Theme Article: Machine Learning Acceleration

ERIDANUS: Efficiently Running Inference of DNNs Using Systolic Arrays

Bahar Asgari, Ramyad Hadidi, Hyesoon Kim, and Sudhakar Yalamanchili
Georgia Institute of Technology

***Abstract*—Systolic arrays with promising attributes, such as high degree of concurrent computation and high data-reuse rate, are attractive solutions for dense linear algebra. Recently, systolic arrays have been used for accelerating the inference of deep neural networks (DNNs). However, as sparsification mechanisms are applied to DNNs during or after training, DNN inference is usually a sparse problem. Therefore, it cannot fully benefit from the fundamental advantages offered by systolic arrays. To solve this challenge, we propose Eridanus, an approach to structured pruning that produces DNNs compatible with the synchronous and rhythmic flow of data from memory to systolic arrays.**

Systolic arrays are of increasing interest in accelerating linear algebra operations, and thus, deep neural network (DNN) inference [e.g., tensor processing unit (TPU)]. Systolic arrays are fine-grained, highly concurrent architectures that are distinctive in their approach for maximizing data reuse. Data reuse minimizes the memory bandwidth demand via interacting data flows in the compute arrays. As a result, systolic arrays work well for computing linear recurrences and *dense* linear algebra computations. However, DNN inferencing, which heavily relies on convolutional neural networks (CNN), is a *sparse* problem, because the close-to-zero weights are often being removed during or after training. Although pruning the individual weights reduces computation and memory footprints,[1,2] it creates irregular sparse models that cannot fully benefit from the fundamental advantages offered by systolic arrays.

To overcome these challenges, *structured* pruning techniques have been proposed, which prune the weights at various coarse granularities.[2–7] Table 1 and the "Prior Structured Pruning Work" section compare the prior structured pruning in details from various aspects and evaluate their

Digital Object Identifier 10.1109/MM.2019.2930057
Date of publication 25 July 2019; date of current version 10 September 2019.

Published by the IEEE Computer Society

Table 1. The impact of pruning granularity on inference using systolic arrays.

Convolution Layer Weights / Granularity	Element-wise	Shape-wise	Vector-wise	Kernel-wise	Filter-wise	Channel-wise	ERIDANUS
	$\mathbf{W}(f_1^{th}, f_2^{th}, c^{th}, k^{th})$	$\mathbf{W}(f_1^{th}, f_2^{th}, c^{th}, :)$	$\mathbf{W}(f_1^{th}, :, c^{th}, k^{th})$	$\mathbf{W}(:, :, c^{th}, k^{th})$	$\mathbf{W}(:, :, :, k^{th})$	$\mathbf{W}(:, :, c^{th}, :)$	$\mathcal{W}(m_1..m_2, k_1..k_2)$
Studies	[2, 3]	[6, 9]	[3, 4, 5]	[3, 4]	[3, 5, 6, 7, 8]	[4, 6]	Our Work
Pruning Opportunity	High	Low	Low	Low	Low	Low	Medium
Systolic Architecture: % Active Units	Low	Medium	Low	Low	Medium	Medium	High
Systolic Architecture: Storage Overhead	High	Medium	Medium	Low	Low	Low	Low
Systolic Architecture: Concurrency*	Low	Low	Low/Medium	Low/Medium	Medium	Medium	High
Systolic Architecture: Data Reuse	Low	High	Low	Low	Low	High	High
Systolic Architecture: No Buffering/Caching	No	No	No	No	No	No	Yes
Systolic Architecture: Indexing Complexity	High	Medium	Medium	Low	Low	Low	Low

* Whether or not the level of concurrency in the resulted pruned weight could be matched with that of the underlying systolic array.

effectiveness to be used for systolic arrays. In summary, the comparisons suggest that the preceding structured pruning techniques are advantageous in creating models suitable for central processing units (CPUs) and graphics processing units (GPUs) by reducing the number of operations and thereby decreasing latency. However, the key challenge is that they do not harness the full benefits (i.e., throughput and energy efficiency) of systolic arrays. This is because the conventional choices of pruning granularities (e.g., filter, kernel) used for CPU and GPU optimizations are not the most effective choice for pruning a model for systolic arrays. In other words, the outputs of the structured pruning algorithms do not match with the underlying data-reuse patterns and indexing functions in systolic arrays. Therefore, DNN inferencing either underutilizes the compute units or requires external buffering/caching, both of which cause performance loss from the best achievable performance gain. Our goal is to propose a pruning technique, the output of which is compatible with the data-reuse patterns in systolic arrays and streaming memory interface.

