DyNNamic: Dynamically Reshaping, High Data-Reuse Accelerator for Compact DNNs

Edward Hanson, Shiyu Li, Xuehai Qian, Hai (Helen) Li, *Fellow, IEEE*, and Yiran Chen, *Fellow, IEEE*

Abstract—Convolutional layers dominate the computation and energy costs of Deep Neural Network (DNN) inference. Recent algorithmic works attempt to reduce these bottlenecks via compact DNN structures and model compression. Likewise, state-of-the-art accelerator designs leverage spatiotemporal characteristics of convolutional layers to reduce data movement overhead and improve throughput. Although both are independently effective at reducing latency and energy costs, combining these approaches does not guarantee cumulative improvements due to inefficient mapping. This inefficiency can be attributed to (1) inflexibility of underlying hardware and (2) inherent reduction of data-reuse opportunities of compact DNN structures. To address these issues, we propose a dynamically reshaping, high data-reuse PE array accelerator, namely *DyNNamic*. DyNNamic leverages kernel-wise filter decomposition to partition the convolution operation into two compact stages: Shared Kernels Convolution (SKC) and Weighted Accumulation (WA). Because both stages have vastly different dimensions, DyNNamic reshapes its PE array to effectively map the algorithm to the architecture. The architecture then exploits data-reuse opportunities created by the SKC stage, further reducing data movement with negligible overhead. We evaluate our approach on various representative networks and compare against state-of-the-art accelerators. On average, DyNNamic outperforms DianNao by $8.4\times$ and $12.3\times$ in terms of inference energy and latency, respectively.

Index Terms—Dataflow architectures, adaptable architectures, machine learning

1 Introduction

Deep Neural Networks (DNN) play an important role at the forefront of modern Machine Learning (ML) due to their potential for high accuracy, immense flexibility, and scalability. In an effort to improve latency and throughput of large models, great emphasis has been placed on compressing the models or implementing DNN hardware specialization. DNNs exhibit highly regular structures and substantial data-reuse opportunities, which are effectively exploited by accelerators incorporating arrays of processing elements (PE) with systolic dataflows. PEs are unit computation engines that typically perform multiply-and-accumulate (MAC) operations, which dominate DNN inference. Accelerators such as TPU [14] are notably efficient for such models due to their low-complexity processing engines and highly parallel arrays. Such low-complexity also improves scalability, enabling higher throughput with the addition of more PEs. However, simply adding more PEs to the accelerator may not proportionally increase throughput; this is caused by under-utilization of the resources due to inefficient mapping from the algorithm to the architecture.

One of the well-known reasons for the under-utilization of PE arrays is the introduction of *compact* and *sparse* DNN layers. Compact layers such as Xception's [6] depthwise-separable (DSC) layers and ResNet's [11] residual connections drastically restrict specific dimensions of DNN filters. Although restricting certain dimensions greatly reduces the number of computations in the layer, it also greatly affects data-reuse patterns, causing inefficient dataflow mappings onto PE arrays. Meanwhile, sparse models [10] require special compressed formats to obtain storage and bandwidth benefits. These compressed formats result in irregular access patterns or non-sequential data accesses, thus limiting parallel processing opportunities when naïvely implemented on general-purpose hardware. Consequently, DNN algorithms must consider their relevant hardware platforms to better translate theoretical computation reductions to real-time speedup.

Orthogonal to DNN algorithm improvements, other efforts have been made to improve the flexibility of underlying hardware [4], [5]. Such works aim to provide broad support to various model compacting schemes; however, continually increasing the flexibility of accelerators increases area, energy, and latency costs due to higher complexity. Allowing arbitrary data movements results in larger complexity overheads, which is contrary to the design principle of DNN accelerators. Instead, because DNNs innately possess highly regular structures and data access patterns, efficient PE array architectures should target specific data access patterns and remove predictably redundant operations wherever possible

The mapping from algorithm to architecture is especially important when considering supporting computation of *low-rank decomposed* filter structures while maintaining high levels of *data-reuse* within PE arrays. In order to efficiently

- *Edward Hanson, Shiyu Li, Hai (Helen) Li, and Yiran Chen are with Electrical and Computer Engineering, Duke University, Durham, NC 27708 USA. E-mail: {edward.t.hanson, shiyu.li, hai.li, yiran.chen}@duke.edu.*
- *Xuehai Qian is with the Ming Hsieh Department of Electrical Engineering, University of Southern California, Los Angeles, CA 90007 USA. E-mail: xuehai.qian@usc.edu.*

Manuscript received 15 Aug. 2021; revised 9 May 2022; accepted 6 June 2022. Date of publication 24 June 2022; date of current version 10 Feb. 2023.
This work was supported in part by NSF under Grants 1955246 and 1937435, and in part by ARO under Grant W911NF-19-2-0107.
(Corresponding author: Edward Hanson.)
Recommended for acceptance by P. Milder.
Digital Object Identifier no. 10.1109/TC.2022.3184272

leverage the reduced computation cost of low-rank decomposed structures, traditionally inflexible systolic arrays must be re-designed to be flexible. This would allow the hardware to support the varying dimensional characteristics of such structures with high utilization and energy efficiency.

In this paper, we propose *DyNNamic*, a dynamically reshaping, high data-reuse PE array accelerator for DNN inference. Using kernel-wise filter decomposition, DyNNamic partitions the convolution operation into two compact stages: Shared Kernels Convolution (SKC) and Weighted Accumulation (WA). Both stages are mapped onto a PE array using the output stationary (OS) dataflow. We then apply array-aware sparsity regularization and a custom compression scheme to the WA stage so that the architecture can fully leverage the model's sparsity. DyNNamic is able to efficiently support both of the vastly different dimensional constraints of SKC and WA through dynamically reshaping its PE array. The architecture then maximally exploits new data-reuse opportunities generated by the SKC stage through circular data-reuse paths. Main contributions of our work are:

- We propose to redesign the conventional PE array by enabling it to dynamically reshape its architecture. Our dynamic reshaping improves PE utilization with negligible controller overhead.
- We propose an efficient partitioned mapping scheme for kernel-wise decomposed convolutional layers. We adopt a hardware friendly convolutional layer compression framework and incorporate a custom compression scheme to explore fine-grained structured sparsity.
- We analyze enhanced data-reuse opportunities of the proposed mapping scheme and design data-reuse paths to drastically reduce on-chip memory accesses. These reuse paths are then leveraged to address heightened on-chip bandwidth demands of the dynamic PE array.

We evaluate our approach on representative networks, including VGGNet [26], ResNet [11], and MobileNetV2 [25], and compare our approach against relevant state-of-the-art accelerators. By directly computing the partitioned and compressed convolution operation, DyNNamic eliminates up to 97% of the weights, translating to improved inference energy and latency by $12.3\times$, respectively, compared to DianNao [3], which has no compression. Against SparTen [8], DyNNamic achieves $1.87\times$ improved energy efficiency with comparable inference latency by leveraging its reshaping PE array for improved on-chip data-reuse.

The paper is structured as follows. Background in relevant DNN computational footprint reduction methodologies (i.e., low-rank decomposition, compact filter structures, and pruning) and systolic-array acceleration are discussed in Section 2. Then, we discuss the core motivations of this work in Section 3, including challenges of improving data-reuse in systolic arrays and the potential for leveraging low-rank decomposition to improve runtime and energy efficiency. In Section 4, the approach to apply low-rank decomposition in a hardware-friendly manner is detailed. Then, we discuss DyNNamic's design and concepts for solving the data-reuse challenges of implementing low-rank decomposed convolutional layers onto a systolic array in Section 5.

TABLE 1
Important Terms and Quantities

Term	Description
h, w, c_l	Height, Width, and number of Channels for a feature map.
d, k	Kernel Dimension and number of basis Kernels.
arr_h, arr_w, $arr_{w,min}$	PE Array Height, Width, and Minimum Width.
IFM, IntFM, OFM	Input Feature Map, Intermediate Feature Map, and Output Feature Map.
SKC	Shared Kernels Convolution. c_l IFMs are convolved with k basis kernels to produce $c_l k$ OFMs.
WA	Weighted Accumulation. IntFMs are accumulated with c_{l+1} coefficient vectors to produce the final OFMs.
PE-SA	Processing Element Sub-Array. Multiple PE-SAs form a full PE array.
GLB	Global Buffer. On-chip memory accessible by all rows (or columns).
KR, AR	Kernel Reuse and Activation Reuse blocks.

Sections 6 and 7 evaluate our proposed co-designed framework. Lastly, Sections 8 and 9 contextualize related works and conclude this paper.

2 Background

2.1 Convolution Operation

To avoid ambiguity, we present the essential notations and terms that shall be used by the following discussion. We refer to each pixel of input feature maps (IFM) as the activation and the convolution parameters as weight. We refer to the weight corresponding to each 2D convolution operation (i.e., each input-output channel pair) as a *kernel*, while using *filter* to denote a 3D filter corresponding to one output channel. Thus, the weight of a conventional convolutional layer can be represented as a 4D tensor, $F \in \mathbb{R}^{c_{l+1} \times c_l \times d \times d}$, where d is the height and width of a kernel, c_l is the number of input channels (i.e., the number of channels per filter), and c_{l+1} is the number output channels (i.e., the number of filters). Table 1 defines all key terms and quantities introduced in this work.

