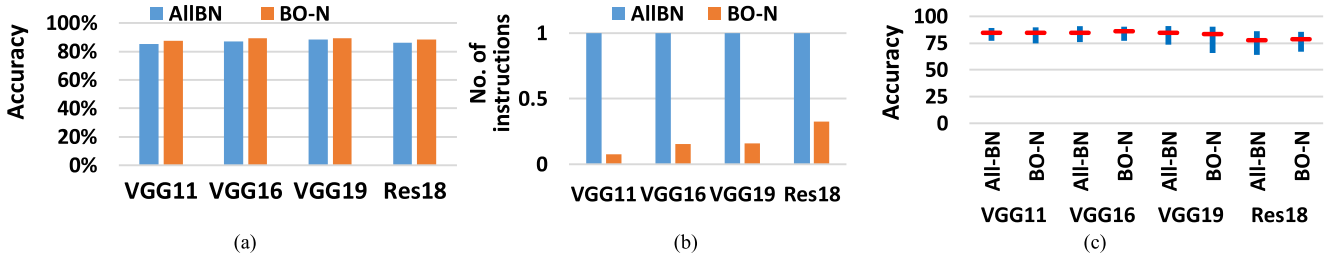

Fig. 9. Comparing (a) prediction accuracy, (b) number of resulting instructions executed on GPUs, for All-BN and the CNN configuration obtained using the BO algorithm (BO-N) discussed in Section V, and (c) accuracy of All-BN and BO-N CNNs trained with various hyperparameter settings.

the All-BN configuration. From Fig. 9(a), we note that the BO-N CNNs achieve All-BN level accuracy for all the four CNNs considered in this work. However, the BO-N configurations use only a small number of BN layers (and operations) compared to All-BN. The BO-N configuration for VGG-11, VGG-16, VGG-19, and ResNet-18 uses two (after C5 and C7), four (after C5, C7, C8, and C13), four (after C5, C8, C11, and C16), and 12 (after C3, C6, C7, C9, C10, C11, C12, C13, C14, C15, C16, and C17) BN layers, respectively, as opposed to 8, 13, 16, and 20 layers in All-BN (placed after each Conv layer). For all the four CNNs, the BO methodology identifies the layers closer to the output (softmax) layer as the most effective locations for using BN. This is understandable as the prediction accuracy of trained CNNs depends on the complex features closer to the output layer, which carry significant information to discriminate among different class labels (e.g., cats *versus* dogs) [30]. BN layers allow us to preserve these important features during training.

However, as different BN layers operate on varying sized inputs, the number of BN layers is not a good measure of the efficacy of the proposed methodology. Hence, we present the total number of instructions (including integer, floating-point, and memory read/write instructions as obtained from GPGPU-Sim) that were executed on the GPUs for both the All-BN and BO-N variants in Fig. 9(b). As Fig. 9(b) shows, the number of instructions executed by the GPUs reduced by 82% on average for all the CNNs considered in this work for BO-N compared to All-BN. This happens as most of the BN layers in BO-N are concentrated closer to the last (softmax) layer where the outputs are relatively smaller. Typically, inputs are downsized as it moves from input to output side. As a result, even when the BO-N configuration uses relatively more BN layers as in ResNet-18, the reduction of GPU workload is significant. Overall, BO-N results in faster execution and energy savings for CNN training using the proposed architecture.

Interestingly, BO-N also retains All-BN's robustness to the choice of hyperparameters despite having fewer BN layers. Fig. 9(c) shows the accuracy observed for various hyperparameter settings. For this experiment, we use the same 150 hyperparameter combinations as in Fig. 2. From Fig. 9(c), we can clearly see that BO-N is as robust as All-BN to the choice of hyperparameters. The average accuracy for BO-N exhibits less than 1% difference than that of the average accuracy of All-BN despite having few BN layers. Overall, from Fig. 9, it is clear that BN layers are necessary but only a few are sufficient. This makes the CNNs lightweight, which

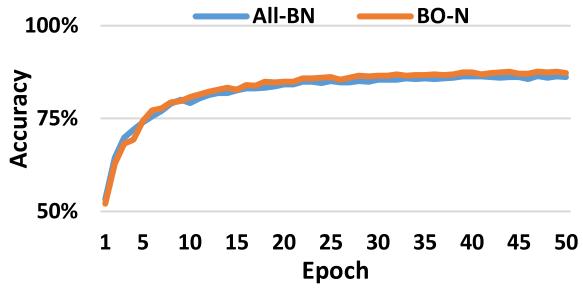

Fig. 10. Accuracy after different epochs of training ResNet-18 using the CIFAR-10 dataset. The BO-N configuration lags All-BN in terms of model accuracy for the first few epochs, beyond which the difference is negligible.