To achieve this goal, we produce DNN models, the nonzero values of which are clustered spatially into *locally dense* regions that can be compactly stored (i.e., less storage or metadata overhead) and efficiently *streamed* from memory. Our technique, Eridanus, applies structured pruning in a way that it maintains a systolic-streamlining access pattern. This is a major source of improvement for systolic arrays, and to reconstruct data-reuse patterns, which sustains throughput and eliminates buffering/caching. Eridanus consists of the following key insights:

- To capture the data reuse patterns in systolic arrays and enable data streaming, modifying the *distribution* of nonzero values is more influential than minimizing the number of operations or the memory footprint.
- To achieve an appropriate distribution of nonzero values, examining the correlation among the filters rather than the individual filters increases the chances of creating a systolic-friendly model.

Based on these insights, Eridanus prunes the two-dimensional (2-D) matrix operand of a matrix-matrix multiplication to capture the correlation among all the filters and to extract *locally-dense* blocks, the widths of which match the width of the target systolic array. We evaluate the accuracy, performance, and energy efficiency of Eridanus by pruning and training LeNet, CifarNet, and VGG16. We prune the models using Eridanus as well as the state-of-the-art structured pruning algorithms, and run the inference of all the pruned DNNs on systolic arrays.

MOTIVATION AND BACKGROUND

Systolic arrays with attractive advantages, such as a high degree of concurrent processing through a dataflow architecture, have become a key core of the hardware-accelerators for DNN[8] acceleration (e.g., TPU). A majority of computations in a DNN is done within the convolutional layers, the computation of which can be viewed as a matrix-matrix multiplication, or a general matrix multiply (GEMM) operation[5,8,9] that have straightforward systolic implementations—while this paper focuses on direct convolution implementations, we can extend this study to Winograd- and

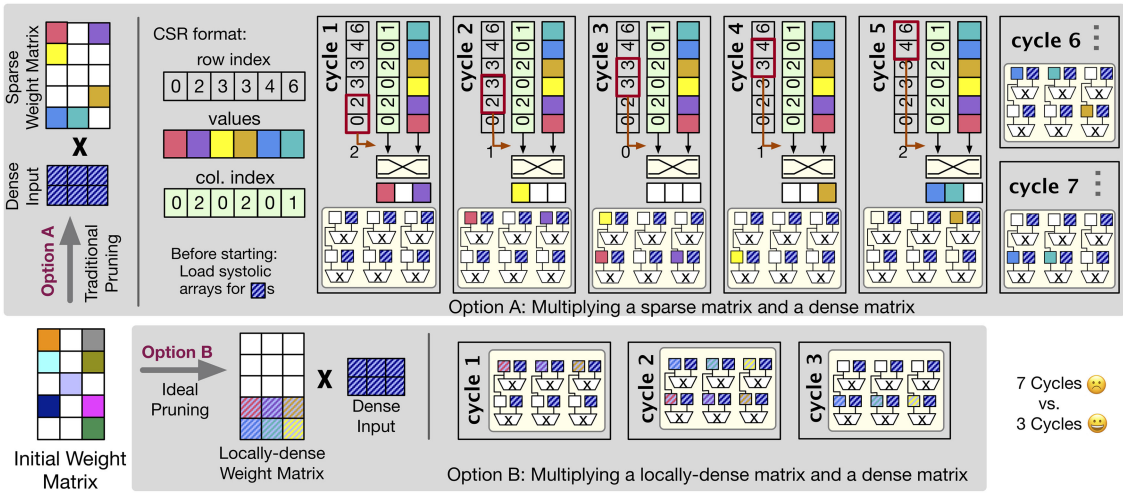

Figure 1. Comparing the execution of a matrix-matrix multiplication using a systolic array when one operand is a dense matrix and the other operand is: (a) a sparse matrix and (b) a locally-dense sparse matrix. See the details of microarchitecture in the "Microarchitecture" section.