2.2 Low-Rank Decomposition

All the weights in the same layer can be organized into a weight matrix by combining certain dimensions. Based on the redundancies of the DNN parameters, it is reasonable to assume that the weight matrix is low-rank and can be well-approximated by the multiplication of two small matrices. For example, these matrices can be obtained by conducting singular value decomposition (SVD) to the original matrix and selecting dominating eigenvectors based on the singular values. As done in PENNI [18], low-Rank approximation method will take part of the eigenvectors (usually the ones with larger eigenvalue) and reshape them back to the filter shape as the basis kernels, and express the approximation by the linear combination of the remaining filters. Specifically, the flattened tensor F' has the shape of $c_l c_{l+1} \times d^2$ and the approximated weight tensor $\bar{F}$ is expressed by $\bar{F} = MW$, where $M \in \mathbb{R}^{c_{l+1} c_l \times k}$ is the coefficient matrix, $W \in \mathbb{R}^{k \times d^2}$ represents the basis kernels, and k denotes the number of eignvectors selected. The decomposed weight is illustrated in Fig. 1a. With this decomposed format, the convolution operation can now be computed in two stages: Shared Kernels

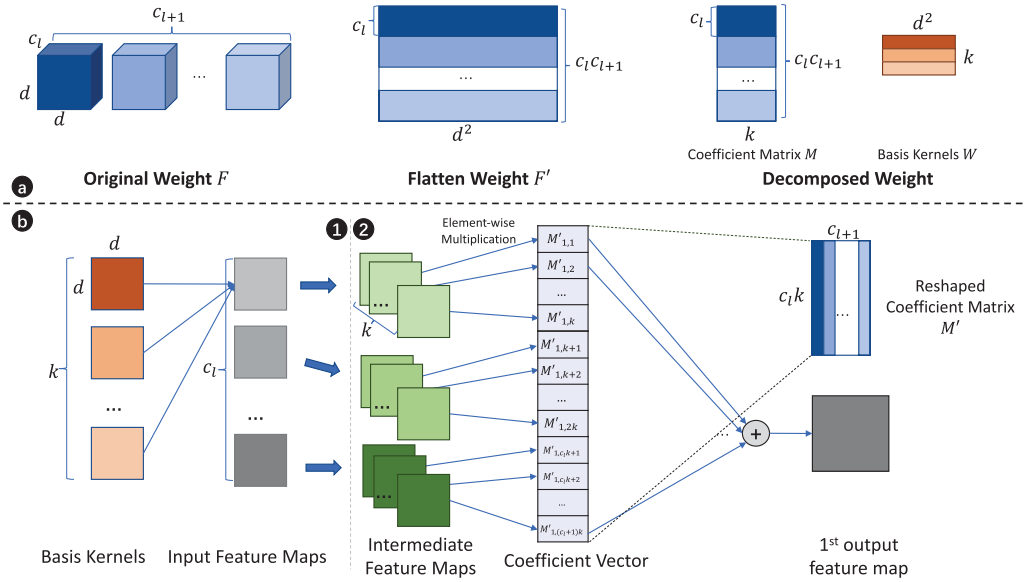


Fig. 1. (a) Illustration of the decomposed weight; (b) DyNNamic's partitioned convolution operation: (1) SKC stage. (2) WA stage.

Convolution (SKC) and Weighted Accumulation (WA). DyNNamic preserves the decomposed format and computes the original convolutional layers by directly using the compact stages. In the first stage, SKC, each *input feature map* (IFM) is convolved with k base kernels and produces $c_l k$ *intermediate feature maps* (IntFM). In the second stage, WA, these IntFMs are then accumulated with different coefficient vectors to produce the *output feature maps* (OFM). With this convolution decomposition scheme, the o-th output feature map is computed as

$$IntFM_{i,j} = IFM_i * W_j, \tag{1}$$

$$OFM_o = \sum_{i=1}^{c_l} \sum_{j=1}^{k} M'_{(i\times k+j,o)} \cdot IntFM_{i,j}, \tag{2}$$

where $*$ denotes the 2D convolution operation, W_j is the j-th base kernel, M' is the is the reshaped coefficient matrix with the shape of $c_l k \times c_{l+1}$. The SKC and WA stages are shown in Fig. 1b.

2.3 Compact Filter Structures and Pruning

Convolutional operation incurs heavy computation costs. A common strategy to combat such high computation costs is to collapse certain dimension of the convolutional filters. For example, depthwise convolution [6] removes the channel dimension of filters, decreasing the total FLOPs for a convolutional layer. A pointwise convolution layer is then implemented to recover inter-channel association necessary for the spatial structure. Pointwise convolution removes the height and width dimensions of filters, which also incur much less computations than conventional filters. Another recent DNN structure is the shift convolution [29]. Shift convolution performs depthwise-separable convolutions via simple shift operations.

To further reduce the computation and storage costs of DNN inference, many recent model compression frameworks incorporate weight pruning, while DNN accelerators like NullHop [2] skip ineffectual computations. Weight pruning removes weights that are not expected to have a significant contribution to model performance. To better match the hardware, many recent pruning schemes are *structured*. Instead of removing weights individually, structured pruning eliminates whole filters, input channels or even a whole layer in a DNN model. This method will not break the data locality of the weights. Thus, the compression directly benefits inference efficiency without any modifications to the software or hardware. Various methods are used to identify unimportant filters, either implicitly like group LASSO [28] or explicitly using metrics like norm [17], geometric median [12].

In terms of compact filter structures, DyNNamic implements SKC, which is analogous to an expanded depthwise convolution, but with several key differences. SKC expands the input channel by k times instead of the identical input/output channels in depthwise convolution. The expansion enhances the representation power of the SKC. Meanwhile, unlike how depthwise uses different kernels for each input channel, SKC utilizes the same set of kernels for each input channel. Thus, the parameters can be shared and the computation flow can be optimized. In terms of pruning, our work adopts group LASSO in a finer-grained, array-aware fashion, thus achieving high compression rate with negligible accuracy loss.

2.4 DNN Hardware Acceleration

The sliding-window nature of convolutional layers generates many data-reuse opportunities. Additionally, convolutional layers can be mapped to matrix-matrix multiplication equivalents. These characteristics motivate systolic-array-based accelerators [4], [5], [14], which can effectively map matrix-matrix and matrix-vector operations. Systolic arrays reduce external bandwidth requirements by internally reusing weight, activation, and/or partial sum data across its PEs. Other computationally intensive DNN layers, such as fully connected (FC), can also be computed with systolic arrays.

3 Motivation

Data-Reuse. Despite being a key characteristic of systolic arrays, data-reuse within these arrays is limited by several

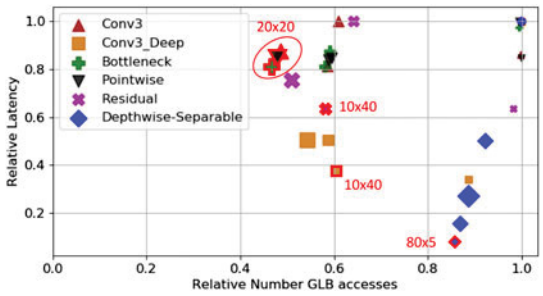


Fig. 2. Simulated relative latency and number of GLB accesses for various DNN layer types, normalized to the worst case for each type. Larger shapes represent more balanced (i.e., square) PE arrays and vice-versa. Optimal PE array shape (Rows/Cols) for each layer type are labeled.

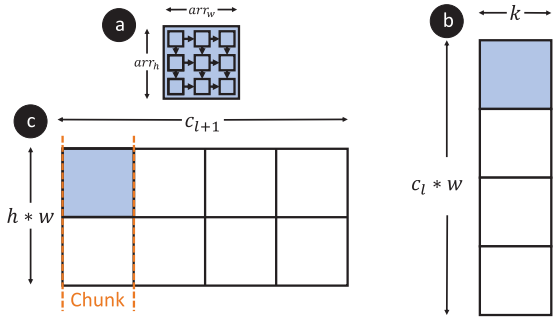


Fig. 3. Mapping SKC and WA using OS dataflow. (a) PE array dimensions. (b) SKC IntFMs mapping. (c) WA OFMs mapping.

factors. First, the size of DNN filters are often much larger than the dimensions of a systolic array, which limits data-reuse to only one direction along the array's rows or columns. In this scenario, a single *pass* over the array cannot be completed in time to further recycle data that has fully propagated through the array. Second, most systolic arrays have inflexible dimensions, which constrains utilization and data-reuse within the array. Such underutilization is made especially worse under DNN layers that are disproportionately small across some of its dimensions. These *unbalanced layers* typically exist in shallow and deeper layers of a DNN—i.e., where the feature maps are disproportionately small or large compared to the layer width—and wherever compact filters are incorporated. To explore the impact of PE array shape and layer type, we simulate various PE array shapes and layer types using SCALE-Sim [24] with an output stationary dataflow. Fig. 2 displays the resulting latency and number of global buffer (GLB) accesses from these experiments. 'Conv3,' 'Conv3_Deep,' and 'Bottleneck' are chosen from ResNet50's layers 13, 49, and 32, respectively. 'Pointwise' and 'Depthwise-Separable' are chosen from MobileNetV2's layers 18 and 17, respectively. Here, GLB access count can inform us of relative energy cost due to data movement. Note that, while the exact details of Fig. 2 are specific to the OS dataflow and specified layers, the overarching observation is applicable to every systolic dataflow and layer type. We observe that certain layer types, including depthwise-separable and deep layers favor unbalanced PE arrays for overall lower latency and energy. Alternatively, PE array shape can be selected to form a trade-off between latency and energy for other layer types. Because DyNNamic employs the compact SKC and WA stages, it is important to understand the data-reuse opportunities of these unbalanced layers given a particular PE array shape.