enables high-performance training of CNNs on *DeepTrain* as we show next. Here, it should be noted that we have verified the correctness of the BO algorithm by comparing the solution quality (prediction accuracy) with that of an exhaustive exploration whenever possible. Our analysis indicates that the BO algorithm can find near-optimal CNN configurations (accuracy difference of $\sim$1% compared to the best solution found by exhaustive exploration).

Next, we compare the number of epochs required by both All-BN and BO-N to reach convergence. Fig. 10 shows the model accuracy after each epoch of training ResNet-18 using the CIFAR-10 dataset as an example. Other CNNs considered in this work exhibit similar behavior. As shown in Fig. 10, the BO-N configuration lags the accuracy of All-BN in the first few epochs (5–10 epochs in most cases). However, the accuracy difference is negligible during the latter part of the training, i.e., both configurations converge at approximately the same time. This shows that BO-N is not only robust to the choice of hyperparameters [as shown in Fig. 9(c)] but is also as fast as All-BN in terms of speed of convergence.

E. Impact of BO-N on Performance

Next, Fig. 11 compares the impact of training the BO-N configurations on *DeepTrain* (similar to Fig. 7). By comparing Fig. 11 with Fig. 7, we can clearly see that the BN layers are no longer performance bottlenecks for either 2R2G or 3R1G. This happens because BO-N uses only a handful of BN layers. Hence, the amount of work per GPU tile is very little and can be handled by only a limited number of GPUs. The optimization procedure enables us to use more ReRAM layers without BN creating a bottleneck which is not possible otherwise.

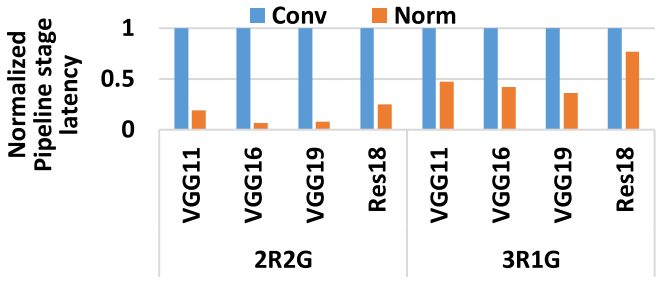

Fig. 11. Worst pipeline stage latency on GPUs (*BN*) and ReRAMs (*Conv/FC*) when BO-N CNNs are executed on different *DeepTrain* configurations (normalized with respect to *Conv/FC* in each case).

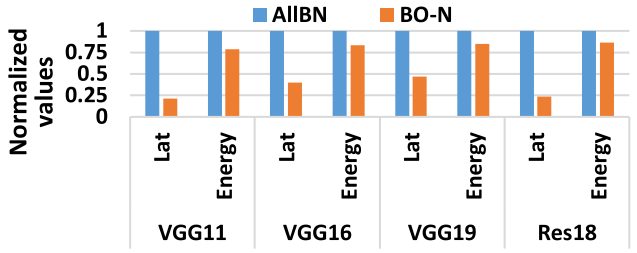

Fig. 12. Pipeline stage latency (Lat) and energy dissipated (normalized) when All-BN and BO-N CNNs are executed on the 3R1G *DeepTrain* architecture.

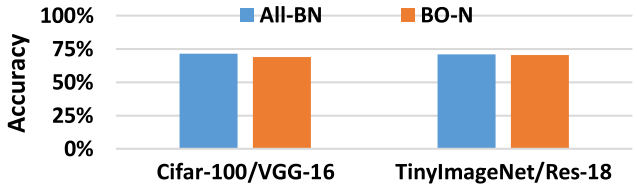

Fig. 13. Prediction accuracy achieved by BO-N and All-BN configurations with other datasets (CIFAR-100 and TinyImageNet).