FFT-based approaches[10], which are only beneficial for small kernels, with batched GEMM approach[11]. Convolving a 3-D input of size WxHxC (C: channels) with K filters of size FxFxC (i.e., a 4-D weight matrix of size FxFxCxK) results in an output of size WxHxK (assuming the same padding). The equivalent matrix-matrix multiplication multiples the equivalent 2-D weight matrix of size KxF^2C by the equivalent 2-D input matrix of size F^2CxWH as follows:

$$\mathbf{W}_{K\text{x}F^2C} \times \mathbf{I}_{F^2C\text{x}WH} = \mathbf{O}_{K\text{x}WH}. \quad (1)$$

Need for a Locally-Dense Model

Pruning is a common practice that reduces the computations and/or storage overhead of DNN models,[1] but makes them sparse. Sparse matrices, represented in formats, such as a compressed sparse row (CSR), necessitate indirect memory accesses and additional meta-data processing that limit effective utilization of systolic arrays. To clarify, Figure 1 compares using a 2×3 systolic array for first, multiplying a dense input matrix and a sparse weight matrix represented in CSR format (*option A*); versus second, multiplying the same dense input matrix and the locally-dense weight matrix pruned by an *ideal* structured pruning algorithm suited for systolic architecture (*option B*). Option A requires accessing extra meta data and reassembling a row before pushing it to systolic arrays, which has the following consequences: first, the number of cycles is defined by the number of rows (in the worst case) of the sparse matrix. Even if the hardware can skip zero rows, in the worst case, the nonzero values could be spread over all the rows. Second, at least one indexed read (the column index) per nonzero value causes inefficient memory accesses. Third, the compute units of the systolic array are poorly utilized (e.g., 28% on average). In contrast, in option B, the rows of the locally-dense weight matrix are streamed to the systolic array, without reading meta data. As a result, multiplications maximizes bandwidth utilization with less hardware complexity.

Prior Structured Pruning Work

As the "Need for a Locally-Dense Model" section showed, a *structured pruning* that creates a locally-dense model is beneficial to implement DNN inference on systolic arrays. We review the prior structured pruning algorithms and explains why they are not right solutions for systolic arrays. Structured sparsity learning[5] formulates shape-wise pruning as well as pruning in the granularity of kernel, filter, channel, and layer, and reported more than 3× speedups on CPUs and GPUs for the GEMM operations of CNNs while sustaining accuracy. Other efforts[7,3,2,6] have studied filter-wise pruning by applying various implementation techniques. Examples of filter-wise pruning are pruning the model based on the global rescaling of a criterion (e.g., the mean, standard deviation) for all layers[6], or selecting

close-to-zero weights based on the smallest sum of absolute values[7]. In another work, Scalpel[3] implements filter-wise pruning for hardware with high parallelism (e.g., GPU), weight grouping for single instruction, multiple data (SIMD), and a combination of both for hardware with moderate parallelism (e.g., CPUs). For efficient systolic implementations of sparse CNNs, column-combining approach[8] prunes all weights on conflicting rows for a selected set of columns except the one with the largest magnitude.

Table 1 evaluates using the prior pruning methods for systolic arrays. While element-wise pruning does not guarantee any spatial locality, vector-, and kernel-wise pruning, for example, do not capture the spatial locality, required by systolic arrays. Thus, not all the compute units of a systolic array are active during inference (see Figure 1). The storage overhead and indexing complexity that are defined by the granularity of pruning, is low (low is better) for kernel-, filer-, and channel-wise pruning. On the other hand, the opportunity of concurrency and data reuse are defined by the shape of the pruning granularity. For instance, row-wise proximity offers higher concurrency, whereas column-wise proximity captures more data-reuse patterns in systolic arrays. Although kernel-, filter-, and channel-wise pruning methods offer high level of concurrency, they might not match the concurrency required by the algorithm. Finally, since none of the pruning methods have the same granularity width as a systolic array has, they all require some sort of buffering/caching mechanisms to enable efficient streaming from memory. In addition to such a hardware complexity, ensuring correct timing to supply data from memory to the compute units of systolic arrays requires complex indexing hardware when the granularity of pruning is small. Note that, according to how the compiler orders the elements of weight matrices, the listed granularities in Table 1 could be inferred as similar implementations, but they still do not capture the specific structured pruning, required by systolic arrays.