Exploiting Low-Rank Decomposition. A key characteristic of DNN filter low-rank decomposition is that the convolution operation can be formatted as a linear operation of the resulting decomposed structures. In this paper, we use the decomposition method described in PENNI [18], which produces basis kernels and a coefficient matrix for the SKC and WA stages, respectively. Dimensions of both the set of basis kernels and the coefficient matrix are smaller than that of a complete filter. We leverage these constrained dimensions by separating the convolution operation into compact stages and designing specialized hardware that can efficiently execute these stages. This partitioned convolution operation is then further improved with an array-aware sparsity regularization method to raise energy efficiency and throughput. We show that the basis kernels creates an avenue for circular data-reuse patterns and can be supported with minor additions to the array's controller under an OS dataflow. Additionally, computation cost on the coefficient matrix is greatly reduced with the combination of a compressed matrix representation and coarse-grained zero-skipping. We propose the partitioned convolution mapping and regularization method in Section 4 and accelerator architecture in Section 5.

4 Partitioned Convolutional Layer

DyNNamic implements a partitioned convolutional operation via several modifications to the underlying systolic array architecture. This section describes how DyNNamic supports the two stages, SKC and WA, and how these stages are mapped onto the hardware. Fig. 1 displays the proposed computation scheme.

4.1 SKC and WA Mapping

Fig. 3 depicts how the SKC and WA stages are mapped onto the array using the OS dataflow. The blue square represents the dimensions of the systolic array while Figs. 3b and 3c display unrolled IntFMs and OFMs for SKC and WA stages, respectively. Note that the array width arr_w and the number of basis kernels k are independent parameters. For the SKC stage, each column of the array maps to a single basis kernel while each row maps to individual columns (w dim.) of the IntFM. The h dimension is processed over time using the corresponding rows. For the WA stage, each column of the array maps to a single output channel while each row maps to individual pixels of the OFM.

Despite arr_w being an independent parameter, it should satisfy a couple constraints to maximize PE utilization and throughput for the two stages. During the SKC stage, we prefer $arr_w = k$; otherwise, any additional columns will be unused. Meanwhile, during the WA stage, we prefer arr_w to be small enough for effective array-aware coefficient pruning (discussed in Section 4.2), but large enough to maintain sufficient inter-column data-reuse.

4.2 Array-Aware Coefficient Pruning

Although PENNI [18] shows that we can effectively apply sparsity regularization to the coefficient matrix, we need to ensure that the underlying hardware can effectively use the

sparsity for improved throughput. Notice from Fig. 3c that the number of filters that can be mapped onto the array at a given time is equal to the width of the array, arr_w. We call this collection of filters a *chunk*, which spans the width of a *sub-row* of the coefficient matrix. Motivated by Wen *et al.* [28], we apply sparse group lasso regularization at the chunk granularity to increase the ratio of the all-zero sub-rows in the coefficient matrix. Suppose we have the coefficient matrix M in shape $c_{l+1} \times c_l k$ where c_l, c_{l+1} is the number of the input and output channels, and k is the number of the basis kernels. By performing a zero-skipping scheme, the coefficient matrix can be represented by $M = [w_1 w_2 \ldots w_n]^T$, where $w_1, w_2, \ldots, w_{n-1}$ is in the shape of $arr_w \times c_l k$, and w_n is in the shape of $min(arr_w, c_{l+1} - (n-1)arr_w) \times c_l k$. The regularization term of the l-th layer is

$$R(W_l) = \sum_{i=1}^{n-1} \sum_{j=1}^{w_{array}} ||w_{i,j}||_2, \tag{3}$$

where $w_{i,j}$ represents the j-th row of the i-th chunk. Specifically, this regularization term computes the l-1 norm of the l-2 norm of each sub-row. The overall loss is represented by

$$L'(W, X, Y) = L(W, X, Y) + \lambda \left(|W| + \sum_{l=1}^{p} R(W_l) \right), \tag{4}$$

where $L(W, X, Y)$ is the original loss function (e.g., cross entropy loss), λ is the parameter that controls the regularization strength, and p is the number of convolutional layers. In addition to enforcing structural sparsity, we also add an l-1 regularization term to explore more sparsity.

We adopt the magnitude-based pruning method proposed by Deep Compression [10] and select standard deviation as the pruning threshold. Any lost accuracy is then recovered by a fine-tuning phase. Experimentally, we find that only a few epochs of fine-tuning is required to recover accuracy. Lastly, we apply the model shrinking process mentioned in PENNI [18] and remove the channels with all-zero coefficients.

5 Systolic Array Architecture

5.1 Overview

An overview of DyNNamic's microarchitecture is shown in Fig. 4a. TPU [14] is an industry-standard systolic array-based accelerator, so we utilize it as the baseline design and perform key modifications described in this section. There are three GLBs in the system: row, column, and output; these are used to hold input activation, weight data, and output activation locally. All GLBs, with the exception of the output GLB, are double-buffered to avoid DRAM-related stalls. The core computation block is the dynamic PE array (Fig. 4b), which is comprised of several *PE sub-arrays* (PE-SA), along with small control blocks (Figs. 4c and 4d) that enable the dynamic reshaping. The dynamic PE array performs MAC computations with an OS dataflow. The actual PE implementation is transparent to the dynamic PE array. For the sake of this study, we employ a simple single-cycle MAC PE design shown in Fig. 4e. In between the column GLB and dynamic PE array is the *kernel reuse* (KR) block, which recycles weight data during the SKC stage,

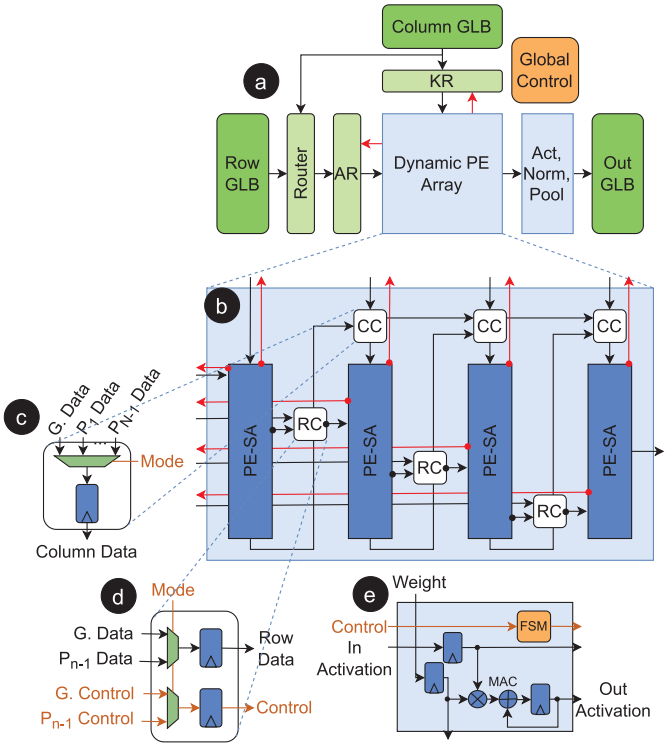


Fig. 4. DyNNamic Microarchitecture. (a) Accelerator Overview, (b) Dynamic PE Array, (c) Column Controller, (d) Row Controller, (e) PE. 'G.' is short for Global and 'P_n' represents the n-th PE-SA.

thus avoiding redundant GLB accesses of the weight data. Between the row GLB and PE array are the activation reuse (AR) block and router, which are used during the SKC stage for reusing activation data. The router *multicasts* data from either the row or column GLB into the PE array's rows. Here, we define multicast as the act of sending one datum to multiple destinations with only one GLB access. Systolic data setup is handled by the global control unit, which orchestrates data movement between the GLBs and functional blocks. Results from the PE array are then sent to the activation, normalization, and pooling module to perform the nonlinear operations of DNNs. The dynamic PE array, router, and AR and KR blocks are the core modifications to the system. The remainder of this section discusses these modules in detail.