Fig. 12 shows the normalized pipeline stage latency (Lat) and the corresponding energy dissipation (Energy) for executing the All-BN and BO-N CNNs. (Note that the normalization here is with respect to the performance parameters and it should not be confused with BN layer of the CNN.) As we can see from Fig. 12, the BO-N CNNs enable *DeepTrain* to achieve 67.1% lower execution time while consuming 16.6% less energy on average. This happens as too many BN layers in All-BN create a performance bottleneck and prevent 3R1G from achieving its full potential (as shown in Figs. 7 and 12). In addition, All-BN involves 82% more operations that need to be executed on GPUs [Fig. 9(b)] leading to the high energy consumption. Overall, the CNN configurations obtained using the BO algorithm discussed in Section V enable the *DeepTrain* architecture to achieve even more speedup than the traditional All-BN configuration for training deep CNNs without losing accuracy.

F. Transferability of BO-N to Other Datasets

So far, we executed the BO methodology to find the CNN configuration with fewer BN layers using the CIFAR-10 dataset. However, the same CNNs (obtained using BO with the CIFAR-10 dataset) can also be used with other datasets as we show next. Fig. 13 shows the prediction accuracy (*Top-1 accuracy*) achieved by All-BN and BO-N models for CIFAR-100 [26] (32×32, RGB, 100 classes, 50 000 training, and 10 000 testing) and TinyImageNet [27] (64×64, RGB, 200 classes, 100 000 training, and 10 000 testing) datasets with VGG-16 and ResNet-18, respectively. Both CIFAR-100 and TinyImageNet datasets are larger and more complex than the CIFAR-10 dataset used so far. The dataset/CNN combinations in Fig. 13 were chosen as an example only. Please note that other datasets (such as ImageNet) and CNNs can also be used here. However, training repeatedly on the entire ImageNet dataset from scratch is prohibitively expensive. Hence, we have used relatively smaller datasets in this work. From Fig. 13, we note that with both the datasets, the BO-N configurations achieve comparable accuracy less than 1% drop on average) as their All-BN counterparts for two different CNNs. This is similar to our previous observations with CIFAR-10 in Fig. 9(a).

Here, it should be noted that the BO-N configurations used in Fig. 13, are the same as those obtained using CIFAR-10, i.e., the BO process need not be repeated for the other datasets. This happens as CNN layers closer to the output layer carry important information that can discriminate among different class labels for any dataset [30]. The BO algorithm learns this knowledge from one dataset, which can then be applied to other datasets as well. The overall idea is similar to the concept of domain adaptation [29] in the machine learning literature, where knowledge learned from one domain can be reused in other domains. Fig. 13 provides strong empirical evidence that the observations from Fig. 9(a) are not specific to one dataset only. This property can be used to optimize deeper CNNs for larger datasets inexpensively. For instance, the BO-N configuration of larger ResNet models can be quickly obtained using CIFAR-10. Due to the small size of CIFAR-10, the time necessary for repeated training involved during BO (Fig. 6) is relatively small. The BO-optimized CNN configuration can then be reused for other larger datasets without having to repeat the optimization process again (which can be time consuming for large datasets such as ImageNet); thereby amortizing the optimization cost.

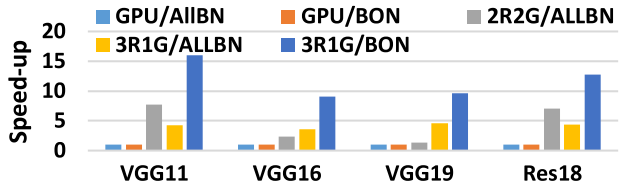

Fig. 14. Full-system execution time speedup achieved by the 3R1G *DeepTrain* architecture compared to GPU. Note that the 2R2G/AllBN configuration is equivalent to the AccuReD architecture [3].

G. Full-System Comparison

Finally, we compare the full-system execution time speedup achieved by *DeepTrain* compared to a conventional GPU-based platform (Nvidia Titan Xp). Here, we consider both BO-N and All-BN CNNs. Fig. 14 shows the full system execution time speedup achieved by the different configurations