Challenges

Despite the advantages of the prior structured pruning algorithms for CPUs and GPUs, the following challenges exist for implementing DNN inference on systolic arrays. First, the structured pruning methods help in reducing the number of operations and memory footprint. However, such optimizations alone are insufficient to exploit the highly concurrent, synchronous, and rhythmic flow of data from memory through the systolic array. In other words, systolic arrays require optimization for data streaming. Further, a pruning algorithm for other concurrent hardware (e.g., SIMD) is not applicable for systolic arrays because they do not capture dependencies in data to satisfy the *data reuse* patterns of systolic arrays. The second challenge is that computation results of pruned model should be compatible with the streaming memory interface for eliminating extra buffering/caching. The storage adjacency of data resulting from algorithm-defined granularity (e.g., kernel, filter) does not match the data organizations necessary to directly stream the interacting data flows to the systolic array.

ERIDANUS

To efficiently use systolic arrays for multiplying the two 2-D matrices in (1), we propose Eridanus. Eridanus is an approach to structured pruning that organizes the weight matrix such that the nonzero values are placed in close proximity in a manner that their computations are related. Eridanus enables streaming of locally-dense matrix from memory and exploits the distinctive data reuse patterns and fine-grained concurrency in systolic arrays.

Pruning Algorithm

In contrast to other pruning algorithms, instead of the individual filters, Eridanus examines and prunes the flattened weight matrix by extracting the potential nonzero blocks, the widths of which are matched with the width of the target systolic array. The blocks are created by first, hypothetically splitting the weight matrix into F^2C/ω chunks (F^2C: the common axis of input and weight matrix, ω: the width of systolic array), and second, extracting the nonzero blocks in each chuck. While the widths of the blocks must match the widths of the systolic arrays to guarantee the correctness of multiplications, their length could be arbitrary. However, to reduce the complexity of the algorithm, we fix the length too. Based on trials, we choose eight as the length, which offers the best tradeoff between sparsity of blocks and storage overhead. Once the blocks are extracted, the adjacent ones are concatenated and stored as a single block by assigning it a single index (i.e., the column index of the first block) and a single length.

Algorithm 1[12] illustrates the pruning algorithm used in Eridanus, the input parameters of which are the weight matrix W, threshold θ, length of the window l (a hyperparameter), and width of the systolic array ω. The width of the window is fixed and is equal to ω. The weight matrix is either the flattened version of the weight matrix in a convolution layer or the 2-D weight matrix itself in a fully-connected layer. During pruning, a window of size $\omega \times l$ slides over W. If the average value of the window (Line 3) is smaller than θ, the block corresponding to that window is set to zero (Line 3). During retraining, by increasing θ in later epochs, the algorithm maintains the accuracy and convergence. In other words, the threshold of the average values for choosing/pruning the zero blocks are gradually increased with training epochs.

As Algorithm 1 does not change the size of the common axis of the operands (i.e., F^2C), it has the following benefits: first, the input matrix does not require modifications, and second, both the pruned weight matrix and the dense input matrix can either be streamed through the systolic array or be stationary during the multiplication. Therefore, based on the relative size of the matrices at each layer, we dynamically swap the role of the two matrices (i.e., the larger is streamed and the smaller is stationary).

Algorithm 1. Pruning

1: **function** Prune($W_{h\text{x}w}, \theta, l, \omega$)
   $W_{h\text{x}w}$: Weight matrix, θ: Threshold,
   ω: Systolic array width l: Window length
2: $i_h := 0, i_w := 0, avg := 0$
3: **while** $i_w < w$ **do**
4: avg = **BlockAvg**($[i_w, i_h], [i_w + \omega - 1, i_h + l - 1]$)
5: **if** $avg < \theta$ **then**
6: $W[i_w : i_w + \omega - 1, i_h : i_h + l - 1] = 0$
7: $i_h = i_h + l$
8: **else**
9: $i_h = i_h + 1$
10: **end if**
11: **if** $i_h > h - l$ **then**
12: $i_h = 0$
13: $i_w = i_w + \omega$
14: **end if**
15: **end while**
16: **end function**

Microarchitecture

Systolic Engine We use a weight-stationary systolic array with one streaming (i.e., *R1*s in Figure 2) and one stationary (i.e., *R2*s in Figure 2) input. The streaming input registers are connected in a column. At each cycle, their contents shift one row down ❶. The stationary registers are also connected to simplify the interconnection between the array and memory. The input from memory is connected to the first row. The stationary values swing through registers until they reach their destination register, where they settle. The streaming registers and the stationary registers share memory bandwidth to obtain their contents. At each cycle, all multipliers are active. The outputs of a row are summed through an adder tree to contribute to create an element of the output. The number of adder trees (i.e., the width of the systolic array) defines the number of output elements generated at each cycle.