5.2 Dynamic Array Shape

The duality of our partitioned convolution operation comes with the caveat of requiring at least two sets of systolic array dimensions for maximal throughput and utilization. The most straightforward approach would be to employ two dedicated structures to handle each stage of the computation. However, doing so would tie the DyNNamic architecture to efficiently executing only the SKC and WA stages. Because systolic arrays are known for being able to execute general matrix-matrix and matrix-vector multiplications, we want to preserve the generalizability of the accelerator. Additionally, introducing another computation structure may generate load-balancing issues and raises the area and power footprint to the accelerator. Instead, we choose to modify the single PE array so that it can dynamically reshape to fit the optimal dimensions of the SKC and WA stages.

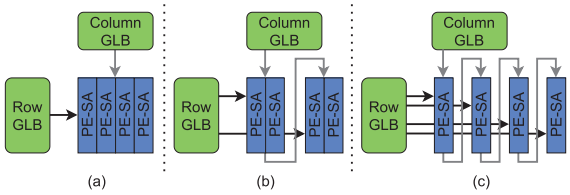


Fig. 5. Conceptual view of dynamic array shaping.

As mentioned in Section 4.1, we prefer that arr_w be equal to k. PENNI [18] shows that $k = 5$ is appropriate for maintaining high accuracy on CNN filters with 3×3 kernels while also balancing the number of FLOPs. 3×3 kernels are frequently used by many modern and conventional CNNs; therefore, to better support the common case, we set the minimum width $arr_{w,min} = 5$. To support more basis kernels, arr_w can be made larger by reshaping the array. Note that we prefer an even rectangular shape for straightforward mapping. To form an even rectangular shape, arr_w must be of the form $arr_{w,min} \times 2^n, n \in \mathbb{N}_0$. For example, given that $arr_{w,min} = 5$, arr_w can equal 5, 10, 20, 40, and so on.

Dynamic array shape is achieved by partitioning the PE array into a collection of PE-SAs, each of which contain $arr_{w,min}$ columns. Fig. 5 depicts the array reshaping concept while Fig. 4b shows its implementation. Between each sub-array is a small row controller that propagates row-wise data from either an adjacent PE-SA or the row GLB. In addition, there are small column controllers above each PE-SA that select among some preceding PE-SAs or the column GLB to propagate column-wise data. Figs. 4c and 4d shows the column and row controller implementations. Multiplexers choose among previous PE-SA(s) or the GLB to receive row or column data while registers temporarily latch the data for the subsequent PE-SA. Control signals are nominally propagated row-wise in this design, so the row controller also has to propagate signals from the previous PE-SA or global control unit. Because the row controllers only ever have to select between two sources, their individual complexity is constant no matter how many PE-SAs are added to the architecture. However, this is not the case for the controllers managing column-wise data. As an example, Fig. 5 shows three possible configurations for an architecture with four PE-SAs. Notice that in order to support all of these configurations, the third PE-SA must be able to accept column-wise data from the first PE-SA, second PE-SA, or the column GLB. In fact, given an array with N PE-SAs, each PE-SA must be able to receive column data from $\log_2(n)$ positions before it, $\forall n < N$. Consequentially, complexity of the column-wise controllers scale logarithmically , which limits scaling to a large number of PE-SAs.

Array Transpose. Despite the limitation to the number of supportable array shape configurations described above, the symmetric nature of the OS dataflow allows us to perform a neat trick to double this number. Activation and weight data feeding patterns are identical and synchronized with each other in the OS dataflow. This means that we can arbitrarily choose either the row or column GLB for the activation or weight data as needed, conceptually transposing the array without additional changes to the controller.

Selecting the Optimal Array Shape. Given that the proposed array can support a selection of possible PE array dimensions, the question naturally arises – which dimensions offer the best performance or efficiency for a given layer? Exhaustively searching for the best array shape by simulating each configuration is the most straightforward approach; however, doing so is time-consuming. In fact, we observe that there is a more convenient way to accurately determine the optimal array shape. Note that the OS dataflow maps the PE array to a subset of an unrolled OFM matrix. Specifically, given an OFM with dimensions $c_{l+1} \times h \times w$, the unrolled OFM matrix will have $h \times w$ rows and c_{l+1} columns. We call each instance of the PE array being mapped onto the unrolled OFM matrix a *pass*. For each pass, it will take approximately $c_l \times d \times d$ feeds from the activation and weight global buffers to fully process the mapped portion of the OFM. No matter how the PE array was mapped onto the OFM, each pass will take the same number of cycles to complete; however, the number of GLB accesses, which signifies relative GLB energy cost, scales with the average bandwidth of the GLBs. Average bandwidth of the GLBs is proporational to the sum of the number of rows and columns in the PE array. In summary, the smaller the number of passes, the lower the latency for processing a given layer. Meanwhile, relative GLB energy cost is determined by the product of the number of passes and the sum of the number of rows and columns in the PE array. The following equations display these relationships:

$$R_{Latency} = \frac{(h \times w)}{arr_h} \times \frac{c_{l+1}}{arr_w}, \tag{5}$$

$$R_{Energy} R_{Latency} \times (arr_h + arr_w), \tag{6}$$

where $R_{Latency}$ and R_{Energy} are the relative latency and GLB energy cost of each PE array shape, respectively. Using these equations to determine the appropriate array shapes, one can optimize for latency or energy cost for most DNN layer types without extensive simulations. Equation (5) also explains why deeper convolution layers prefer unbalanced arrays. When $h \times w$ is small and c_{l+1} is large, having a wider array better fits the unrolled OFM.

Equations (5) and (6) do not cover every layer type. As shown in Table 2, certain layer types limit data-reuse across key dimensions, thus impacting utilization when their OFMs are mapped onto a systolic PE array. For depthwise layers such as that proposed by Xception [6] and our proposed SKC, the mapping of filters to the columns of the PE array is significantly constrained. In the case of the SKC stage, the maximum number of active columns in the PE array for one pass is equal to the number of bases k. Meanwhile, depthwise convolution can only utilize one column

TABLE 2
Example Layer Types of DNNs

Layer Type	Impact on Layer Dimension or PE Array Mapping
Standard Convolution	N/A
Deep Convolution Layer	Small feature map ($h \times w$)
Depthwise-Separable Convolution	$c_l = 1$, $arr_{w,utilized} = 1$
Pointwise Convolution	$d = 1$
Bottleneck Layer	Relatively small c_{l+1}
Downsampling Layer	Relatively large c_{l+1}
SKC	$c_l = 1$, $arr_{w,utilized} = k$
WA	$d = 1$

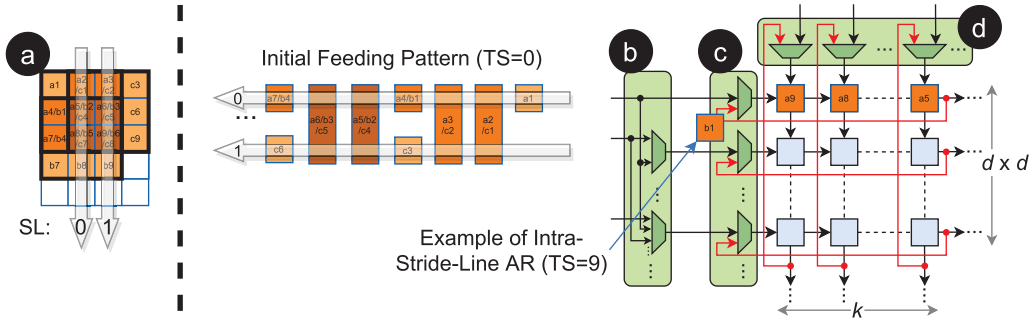


Fig. 6. Activation feeding patterns (left) and reuse blocks with working example (right). SL: Stride Line, TS: Time Stamp.

per pass, which is an extreme case. Under these conditions, the equation for relative latency changes to

$$R_{Latency} = \frac{(h \times w)}{arr_h} \times \frac{c_{l+1}}{g}, \tag{7}$$

where $g = k$ for the SKC stage and $g = 1$ for depthwise. From this result, it is easy to see the SKC and other depthwise layers prefer unbalanced PE arrays. We verify Equations $5-7$ through simulation and obtain equivalent results to Fig. 2.

In this study, we specially design the hardware to suit the partitioned convolution operation; but, we keep the architecture flexible enough to support other layer types. Improvements due to dynamic PE array are shown in Section 7.5.

5.3 SKC Circular and Multicast Data Paths

The SKC stage is characterized by a small collection of small basis filters. As mentioned in Section 4.1, these basis kernels are reused across the c_l input feature map channels. Naïvely fetching basis kernel data repeatedly for each input channel would generate wasteful energy costs. Assuming a kernel size of $d \times d$ and k basis kernels the maximum number of times basis kernel data is fetched in one pass during the SKC stage is $k \times d \times d$. The sliding-window behavior of the convolution operation also causes activation data to be repeatedly fetched from the GLBs. However, the small size and repeated use of these basis kernels exposes two forms of reuse: *kernel reuse* and *intra-stride-line activation reuse*.