compared to the conventional GPU (referred as GPU/AllBN in Fig. 14). Note that 2R2G/AllBN in Fig. 14 is the same as the AccuReD architecture [3]. To the best of our knowledge, AccuReD represents the state of the art in ReRAM-based architectures for training deep CNNs with BN support. As we can see from Fig. 14, all the *DeepTrain* configurations achieve better execution time than a conventional GPU regardless of the architecture. This happens as the ReRAMs can perform MAC operations much faster than GPUs. However, as Fig. 14 shows, the amount of speedup that can be achieved by All-BN is limited. For all the four CNNs, the BO-N configurations with the 3R1G *DeepTrain* architecture achieve the best speedup among all the configurations that we considered here (including AccuReD, i.e., 2R2G/AllBN). This happens as more ReRAMs are available to accelerate the Conv/FC layers and the few BN layers used, do not pose a performance bottleneck. Overall, Fig. 14 shows that too many BN layers can prevent *DeepTrain* from reaching its full potential. Hence, we should use BO-N CNNs with the 3R1G *DeepTrain* architecture for training deep CNNs. Therefore, a key insight that this article provides is that "More is not always better" in the context of CNN. Interestingly, note that BO-N does not improve the training time for GPUs as shown in Fig. 14 (referred as GPU/BON in Fig. 14). This happens as convolution layers tend to be the most time-consuming layers when a CNN is trained using GPUs (more than 90% of overall training time) unlike in ReRAM-based systems. Note that ReRAMs can perform many MAC operations in $O(1)$ time [2]. As a result, reducing a few BN layers using BO-N is relatively inconsequential for GPUs.

VII. Conclusion

Deep CNNs suffer from the vanishing/exploding gradients problem which often results in unsuccessful training. Non-BN methods such as Xavier/Kaiming initializations for training deep CNNs are not reliable and require careful hyperparameter tuning. In this work, we have shown that while BN is important, only a few BN layers are sufficient for high-accuracy CNN training. We have proposed a BO algorithm that can easily find the optimal CNN configuration. These lightweight CNNs use few BN layers but achieve high accuracy and consume 16.6% lower energy than conventional All-BN CNNs on a 3-D ReRAM/GPU-based architecture. Overall, BO-N achieves up to 15× speedup compared to a GPU for training deep CNNs.

References

[1] L. Song, X. Qian, H. Li, and Y. Chen, "PipeLayer: A pipelined ReRAM based accelerator for deep learning," in *Proc. HPCA*, 2017, pp. 541–552.

[2] A. Shafiee *et al.*, "ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proc. SIGARCH ISCA*, 2016, pp. 14–26.

[3] B. K. Joardar, J. R. Doppa, P. P. Pande, H. Li, and K. Chakrabarty, "AccuReD: High accuracy training of CNNs on ReRAM/GPU heterogeneous 3D architecture," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 40, no. 5, pp. 971–984, May 2021.

[4] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating deep network training by reducing internal covariate shift," in *Proc. ICML*, 2015, pp. 448–456.

[5] P. Micikevicius *et al.*, "Mixed precision training," in *Proc. ICLR*, 2018, pp. 1–12.

[6] T. Na, J. H. Ko, J. Kung, and S. Mukhopadhyay, "On-chip training of recurrent neural networks with limited numerical precision," in *Proc. IJCNN*, 2017, pp. 3716–3723

[7] D. Das *et al.*, "Mixed precision training of convolutional neural networks using integer operations," 2018. [Online]. Available: arXiv: 1802.00930.

[8] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proc. AISTATS*, 2010, pp. 249–256.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on Imagenet classification," in *Proc. ICCV*, 2015, pp. 1026–1034.

[10] *PyTorch.* Accessed: May 20, 2020. [Online]. Available: https://github.com/pytorch

[11] J. Bjorck, C. Gomes, B. Selman, and K. Q. Weinberger, "Understanding batch normalization," in *Advances in Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates, 2018

[12] Z. Li and S. Arora, "An exponential learning rate schedule for deep learning," 2019. [Online]. Available: arXiv:1910.07454.

[13] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas, "Taking the human out of the loop: A review of Bayesian optimization," *Proc. IEEE*, vol. 104, no. 1, pp. 148–175, Jan. 2016.

[14] R. Baptista and M. Poloczek, "Bayesian optimization of combinatorial structures," in *Proc. ICML*, 2018, pp. 471–480.

[15] A. Ankit *et al.*, "PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference," in *Proc. ASPLOS*, 2019, pp. 725–731.

[16] B. K. Joardar, B. Li, J. R. Doppa, H. Li, P. P. Pande, and K. Chakrabarty, "REGENT: A heterogeneous ReRAM/GPU-based architecture enabled by NoC for training CNNs," in *Proc. DATE*, 2019, pp. 522–527.

[17] B. Li, J. R. Doppa, P. P. Pande, K. Chakrabarty, J. X. Qiu, and H. Li, "3D-ReG: A 3D ReRAM-based heterogeneous architecture for training deep neural networks," *J. Emerg. Technol. Comput. Syst.*, vol. 16, no. 2, p. 20, 2020

[18] F. Chen, L. Song, and Y. Chen, "ReGAN: A pipelined ReRAM-based accelerator for generative adversarial networks," in *Proc. ASPDAC*, 2018, pp. 178–183.