The width of the systolic array defines the degree of concurrency. Thus, normally, we want the width to match the width of the 2-D-weight matrix to maximize the fine-grained parallelism. However, to be flexible and scalable, we prefer to employ several narrow systolic arrays, instead of one large array, so that based on the size of a weight matrix, we assign as many narrow systolic arrays as required. The depth of the systolic array directly impacts the data-reuse rate. Thus, a deeper systolic array is preferred. However, pruning causes variations in the length of the locally-dense blocks, which are going to stay in the systolic array. Therefore, while choosing a large depth for the systolic array leads to underutilization of the systolic array, a very small depth prevents achieving peak throughput. As a result, to optimize for the common case, we choose a depth of 64.

Memory Management To maximize bandwidth utilization and avoid random memory accesses, we map locally-dense weights and the inputs corresponding to sequential multiplications in the sequential addresses of memory. Therefore, for each layer, the stationary operand is streamed, followed by the streaming operand. The *type* identifier (that indicates stationary and streaming data) is used to direct data to the right registers. The pruned weight of each layer is stored as locally-dense blocks. The header of each block includes the index, the length, the type of data, and an

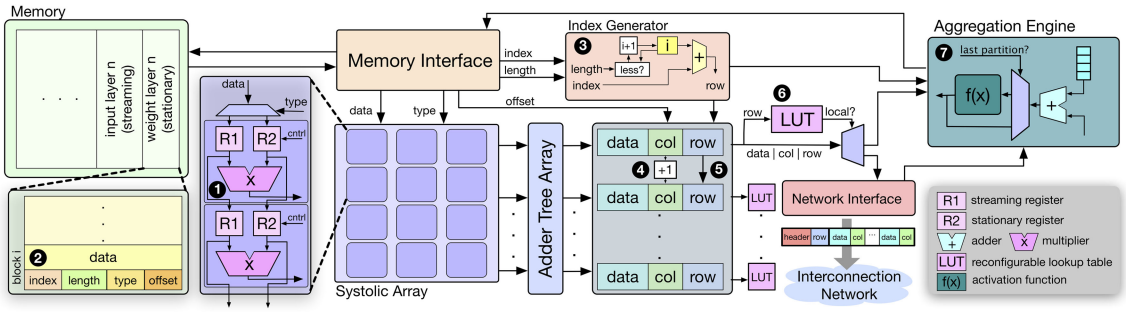

Figure 2. Overview of the systolic-based microarchitecture, used in Eridanus.

offset ❷. When the width of the stationary operand matrix is larger than the depth of the systolic array, we split a multiplication into submultiplications. The *offset* is the index of the submultiplications and is used to generate the column-index of the output elements. The memory interface reads from memory and directs *data* and its *type* to the systolic array, and sends the *index*, *length*, and *offset* of blocks to the index generator ❸.

Indexing In multiplying $W_{K\times F^2C}$x$I_{F^2C\times WH}$, the column and row indices of the output elements are defined by the column index of I and the row index of W, respectively. Therefore, the index of an adder tree simply indicates the column index of the output elements—the offset will also be added to it if the matrix does not fit in the systolic array. This is implemented by the increment units between the column indices ❹. The block index and length indicate the row within the selected column of the output. As the row indices of the output elements corresponding to a single block are sequential, the row indices are reused by shifting them down ❺.

Handling Larger Matrices When $F^2C > \omega$, more than one systolic array will contribute to calculating an element of the output. As a result, the partial results will have to be aggregated in the final destination (i.e., based on the mapping of the next layer of DNN across the systolic array). The mapping of the submultiplications to the systolic arrays is programmed in a look-up table (LUT) ❻. Once the column and row indices are assigned to the outputs of the adder trees, based on the LUT, they will be directed to other systolic arrays or to the aggregation engine of the current one. The aggregation engine ❼ sums the partial results, and if the current partial result is the last portion of the output element, it applies the activation function to the final result and sends it to the memory interface to be written in memory. If an element of output is not local, it is directed to the network interface. We use a multidrop express channels (MECS) topology, a bandwidth-efficient interconnection network. The elements with the same destination, which also have common row index, are packetized together. When a packet is received, it is de-packetized and sent to the aggregation engine.