Kernel Reuse (KR). After the first set of kernel weights are supplied from the global weight buffer, we can recycle the weight data within each column. This recycling behavior results in no more need for accessing weight data from the GLB for the remainder of the pass, thus reducing its bandwidth to zero during this computation. We exploit this idle GLB via GLB sharing, described in Section 5.4. Fig. 6a depicts the KR block. Multiplexers are used to select between GLB feed or recycle feed(s). Kernel sizes of 9, 25, and 49 are supported, corresponding to kernel dimensions of 3×3, 5×5, and 7×7, respectively. We recognize that an overwhelming majority of modern CNNs utilize 3×3 kernels, but include less popular kernel sizes to increase flexibility.

Activation Reuse (AR). Similar to the KR block, the AR block recycles activation data within each row. Note that because basis kernels are reused both within and across input channels, we can select how data is mapped across the array's rows to maximize activation reuse opportunity while limiting control complexity. We define a *stride-line* as the path of a convolution operation as it propagates along the feature map. Fig. 6 depicts stride-lines with clear arrows. Assuming each row is assigned to the collection of OFM pixels along one stride-line and $stride < d$, subsequent convolution windows (represented with 'a' and 'b' pixels) will reuse activation data. As a result, a majority of the data being fed into each row can be recycled instead of being refetched. Specifically, using this AR scheme, the number of GLB accesses within a stride-line is reduced to

$$APL = (d \times d) + (d \times min(d, stride)) \times (y - 1), \tag{8}$$

where APL is the number of GLB accesses per stride-line and y is the length of the OFM along either its height or width dimension. Let x be the length of the dimension opposite of y. Note that the above scheme applies only within each stride-line and that there are a total of $x/stride$ stride-lines. Thus, the overhead of $(d \times d)$ is incurred $x/stride$ times with only this approach. Fig. 6b portrays intra-stride-line AR.

Another approach to recycle activation data is across stride-lines (i.e., inter-stride-line AR). Intra and inter-stride-line AR are orthogonal approaches and can be combined for further improved data-reuse. Assuming adjacent PE array rows are set to adjacent stride-lines, shared activations can be multicast. Combining these two techniques, the total number of accesses to the GLBs for activation data is reduced, at minimum, to

$$Accesses = APL + APL \times \frac{y}{stride} \times \frac{min(d, stride)}{d}. \tag{9}$$

Note that Equation (9) is the best case number of GLB accesses for activation data using both AR approaches. In practice, this number is slightly increased because the number of rows in the PE array can be less than the total number of stride-lines. Fig. 6c displays the inter-stride-line AR block implementation, which is a router that multicasts activation data. The router is implemented with a cascade of multiplexers. In order to limit wiring complexity of the router, each destination row can only receive input from source rows located up to $d_{max} - 1$ rows before it, where $d_{max} = 7$ is the maximum kernel height or width supported by this architecture. This limits the largest size of a multiplexer in the router to $7:1$. Also, due to the reduced bandwidth needed for activation data accesses, we can address the increased bandwidth demands of unbalanced PE array shapes. The next section describes how we leverage this

property alongside GLB sharing to mitigate the penalty of dynamic array shaping.

5.4 Global Buffer Sharing

Dynamically shaping a PE array's dimensions causes the GLBs' bandwidth demands to change significantly. Take Fig. 5 as an example. In a square array scenario, i.e., Fig. 5a, the maximum bandwidth required from the row and column GLBs are equal. Let us refer to the bandwidths of the GLBs under a square array as BW_{square}. As the array shape becomes more unbalanced, the maximum required bandwidth for the row and column GLBs scale proportionally to the array's length and width, respectively. Specifically, the array shape in Fig. 5b is twice the original length and half the original width. This equates to maximum bandwidth requirements of $2 \times BW_{square}$ and $0.5 \times BW_{square}$ for the row and column GLBs, respectively. Likewise, the bandwidth requirements shown in Fig. 5c are $4 \times BW_{square}$ and $0.25 \times BW_{square}$ for the row and column GLBs, respectively. Without any changes to routing of the GLBs, this trend suggests that the row GLB must have a maximum bandwidth equal to the maximum number of rows in the array. However, simply implementing a row GLB with such large bandwidth can be very costly. In fact, implementing such a high bandwidth GLB is wasteful if the array shape does not frequently use its most unbalanced configuration. Instead, we leverage the unused bandwidth of the column GLB as well as the lower bandwidth demands discussed in Section 5.3 to balance the bandwidth demand without modifying the row GLB.

GLB Sharing Implementation. GLB sharing is implemented by connecting column GLB outputs to the router shown in Fig. 4a. Prior to the SKC stage, the column GLB is preloaded with the small number of basis kernels as well as activation data corresponding to the lower rows of the PE array. At the start of the SKC stage, the column GLB supplies the basis kernels into the PE array's columns. Once finished, the column GLB is used to supply activations into the PE array's rows alongside the row GLB. During the SKC stage, the bandwidth demand for activation data is cut to one-third the usual amount, which means that the GLBs are able to effectively supply up to $6 \times BW_{square}$ bandwidth of data. Thus, we choose the most unbalanced array dimensions of the PE array such that its maximum row-wise bandwidth demand equates to $4 \times BW_{square}$. In summary, we combine the multiple data reuse opportunities of the SKC stage with GLB sharing to mitigate the heightened bandwidth demands of unbalanced array configurations.

5.5 WA Computation Mapping and Control

Unlike the dense basis kernels of the SKC stage, the WA stage involves a sparse coefficient matrix. We restructure the coefficient matrix by decoupling the chunks, compressing the chunks by pruning their all-zero sub-rows, and stacking the chunks into a compressed matrix. Assuming n chunks, we can mark the start and end indices of the chunks within the compressed matrix using a *chunk pointers* array of size $n + 1$. The first entry of this array is always '0' and the last entry $n + 1$ is the height of the compressed coefficient matrix plus one. Let $[c_l k]_n$ represent the the height of chunk n. In addition to distinguishing separate chunks in the compressed matrix, we must ensure that IntFM data being fed into the systolic array align with the coefficients. By pruning sub-rows of the chunks, a conventional OS IntFM access pattern no longer works. To align the IFM data with the coefficients, we incorporate another array of pointers called *IFM pointers*. The length of the array is equal to the total height of the compressed coefficient matrix. Elements in this array point to the IFM channel that corresponds to each row of the compressed matrix. Recall that there are $c_l k$ IFM channels in total prior to pruning. Fig. 7 displays these structures. The proposed compressed matrix format generates savings in the form of greatly reduced computation count and less data transfers among both on- and off-chip buffers.

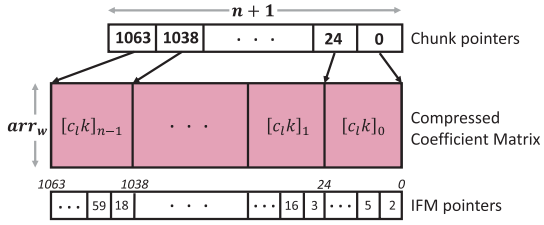


Fig. 7. Coefficient matrix compressed format.

6 Experimental Methodology

6.1 DNN Model Performance Validation

To validate our partitioned convolution operation and pruning method, we evaluate TOP-1 accuracy and compression rate of a variety of DNN models and two datasets. Specifically, we choose VGGNet [26], ResNet [11], and MobileNetv2 [25] as representative DNNs to show that our method is applicable to – and incurs marginal accuracy loss on – a wide variety types of DNN models. VGG models are heavily overparameterized networks that are ideal candidates for model compression. To further explore more complex networks, we also include results from ResNet and MobileNet. On top of the conventional 3×3 convolutional window, ResNet models incorporate bottleneck layers, which are significantly less computationally intensive. Additionally, ResNet models use residual connections. Lastly, MobileNet models are mostly comprised of compact filter structures: depthwise and pointwise convolutions. To better represent different complexities of datasets, we choose CIFAR-10 [15] and ImageNet [7].

All models are trained with Stochastic Gradient Descent (SGD) with 0.9 nesterov momentum. Initial learning rate was set to 0.1 and cosine annealing was adopted in all training procedures. On CIFAR10, We train the baseline model for 300 epochs with 0.0001 weight decay. The regularization retraining phase lasts for 300 epochs with 0.01 initial learning rate, 0.00002 regularization strength. The fine-tuning phase after pruning lasts for 100 epochs. On ImageNet, we use the same regularization settings, and the training and fine-tuning last 50 and 30 epochs, respectively. Lastly, we quantize both the weight and activation to 8-bit fixed-point integers. Table 3 lists baseline accuracies for all models.