[19] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. ICLR*, 2015.

[20] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE CVPR*, 2016, pp. 770–778.

[21] M. M. S. Aly *et al.*, "Energy-efficient abundant-data computing: The N3XT 1,000x," *Computer*, vol. 48, no. 12, pp. 24–33, Dec. 2015.

[22] X. Li, G. Zhang, H. H. Huang, Z. Wang, and W. Zheng, "Performance analysis of GPU-based convolutional neural networks," in *Proc. ICPP*, Philadelphia, PA, USA, 2016, pp. 67–76.

[23] Q. Liao, K. Kawaguchi, and T. Poggio, "Streaming normalization: Towards simpler and more biologically-plausible normalizations for online and recurrent learning," 2016. [Online]. Available: arXiv:1610.06160.

[24] W. R. Thompson, "On the likelihood that one unknown probability exceeds another in view of the evidence of two samples," *Bull. Amer. Math. Soc.*, vol. 25, nos. 3–4, pp. 285–294, 1933.

[25] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proc. IEEE ISPASS*, Boston, MA, USA, 2009, pp. 163–174.

[26] A. Krizhevsky, "Learning multiple layers of features from tiny images," Dept. Comput. Sci., Univ. Toronto, Toronto, ON, Canada, Rep. TR-2009, 2009.

[27] *TinyImageNet Dataset*. Accessed: May 5, 2020. [Online]. Available: https://tiny-imagenet.herokuapp.com/

[28] A. Deshwal, S. Belakaria, J. R. Doppa, and A. Fern, "Optimizing discrete spaces via expensive evaluations: A learning to search framework," in *Proc. AAAI*, 2020, pp. 3773–3780.

[29] W. M. Kouw and M. Loog, "An introduction to domain adaptation and transfer learning," 2018. [Online]. Available: arXiv:1812.11806.

[30] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," 2013. [Online]. Available: arXiv:1311.2901.

[31] X. Wang, M. Palesi, M. Yang, Y. Jiang, M. C. Huang, and P. Liu, "Low latency and energy efficient multicasting schemes for 3D NoC-based SoCs," in *Proc. IEEE/IFIP 19th Int. Conf. VLSI System-on-Chip*, Hong Kong, China, 2011, pp. 337–342.

[32] D. Russo and B. V. Roy, "Learning to optimize via posterior sampling," *Math. Oper. Res.*, vol. 39, no. 4, pp. 1221–1243, 2014.

[33] J. M. H. Lobato, J. Requeima, E. O. P. Knapp, and A. A. Guzik, "Parallel and distributed Thompson sampling for large-scale accelerated exploration of chemical space," in *Proc. Int. Conf. Mach. Learn.*, 2017, pp. 1470–1479.
[34] K. Kandasamy, A. Krishnamurthy, J. Schneider, and B. Póczos, "Parallelised Bayesian optimisation via Thompson sampling," in *Proc. Int. Conf. Artif. Intell. Stat.*, 2018, pp. 133–142.

Biresh Kumar Joardar (Member, IEEE) received the Ph.D. degree from Washington State University, Pullman, WA, USA, in 2020.

He is currently a Postdoctoral Computing Innovation Fellow (CI-Fellow) with the Department of Electrical and Computer Engineering, Duke University, Durham, NC, USA. His current research interests include machine learning, manycore architectures, accelerators for deep learning, hardware reliability, and security.

Dr. Joardar received the Outstanding Graduate Student Researcher Award at Washington State University in 2019. His works have been nominated for Best Paper Awards at prestigious conferences, such as DATE and NOCS.

Aryan Deshwal (Student Member, IEEE) received the B.S. degree in mathematics and computing from Delhi Technological University, New Delhi, India, in 2017. He is currently pursuing the Ph.D. degree with the School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA, USA.

His current research focuses on developing fundamental machine learning algorithms with applications to electronic design automation and nano porous materials design.

Mr. Deshwal's M.S. dissertation on machine learning to optimize manycore systems design won the Outstanding Dissertation Award from Washington State University in 2020. He won Outstanding Reviewer Awards from ICML 2020, ICLR 2021, and ICML 2021 conferences.