Locally-Dense Matrix Multiplication Using Systolic Arrays

To illustrate the functionality of the systolic array, when $F^2C > \omega$, Figure 3 shows the steps of multiplying matrix W and I and creating output matrix O, in which $F^2C = 9$ and the size of the systolic array is 3×3. Since $F^2C > \omega$ (i.e., $9 > 3$), we need to split the main multiplication into three submultiplications [Figure 3(a)]. Depending on the number of available systolic arrays, the submultiplications can be performed sequentially or in parallel. Figure 3(b)–(d), respectively, illustrate using one, two, and three systolic arrays for performing the three submultiplication. During each submultiplication, a partition of I is stationary in the systolic array. Before a submultiplication starts, its corresponding stationary operand is loaded to the systolic array. Because of the arrangement of the registers (explained in the "Systolic Engine" section), loading the stationary operand takes three cycles (i.e., in general, equal to the depth of the systolic array). Once the stationary operand is settled, the blocks of W sequentially pass through the systolic array and generate the elements of output. Since in this example, the lengths of the locally-dense blocks are

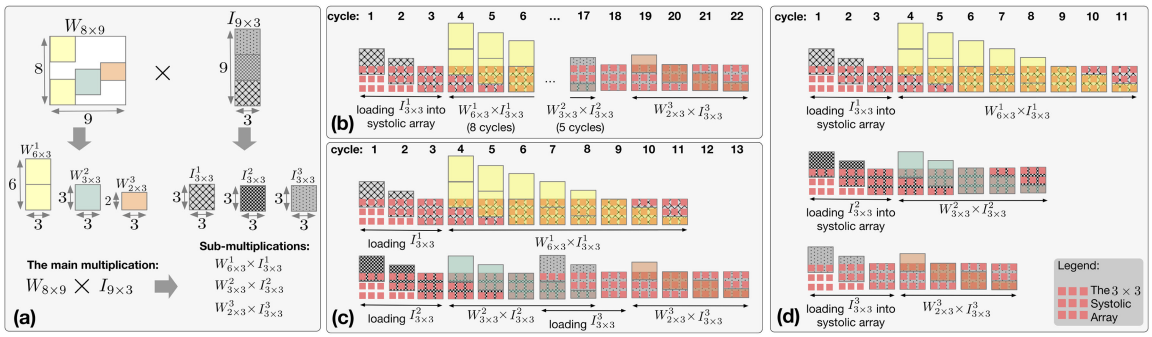

Figure 3. Example of multiplying a locally-dense matrix $W_{8\times9}$ and a dense input matrix $I_{9\times3}$, by using 3×3 systolic arrays. (a) Since the common axis of W_{8x9} and $I_{9\times3}$ (i.e., 9) is larger than the width of systolic array, we perform three submultiplications. We compare the execution time when using (b) one systolic array, in which three submultiplications are sequential and are done in 22 cycles, (c) two systolic arrays, in which W^1xI^1 is performed in parallel with W^2xI^2 and W^3xI^3 and result in 13 cycles, and (d) three systolic arrays to perform all three submultiplications in parallel in 11 cycles.

varied, the execution time is defined by the longest submultiplication, and hence, using two or three systolic arrays result in almost similar latencies.

EVALUATION

Experimental Setup

We use Tensorflow for iteratively training and pruning three DNNs including VGG16, CifarNet, and LeNet on ImageNet, Cifar10, and MNIST datasets, respectively. For VGG16, we use a pretrained model, whereas for LeNet and CifarNet, which are smaller, we start training from scratch. We compare the performance of Eridanus against state-of-the-art structured pruning algorithms[2–7] with granularities of shape, vector, kernel, filter, and channel, as well as element-wise pruning[1] (see Table 1). The accuracy of the models pruned by various methods is the same, whereas their sparsity vary. We run the inference of all pruned models on the systolic-based engine. Seeking fair comparison, each baseline pruning method is compiled based on its best format. When required, additional buffering/caching mechanisms are implemented. The pruned models are used as the input to our in-house cycle-level simulator, which models the microarchitecture shown in Figure 2. We use the version 1 of high bandwidth memory (HBM) as the memory connected to 8×64 systolic arrays. We estimate the power consumption of the compute units by using Kitfox1.1 library at 16-nm technology and McPAT model. We assume the access energy per bit of 6 pJ/bit for HBM. We connect eight of the modules shown in Figure 2 in a MECS topology. On an average, a packet consumes 0.52 nJ energy at routers and links. The latency of each multiplier is three cycles @2 GHz. We process batches of size 16. By choosing a relatively small batch size, we neither increase the number of reloads at mid-size layers, nor destroy the compute utilization at fully-connected layers.