6.2 Accelerator Architecture Modelling

Baseline Accelerators. DyNNamic targets improving performance and efficiency of DNN inference tasks. For this

TABLE 3
Accuracy and Accelerator-Aware Sparsity (AA-Sparsity) of the Decomposition and Pruning Algorithm

Dataset	Model	Base Acc. (%)	Final Acc. (%)	AA-Spar. (%)	Param. Spar. (%)
CIFAR-10	VGG-16	93.49	93.7	94.68	93.12
	VGG-19 [16]	93.70	94.70	-	84.00
	VGG-19	93.70	93.48	90.80	91.14
	ResNet-18	93.79	94.3	93.65	96.59
	MobileNetV2	94.09	93.35	95.66	90.98
ImageNet	VGG-16	73.37	73.01	87.88	91.35
	VGG-19 [16]	74.24	71.81	-	87.40
	VGG-19	74.24	73.80	89.31	90.12
	ResNet-50	76.22	74.11	71.86	81.70
	MobileNetV2	71.88	70.21	63.64	78.55

TABLE 4
Hardware Parameters

Accelerator	Global Mem.	Number PEs	Mem. per PE	MAC units per PE
DianNao	450 KB	25	2 KB	16
Eyeriss	325 KB	400	448 B	1
Cambricon-X	450 KB	25	2 KB	16
SCNN	N/A	25	20 KB	16
SparTen	N/A	400	1.25 KB	1
Ours	316 KB Row GLB 180 KB Col GLB 2 KB Output GLB	400	1 B Act. Buffer 1 B Weight Buffer 3 B Accum. Buffer	1

reason, we evaluate end-to-end latency and energy consumption of a single-frame inference task and compare against various state-of-the-art and baseline DNN accelerators. DianNao [3] is an early DNN accelerator with notable throughput that incorporates 3-level memory (i.e., dedicated registers, GLB, and off-chip memory). Eyeriss [4] employs the energy-efficient RS dataflow alongside a network-on-chip (NoC). Cambricon-X [30] combines a compressed data format with efficient indexing to avoid wasteful data transfers and computation. SCNN [21] targets convolution operations and leverages a compressed encoding scheme and high-data-reuse multiplier arrays for improved energy-efficiency. SparTen [8] uses a bit-mask-based compression scheme alongside load balancing for reduced data movement and improved utilization. Column Combining [16] addresses underutilization of systolic arrays by packing non-zero weights of sparse filters into dense tensors; we evaluate this work separately from the others because it is a co-designed approach that requires separately trained models. Lastly, to expose the individual contributions of DyNNamic, we also include the results of several configurations: a vanilla systolic array, DyNNamic with only the dynamic PE array (Ours-D), DyNNamic with only the proposed algorithm and compression methods (Ours-A), and the combination of all techniques discussed in this paper (Ours). For all experiments, only convolutional layers are considered; this is to be fair to SCNN and because convolutional layers form the DNN computation bottleneck.

Architecture Modelling. Table 4 summarizes key technology parameters of all accelerators for the experiment. Accelerator efficiency and performance are directly influenced by MAC unit count and cumulative buffer size, so these parameters are kept constant across all accelerators. Frequency and computation precision are set to 300 MHz and 8-bit, respectively. Also, all designs are scaled to 65 nm technology node for apples-to-apples comparison. These technology projections are done using traditional scaling rules for short-channel devices, described by Stillmaker and Baas [27]. The scaling rules are quadratic, linear, and constant for area, frequency, and power, respectively. Due to the large buffer size required by SCNN and relatively small buffer size of other baseline architectures, we compromise among the accelerators with a total on-chip capacity of 500 KB. Our design is synthesized to estimate the power/area consumption using Synopsys Design Compiler (DC) with TSMC 65nm library. For the design's GLBs and off-chip memory access, energy costs of SRAM and DRAM, respectively, are estimated with CACTI 7.0 [1]. Specifically, energy consumption of the memory subsystem is modelled by computing the buffer leakage power and multiplying the number of accesses by the unit-energy cost of a read or write.

We evaluate the performance of DyNNamic by modifying the cycle-accurate simulator, SCALE-Sim [24]. We retrieve DNN weights after training on PyTorch [22] and pre-process the coefficient matrices to generate the proposed compression format. The resulting weights are then fed into the simulator, thus producing cycle-accurate performance measurements. Column Combining is modelled similarly. We simulate DianNao and Eyeriss performance using Timeloop [20]. Likewise, we simulate SCNN using DNNSim [13]. Lastly, neither Cambricon-X nor SparTen have a published open-source simulator. Instead, we closely follow the description of the original papers to reproduce and validate in-house cycle-accurate simulators. To safely compare across the various simulators, we unify the modelling methodology of registers, on-chip and off-chip memory for all simulators. For example, all simulations assume double-buffered on-chip memories; also, each MAC unit is set to execute in one cycle while on-chip buffers and multi-hop NoCs induce per-hop cycle penalty. These modifications are consistent with Timeloop's simulation methodology that highlights dataflow impact rather than the low-level micro-architecture. Switching energy of baseline designs is conservatively estimated by assuming a per-MAC cost of 0.081 pJ, which is obtained via synthesizing one MAC module in DC.

7 Experimental Results

7.1 Array-Aware Sparsity Evaluation

The results obtained by array-aware coefficient pruning is displayed in Table 3. 'Param. Spar.' presents the portion of zero-valued weights after decomposition. 'AA-Spar.' stands for accelerator-aware sparsity after decomposition, also interpretable as chunk-wise sparsity; here, we only count the sub-rows with zero-valued weights. Specifically, AA-Spar equals the number of all-zero sub-rows divided by the total number of sub-rows in the filter tensors. Note that we consider the sub-rows in the last chunk as part of the computation, which may result in 'AA-Spar.' being larger than 'Param. Spar.' if the last chunk has a width smaller than arr_w. On CIFAR10, we can achieve more than 90% sparsity on all evaluated models. For the relatively large models like VGG and ResNet-18, the pruned models have even higher

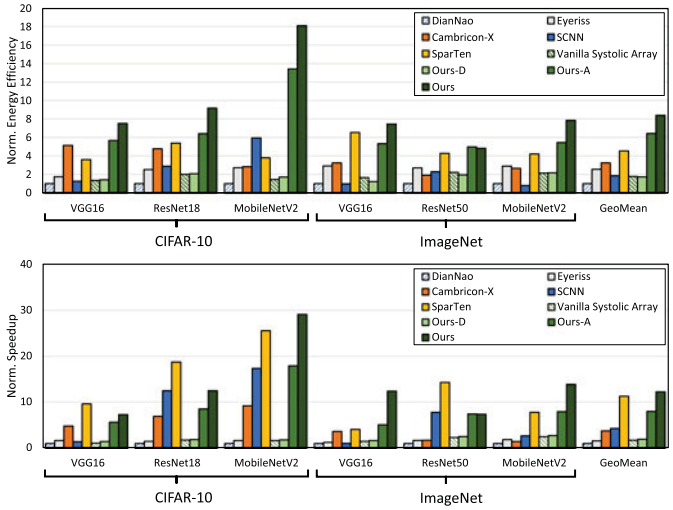


Fig. 8. Normalized energy efficiency and speedup of all accelerators.

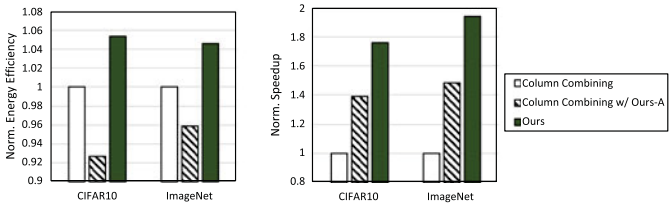


Fig. 9. Comparison to Column Combining on VGG-19.

accuracy since the coefficient pruning works like regularization and prevents overfitting. For the compact model MobileNetV2, although the accuracy loss rises to 0.7%, we can still maintain over 93.3% accuracy while pruning more than 90% of parameters.

On ImageNet, we also achieve high sparsity relative to accuracy loss on the VGG and ResNet-50 models. Performance on MobileNetV2 is particularly impacted since it is already a compact model; obtaining a high sparsity on such a model is difficult without sacrificing significant accuracy. ImageNet's higher complexity causes the model to be more sensitive to this effect, explaining the performance gap between the datasets. These reasons combined are why compression works avoid pruning MobileNetV2, but we include the results to show the effectiveness of our approach.

7.2 Accelerator Performance

Fig. 8 shows the energy efficiency and speedup of the accelerators on the evaluated models. All values are normalized to DianNao.

First, we will discuss the energy efficiency results compared to competing accelerators. Energy efficiency is defined as the number of inferences per unit energy. Fig. 8 shows that DyNNamic consumes the least amount of energy per DNN inference and offers energy efficiency improvements ranging from $4.8\times$ to $18.1\times$ compared to DianNao. Compared to SparTen, the most competitive accelerator among those evaluated, DyNNamic improves energy efficiency between $1.1\times$ and $4.8\times$. Unlike SCNN and SparTen, both of which skip zeros at a fine granularity via fine-grained indexing, DyNNamic leverages the decomposed convolutional layer format and chunk-granularity sparsity with significantly lower indexing overhead. In most cases, Cambricon-X is more efficient than SCNN because its unified buffer allows all PEs to fetch shared activation data without incurring redundant off-chip access; SparTen avoids this issue by pinning spatial positions to individual CUs and avoiding additional inter-CU communication related to the computation's inner-join. SCNN and SparTen perform worse under the WA and SKC stages, respectively, due to how the workload dimensions are split across their CUs, so we compare against the accelerators using the recomposed convolutional layers for stronger baselines. DyNNamic performs particularly well on MobileNetV2 because the model heavily uses compact layer types depthwise and pointwise, which DyNNamic's reshaping PE array handles particularly well. These significant energy improvements are the result of DyNNamic's heavy data-reuse during the SKC stage and efficient coefficient matrix compression during the WA stage, revealed by comparing 'Ours-A' to 'Vanilla Systolic Array'. Additionally, the dynamic PE array further boosts energy efficiency by avoiding array shapes that adversely affect data-reuse. 'Ours-D' actually performs worse than the 'Vanilla Systolic Array' baseline because the array is configured to dynamically reshape for optimal speedup, not energy efficiency.