Janardhan Rao Doppa (Member, IEEE) received the Ph.D. degree in computer science from the Oregon State University, Corvallis, OR, USA, in 2014.

He is currently a George and Joan Berry Chair Associate Professor with Washington State University (WSU), Pullman, WA, USA. His current research interests are at the intersection of machine learning and computing systems design.

Dr. Doppa received the NSF CAREER Award in 2019, the Outstanding Paper Award at the AAAI Conference in 2013, the Google Faculty Research Award in 2015, the Outstanding Innovation in Technology Award from Oregon State University in 2015, the Reid-Miller Teaching Excellence Award in 2018, and the Outstanding Junior Faculty in Research Award from the Voiland College of Engineering and Architecture at WSU in 2020. He is among the 15 outstanding young researchers selected to give Early Career Spotlight talk at the International Joint Conference on Artificial Intelligence in 2021.

Partha Pratim Pande (Fellow, IEEE) received the Ph.D. degree in electrical and computer engineering from the University of British Columbia, Vancouver, BC, Canada, in 2005.

He is a Professor and holder of the Boeing Centennial Chair of Computer Engineering with the School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA, USA, where he is currently the Director. His current research interests are novel interconnect architectures for manycore chips, on-chip wireless communication networks, and heterogeneous architectures.

Prof. Pande has won the NSF CAREER Award in 2009. He is the winner of the Anjan Bose Outstanding Researcher Award from the College of Engineering, Washington State University in 2013. He was the Technical Program Committee Chair of IEEE/ACM Network-on-Chip Symposium 2015 and CASES 2019–2020. He also serves on the program committees of many reputed international conferences. He currently serves as the Associate Editor-in-Chief for IEEE DESIGN AND TEST. He is on the editorial boards of IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION SYSTEMS, *ACM Journal of Emerging Technologies in Computing Systems*, and IEEE EMBEDDED SYSTEMS LETTERS.

Krishnendu Chakrabarty (Fellow, IEEE) received the B.Tech. degree from the Indian Institute of Technology Kharagpur, Kharagpur, India, in 1990, and the M.S.E. and Ph.D. degrees from the University of Michigan at Ann Arbor, Ann Arbor, MI, USA, in 1992 and 1995, respectively.

He is currently the John Cocke Distinguished Professor and the Chair of the Department of Electrical and Computer Engineering, Duke University, Durham, NC, USA, where he is a Professor of Computer Science. He is a Research Ambassador with the University of Bremen, Bremen, Germany. He was a Hans Fischer Senior Fellow with the Institute for Advanced Study, Technical University of Munich, Munich, Germany, from 2016 to 2019. His current research projects include: design-for-testability of integrated circuits and systems (especially 3-D integration and system-on-chip); AI accelerators; microfluidic biochips; hardware security; machine learning for healthcare; and neuromorphic computing systems.

Prof. Chakrabarty is a recipient of the National Science Foundation CAREER Award, the Office of Naval Research Young Investigator Award, the Humboldt Research Award from the Alexander von Humboldt Foundation, Germany, the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS Donald O. Pederson Best Paper Award, the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION SYSTEMS Prize Paper Award, the *ACM Transactions on Design Automation of Electronic Systems* Best Paper Award, multiple IBM Faculty Awards and HP Labs Open Innovation Research Awards, and over a dozen best paper awards at major conferences. He is also a recipient of the IEEE Computer Society Technical Achievement Award, the IEEE Circuits and Systems Society Charles A. Desoer Technical Achievement Award, the IEEE Circuits and Systems Society Vitold Belevitch Award, the Semiconductor Research Corporation Technical Excellence Award, and the IEEE Test Technology Technical Council Bob Madge Innovation Award. He is a 2018 recipient of the Japan Society for the Promotion of Science Invitational Fellowship in the "Short Term S: Nobel Prize Level" category. He was a Distinguished Visitor of the IEEE Computer Society from 2005 to 2007 and from 2010 to 2012, a Distinguished Lecturer of the IEEE Circuits and Systems Society from 2006 to 2007 and from 2012 to 2013, and an ACM Distinguished Speaker from 2008 to 2016. He served as the Editor-in-Chief for IEEE DESIGN & TEST OF COMPUTERS from 2010 to 2012, *ACM Journal on Emerging Technologies in Computing Systems* from 2010 to 2015, and IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION SYSTEMS from 2015 to 2018. He is a Fellow of ACM and AAAS, and a Golden Core Member of the IEEE Computer Society.