Accuracy

Figure 4(a) illustrates the top-1 accuracy of the CifarNet and VGG16 pruned by Eridanus, normalized to the accuracy of the unpruned model, along with the average percentage of zero blocks. For CifarNet [Figure 4(a)], pruning is applied between steps 20k and 100k. For VGG16, since we use a pretrained model, we start pruning from the beginning (i.e., step 1 to 10k). As Figure 4(a) shows, during pruning, the percentage of zero blocks increases. However, since the distribution of zero blocks and/or their densities keep changing, the accuracy oscillates. After pruning stops, training continues to maximize the accuracy by adjusting the values of nonzero blocks. For LeNet, CifarNet, and VGG16, we prune 75%, 79.8%, and 42% of models and, respectively, achieve 99%, 93.6%, and 70% top-1 accuracy on validation set. The top-1 accuracy of unpruned models are 99% for LeNet, 94% for CifarNet, and 71.5% for VGG-16. Note that parameters, such as threshold (θ in Algorithm 1), the start and the end steps, the

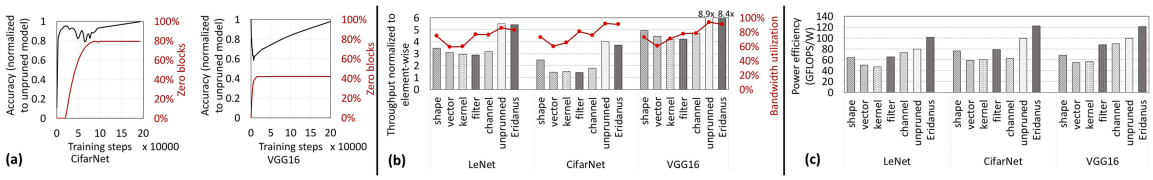

Figure 4. (a) The accuracy and the percentage of zero blocks for CifarNet (pruned between steps 20k and 100k) and VGG16 (pruned between steps 1 and 10k). (b) The comparison of throughput and bandwidth utilization. (c) The power efficiency of three DNN models pruned by various structured techniques.

length of the sliding window, and the maximum desirable sparsity, impact the tradeoff between accuracy and zero distribution.

Performance

Throughput and Memory Bandwidth Utilization

We evaluate the speed of inference on systolic arrays by the relevant metric of throughput, rather than inference time. Parameters, such as the level of concurrency, the number of times we need to load the stationary operands, the additional caching/buffering/decoding for directing data from memory into the systolic array, and memory bandwidth utilization impact throughput. Bandwidth utilization depends on the number of indirect memory accesses, and density of the models. As a result, the unpruned (i.e., dense) model is expected to have the highest throughput and bandwidth utilization comparing to structured models. Figure 4(b) shows the throughput and bandwidth utilization of DNN inference on systolic arrays for the models pruned with various granularities. The trend of the overall performance is similar to that of the throughput since the amount of computation remains similar.

As Figure 4(b) illustrates, the bandwidth utilization and throughput of Eridanus, which optimizes all the effective parameters (listed in Table 1) together, is very similar to those of the unpruned models. The other structured pruning approaches, however, are not as effective as Eridanus, because they are not jointly optimized for capturing the data-reuse patterns and concurrency offered by systolic array, and hence, incur some overheads. For instance, kernel-wise captures less data-reuse patterns, whereas it enables a high level of parallelism. On the other hand, compared to other baseline structured models, shape-wise and channel-wise yield a better performance on systolic arrays, because they can capture more data-reuse patterns. However, they limit the level of concurrency. Although the number of operations in the models, created by Eridanus could be more than those in irregular sparse models (e.g., element-wise), the locality in the structured model of Eridanus leads to lower latency. As a result, the combination of fast computation and high bandwidth utilization leads Eridanus to work more closely to the peak throughput.