Now, we discuss the latency speedup results. DyNNamic has significant speedup compared to the baseline accelerators and is comparable to SparTen. As shown in Fig. 8, DyNNamic achieves a speedup ranging from $7.3\times$ to $29.1\times$ compared to DianNao. DyNNamic reaches comparable or higher speedup than all baselines, with the exception of SparTen on VGG-16 (CIFAR-10) and on the ResNet models. ResNet models heavily employ Pointwise and Residual layers, which SparTen excels at computing because it pins each spatial position to a CU; additionally, SparTen exploits activation sparsity for an added advantage to runtime speedup. Meanwhile, DyNNamic's speedup is mainly attributed to directly supporting the decomposed convolutional operation format, the significant chunk sparsity of the coefficient matrix during the WA stage, as shown by 'Ours-A'. The heightened utilization of the PE array from dynamic reshaping then allows the accelerator to exploit the compact filters generated by 'Ours-A'.

7.3 Comparison to Column Combining

Because Column Combining is a co-designed approach, we first compare its sparsity to ours. As shown in Table 3, on both CIFAR-10 and ImageNet, our VGG-19 models are sparser by 7.14% and 2.72%, respectively. By exploiting low-rank decomposition to reduce parameter size and expose the easily-prunable coefficient matrix, we achieve higher sparsity with comparable accuracy compared to Column Combining, despite structured pruning methods typically producing less sparse models.

Fig. 9 shows the relative energy and runtime of an inference task on VGG-19. As shown, DyNNamic is only able to improve energy cost by up to 5% due to the inherently lower data-reuse available using the partitioned convolution operation. Meanwhile, DyNNamic can improve overall runtime by up to $1.9\times$ thanks to the lower parameter count and superior compression of Ours-A. We also show the result of directly using Ours-A on Column Combining. Column Combining cannot efficiently support the compact SKC stage, so its energy efficiency is degraded. However,

TABLE 5
On-Chip Area and Energy Breakdown

Module	Area (μm^2)	Area %	Energy (μJ)	Energy %
PE Array	522648	88.68	1142.773	86.25
PE-SA ($\times 4$)	130320	22.11	285.158	21.45
RC ($\times 3$)	261	0.04	0.571	0.04
CC avg. ($\times 3$)	195	0.03	0.429	0.03
KR & AR	4952	0.84	10.856	0.82
Multicast Router	41421	7.03	81.218	6.13
Act., Norm., & Pooling	7840	1.33	18.979	1.43
Control	12472	2.12	71.157	5.37
Total	589333	100.0	1324.987	100.0

Ours-A can reduce Column Combining's runtime by up to $1.5\times$ via the reduced parameter count and computation footprint.

7.4 Power, Area, and Energy Breakdown

Table 5 displays the synthesized area and energy breakdowns of DyNNamic for an inference on ResNet-50, excluding on- and off-chip memory. A significant portion of the on-chip area and energy are due to the PE array, equating to roughly 89% and 86%, respectively. Most of the PE array's footprint comes from the PE-SAs while the RCs and CCs have negligible footprints of less than 0.05%. Meanwhile the area and energy overheads of the circular reuse blocks total less than 1%.

7.5 Utilization of Dynamic Array

This section evaluates the effectiveness of the proposed dynamic array shaping on the SKC and WA stages. Fig. 10 displays the decomposed layer-wise throughput of a static systolic array and of our dynamic array, evaluated on ResNet-18. Note that each convolutional layer with non-unit kernel dimensions is decomposed to two stages, which we visualized as decomposed 'sub-layers'. The peak throughput of a 400 PE array at 300 MHz is 240 GFLOPS. On average, the static and dynamic arrays have an average throughput of 125 (52 %) and 195 (81 %) GFLOPS (utilization), respectively. The SKC stage cripples the throughput and utilization of the static PE array because of its depthwise nature. The dynamic PE array solves this issue by adjusting its dimensions, thus achieving high utilization on both stages.

7.6 Ablation Study

Fig. 11 showcases the impact of isolating each approach described in this work. Figs. 11a and 11b display the impact of each approach given differing layer types, normalized to the most-performant combination of techniques. The evaluated layer types include: First Layer (FL), Bottleneck Down-Scaling (BNDS) and Up-Scaling (BNUS), 3×3 Convolution (Conv3), Residual (Resid), Depthwise (DSC), and Pointwise (PW).

Due to the small number of channels for FL and DyNNamic optimizing for speedup, data-reuse is sacrificed for speedup compared to vanilla systolic arrays; DianNao also suffers on FL because it depends on channel-wise parallelism. Results for BNDS, BNUS, Resid, and PW are mostly consistent due to their similarity in employing 1×1 kernels. Conv3 is similarly influenced by the proposed techniques, but is more performant on DianNao and the vanilla systolic array due to its larger kernel dimensions. DSC layers serve to be the biggest issue for the proposed techniques, but does not form a computation bottleneck. This is because the proposed algorithm relies on 4-D filters to decompose into SKC and WA stages. Because DSC layers do not have any interaction across the channels, the large WA stage incurs computation overhead. These results show that directly applying the dynamic PE array without the algorithm is preferred on DSC layers.

Figs. 11c and 11d showcase the overall contribution of each approach on efficiency and speedup, respectively. The contributions of the approaches are ranked as follows. On MobileNetV2, dynamic PE array, algorithm, buffer sharing, and circular reuse. The order of significance is slightly different on ResNet-50: algorithm, dynamic PE array, buffer sharing, and circular reuse. ResNet-50 is primarily composed of Conv3, BNDS, BNUS, and Resid layers, which makes it a compelling target for the algorithm and compression methods. Meanwhile, MobileNetV2 is composed of compact DSC and PW layers, causing the dynamic PE array to be of greater importance.

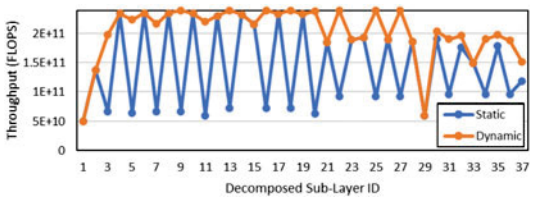


Fig. 10. Static vs. dynamic PE array on decomposed layers.

7.7 Optimal Number of PEs

Larger PE count typically improves throughput of the chip, but overhead of the dynamic PE array is not negligible if the

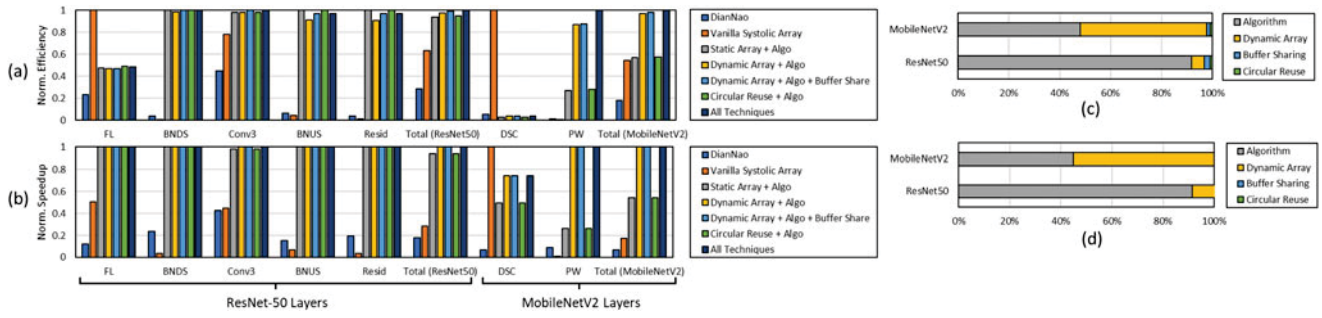


Fig. 11. Isolated techniques. (a) Layer-Wise Efficiency, (b) Layer-Wise Speedup, (c) Efficiency Contribution, and (d) Speedup Contribution.

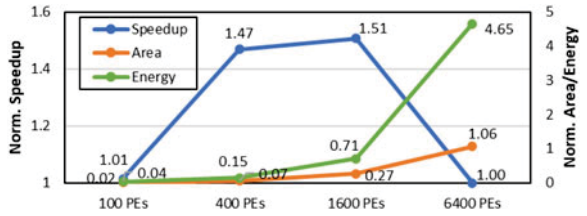


Fig. 12. Varying PE array size.