Power Efficiency

The power consumption of running inference on systolic array is defined by the number of memory accesses as well as the number of operations. Comparing to other pruning approaches, Eridanus creates a model that requires the lowest number of memory accesses. However, the effect of pruning algorithms on the number of computations is the opposite. Comparing to element-wise pruning, the structured pruning approaches may require more number of operations (i.e., because of more number of nonzero values). On the other hand, the systolic array executes spatial operations more quickly than sparse operations. As a result, the ratio of memory-access reduction to compute-density reduction is the key factor in defining the power efficiency. Figure 4(c) illustrates the combined effect of the number of memory accesses and computation density, on power efficiency. As the figure shows, for Eridanus, the reduction in memory accesses carries more weight and helps achieve higher power efficiency.

CONCLUSION

This article proposed Eridanus, a novel approach to structured pruning of DNNs based on the requirements of systolic arrays. Eridanus emphasized the importance of the distribution

of nonzero values in sparse DNN models when we implement them on systolic arrays.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the support of the National Science Foundation NSF CCF under Grant 1533767.

REFERENCES

1. 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*.
2. H. Mao *et al.*, "Exploring the regularity of sparse structure in convolutional neural networks," 2017, *arXiv:1705.08922*.
3. J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing DNN pruning to the underlying hardware parallelism," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 548–560, 2017.
4. S. Anwar *et al.*, "Structured pruning of deep CNNs," *J. Emerg. Technol. Comput. Syst.*, vol. 13, no. 3, 2017, Art. no. 32.
5. 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.
6. P. Molchanov *et al.*, "Pruning convolutional neural networks for resource efficient inference," 2016, *arXiv:1611.06440*.
7. H. Li *et al.*, "Pruning filters for efficient convnets," 2016, *arXiv:1608.08710*.
8. 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.
9. S. Chetlur *et al.*, "cuDNN: Efficient primitives for deep learning," 2014, *arXiv:1410.0759*.
10. A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 4013–4021.
11. U. Koster and S. Gray, "Going beyond full utilization: The inside scoop on Nervana's Winograd kernels," 2019. Accessed: Apr. 16, 2019. [Online]. Available: www.intel.ai/winograd-2/#gs.61t136
12. B. Asgari, R. Hadidi, H. Kim, and S. Yalamanchili, "LODESTAR: Creating locally-dense CNNs for efficient inference on systolic arrays," *Proc. 56th Annu. Des. Autom. Conf.*, 2019, Art. no. 233.

Bahar Asgari is working toward a PhD at the School of Electrical and Computer Engineering, Georgia Institute of Technology, and is a member of the Computer Architecture and System Laboratory. As a graduate research assistant under the supervision of Prof. Sudhakar Yalamanchili and Prof. Hyesoon Kim, she conducts research in the field of computer architecture. Her research interests include but are not limited to accelerating sparse problems and deep neural networks, and scalable memory systems. Contact her at: bahar.asgari@gatech.edu.

Ramyad Hadidi is currently working toward a PhD in computer science under the supervision of Prof. Hyesoon Kim at Georgia Institute of Technology. He received the bachelor's degree in electrical engineering from Sharif University of Technology and the master's degree in computer science at Georgia Institute of Technology. His research interests include but are not limited to computer architecture, edge computing, and machine learning. Contact him at: rhadidi@gatech.edu.

Hyesoon Kim is an associate professor with the School of Computer Science, Georgia Institute of Technology. Her research areas include the intersection of computer architectures and compilers, with an emphasis on heterogeneous architectures, such as GPUs and accelerators. She has a PhD in electrical and computer engineering from the University of Texas at Austin. She is a member of the IEEE. Contact her at: hyesoon.kim@gatech.edu.

Sudhakar Yalamanchili was a Regents Professor and Joseph M. Pettit Professor of computer engineering with the School of Electrical and Computer Engineering, Georgia Institute of Technology. He has a BE in electronics from Bangalore University, India, and a PhD in electrical and computer engineering from the University of Texas at Austin. Prior to joining Georgia Tech in 1989, he was a Senior and then Principal Research Scientist at the Honeywell Systems and Research Center in Minneapolis. He is a member of the ACM, and an IEEE Fellow. Contact him at: sudha@gatech.edu.