PE count gets too large. One clear parameter that defines the set of possible array sizes is $arr_{w,min}$. As discussed in Section 5.2, we choose $arr_{w,min} = 5$ to maximize utilization of the SKC stage for the common case (i.e., 3×3 kernels). Given the minimum array width and the dependency with the array's dimensions discussed in Section 5.2, the set of possible array sizes is limited to 25, 100, 400, 1600, and so on. Fig. 12 shows the relative speedup, area, and energy of the dynamic PE array with various possible array sizes. Note that speedup is relative to a static array while area and energy are relative to the GLBs' cumulative area and energy; we compare to the GLBs here because they form the on-chip area and power bottleneck. Thus, it is important that the PE array's area and energy footprints do not approach the GLBs' footprints. This figure shows the dynamic PE array generates the most speedup with 400 or 1600 PEs. Additionally, the energy and area footprints are *both* significantly less than that of the GLBs for PE counts less than 1600. Larger PE counts will generate unacceptable area and energy consumption from the PE array and dynamic shaping controllers; also, as PE count grows, dynamically reshaping the PE array provides limited throughput improvement. This limited improvement is caused by the array's dimensions being too large to fully utilize its range of possible configurations. For example, the most unbalanced configuration of the array with 6400 PEs is 5×1280. On ResNet-18 using CIFAR-10, the maximum OFM size is 1024 while the maximum layer width is 512, neither of which are equal-to or larger than 1280. In conclusion, we observe that a PE array size 400 (i.e., four 20×5 PE-SAs) is optimal for the target partitioned convolution operation.

8 Related Works

DNN applications are heavily data-centric and typically involve many memory transactions. To minimize the impact of data transfers, recent accelerators like DianNao [3] incorporate multi-level memory, which reduces the cost of frequently accessed weight and activation data via temporal locality. Other accelerators opt to avoid redundant memory accesses via systolic array processing, which heavily exploits spatial locality. TPU [14] boasts up to $20\times$ higher throughput and $80\times$ higher energy efficiency when compared to contemporary GPUs and CPUs; however, it suffers from low utilization due to a lack of reconfigurability. Guo *et al.* [9] combine systolic array processing with a GPU to improve processing flexibility and efficiency of the underlying hardware. DyHard-DNN [23] determines the best configuration of a systolic array and dynamically configures the architecture to reduce latency and energy cost. Liu *et al.* [19] propose a flexible computation mapping and dataflow scheme. These methods address underutilization of systolic arrays; however, they do not modify the algorithm to better leverage the hardware characteristics.

9 Conclusion

This paper presents DyNNamic, a DNN inference accelerator. DyNNamic leverages characteristics of kernel-wise decomposition by partitioning the convolution operation into two stages: Shared Kernels Convolution (SKC) and Weighted Accumulation (WA); this greatly enhances execution energy efficiency and latency. We map the two-stage convolution operation onto the dynamic PE array, which accommodates the vastly different dimensional constraints of each stage. We then aggressively compress the large coefficient matrix of the WA stage using using a structured compressed format. Lastly, we analyze and exploit data-reuse opportunities exposed by the SKC stage. Experiments show that DyNNamic outperforms recent state-of-the-art accelerators, with on average $8.4\times$ energy efficiency and $12.3\times$ speedup.

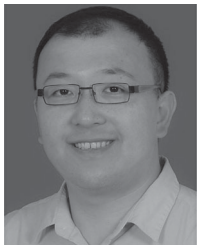
References

[1] CACTI:an integrated cache and memory access time, cycle time, area, leakage, and dynamic power model, [EB/OL], 2013. [Online]. Available: https://github.com/HewlettPackard/cacti
[2] A. Aimar *et al.*, "Nullhop: A flexible convolutional neural network accelerator based on sparse representations of feature maps," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 3, pp. 644–656, 2019.
[3] T. Chen *et al.*, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM SIGARCH Comput. Archit. News*, vol. 42, no. 1, pp. 269–284, 2014.
[4] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.
[5] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible and high-performance accelerator for emerging deep neural networks," *IEEE J. Emerg. Sel. Top. Circuits Syst.*, vol. 9, no. 2, pp. 292–308, Jun. 2019.
[6] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 1251–1258.
[7] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2009, pp. 248–255.
[8] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. N. Vijaykumar, "Sparten: A sparse tensor accelerator for convolutional neural networks," in *Proc. 52nd Ann. IEEE/ACM Int. Symp. Microarchit.*, 2019, pp. 151–165.
[9] C. Guo *et al.*, "Balancing efficiency and flexibility for DNN acceleration via temporal GPU-systolic array integration," in *Proc. 57th ACM/IEEE Des. Automat. Conf.*, 2020, pp. 1–6.
[10] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," 2015, *arXiv:1510.00149*.
[11] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.
[12] Y. He, P. Liu, Z. Wang, Z. Hu, and Y. Yang, "Filter pruning via geometric median for deep convolutional neural networks acceleration," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2019, pp. 4340–4349.
[13] isakedo, isakedo/dnnsim, [EB/OL], 2020. [Online]. Available: https://github.com/isakedo/DNNsim
[14] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 1–12, Jun. 2017.
[15] A. Krizhevsky *et al.*, "Learning multiple layers of features from tiny images," Univ. Toronto, Toronto, Ontario, Canada, Tech. Rep., 2009.

[16] H. Kung, B. McDanel, and S. Q. Zhang, "Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2019, pp. 821–834.
[17] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," 2016, *arXiv:1608.08710*.
[18] S. Li, E. Hanson, H. Li, and Y. Chen, "PENNI: Pruned kernel sharing for efficient CNN inference," *Proc. 37th Int. Conf. Mach. Learn.*, vol. 119, pp. 5863–5873, 2020. [Online]. Available: http://proceedings.mlr.press/v119/li20d.html
[19] B. Liu *et al.*, "Addressing the issue of processing element under-utilization in general-purpose systolic deep learning accelerators," in *Proc. 24th Asia South Pacific Des. Automat. Conf.*, 2019, pp. 733–738.
[20] A. Parashar *et al.*, "Timeloop: A systematic approach to DNN accelerator evaluation," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2019, pp. 304–315.
[21] A. Parashar *et al.*, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 27–40.
[22] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 8026–8037.
[23] M. Putic, S. Venkataramani, S. Eldridge, A. Buyuktosunoglu, P. Bose, and M. Stan, "Dyhard-DNN: Even more DNN acceleration with dynamic hardware reconfiguration," in *Proc. 55th Annu. Des. Automat. Conf.*, 2018.
[24] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "SCALE-sim: Systolic CNN accelerator simulator," 2018, *arXiv: 1811.02883*.
[25] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 4510–4520.
[26] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.
[27] A. Stillmaker and B. Baas, "Scaling equations for the accurate prediction of CMOS device performance from 180nm to 7nm," *Integration, VLSI J.*, vol. 58, pp. 74–81, 2017.
[28] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 2074–2082.
[29] B. Wu *et al.*, "Shift: A zero flop, zero parameter alternative to spatial convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 9127–9135.
[30] S. Zhang *et al.*, "Cambricon-x: An accelerator for sparse neural networks," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2016, pp. 1–12.

Edward Hanson received the BS degree in computer engineering from the University of Maryland, Baltimore County (UMBC) in 2019 and was awarded the Meyerhoff Premier Scholarship. He is currently working towards the PhD degree in computer rngineering with Duke University under the supervision of Prof. Yiran Chen. His research interests are in machine learning system acceleration and computer architecture.

Shiyu Li received the BEng degree in automation from Tsinghua University, Beijing, China, in 2019. He is currently working toward the PhD degree with the Department of Electrical and Computer Engineering, Duke University, Durham, NC, USA, supervised by Prof. Yiran Chen. His research interests include computer architecture, algorithm-hardware co-design of deep learning systems, and neural architecture search.



Xuehai Qian received the PhD degree from the University of Illinois Urbana Champaign. He is currently an assistant professor with the University of Southern California. His research interests include domain-specific systems and architectures, performance tuning and resource management of cloud systems, and parallel computer architectures. He is the recipient of W.J Poppelbaum Memorial Award at UIUC, NSF CRII and CAREER Award, and the inaugural ACSIC Rising Star Award.

Hai (Helen) Li (Fellow, IEEE) received the PhD degree from Purdue University in 2004. She is currently a professor with the Department of Electrical and Computer Engineering, Duke University. Her current research interests include neuromorphic computing systems, machine learning and deep neural networks, and memory design and architecture. She is a recipient of the NSF Career Award (2012), DARPA Young Faculty Award (2013), TUM-IAS Hans Fischer Fellowship from Germany (2017), and ELATE Fellowship (2020).

Yiran Chen (Fellow, IEEE) received the PhD degree from Purdue University, in 2005. He is now a professor with the Department of Electrical and Computer Engineering, Duke University and serving as the director with the NSF AI Institute for Edge Computing Leveraging the Next-generation Networks (Athena) and the NSF Industry–University Cooperative Research Center (IUCRC) for Alternative Sustainable and Intelligent Computing (ASIC), and the co-director with Duke Center for Computational Evolutionary Intelligence (CEI). He is an ACM Fellow.