Procrustes: a Dataflow and Accelerator for Sparse Deep Neural Network Training

Dingqing Yang[†], Amin Ghasemazar[†*], Xiaowei Ren[†*], Maximilian Golub[‡], Guy Lemieux[†], and Mieszko Lis[†]
[†]The University of British Columbia [‡]Microsoft Corporation
{dingqingy,aming,xiaowei}@ece.ubc.ca, magolub@microsoft.com, {lemieux,mieszko}@ece.ubc.ca

***Abstract*—The success of DNN pruning has led to the development of energy-efficient inference accelerators that support pruned models with sparse weight and activation tensors. Because the memory layouts and dataflows in these architectures are optimized for the access patterns during *inference*, however, they do not efficiently support the emerging sparse *training* techniques.**

In this paper, we demonstrate (a) that accelerating sparse training requires a co-design approach where algorithms are adapted to suit the constraints of hardware, and (b) that hardware for sparse DNN training must tackle constraints that do not arise in inference accelerators. As proof of concept, we adapt a sparse training algorithm to be amenable to hardware acceleration; we then develop dataflow, data layout, and load-balancing techniques to accelerate it.

The resulting system is a sparse DNN training accelerator that produces pruned models with the same accuracy as dense models without first training, then pruning, and finally retraining, a dense model. Compared to training the equivalent unpruned models using a state-of-the-art DNN accelerator without sparse training support, Procrustes consumes up to 3.26× less energy and offers up to 4× speedup across a range of models, while pruning weights by an order of magnitude and maintaining unpruned accuracy.

I. Introduction

Deep neural networks are known to be vastly overparameterized: pruning techniques can typically reduce the weight count by an order of magnitude [14, 15, 16, 25, 28, 29, 43, etc.]. This sparsity comes at the cost of irregular memory accesses and computation patterns, and several accelerators have been proposed to enable efficient inference on sparse models [7, 11, 13, 36, 50, etc.].

None of these approaches were, however, designed for energy-efficient *training*. This is because they target a context where pruning occurs *after* training: a model is first trained with the full parameter set, then pruned, and finally re-trained to recover accuracy [15]. While this saves energy at inference time, training the pruned network takes *more* time and energy than training an equivalent dense network to the same accuracy. Skipping the pre-training step is not an option: even if oracular knowledge of the pruned model connectivity is assumed, training the pruned model from scratch sacrifices accuracy compared to the original network [15, 28].

Still, the very existence of pruned networks suggests that it must be possible to somehow train them. Recent work has demonstrated that a model pruned by an order of magnitude can be trained *provided* that the initialization for the unpruned subset of weights is preserved [8]; this can be achieved either by dynamically selecting the most productive gradient subspace [10] or by iteratively increasing sparsity [33, 49].

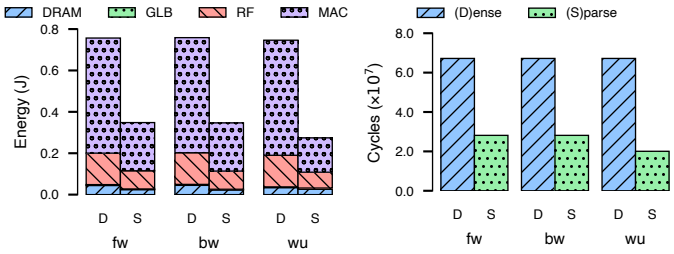


Fig. 1: Potential training energy savings and speedup from ideally leveraging all weight sparsity (here, 5×) while training VGG-S (15M weights) to convergence with Dropback [10]. fw/bw/wu = forward/backward/weight-update phases.

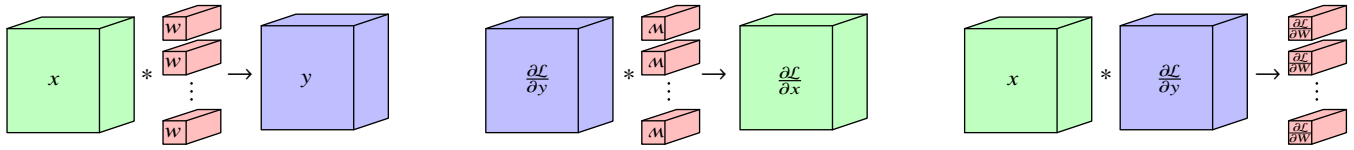
Ideally, such sparse-from-scratch training can offer significant savings. Figure 1 shows this for VGG-S [46] pruned 5× (15M→3M weights) using the Dropback algorithm [10], in an idealized 16×16 PEs training system where (i) sparsity is evenly distributed within each layer so all PEs receive the same workload (i.e., perfect load balancing), and (ii) sparse weights are stored in an idealized compressed format with no overhead, and (iii) retained weights selection is instant and cost-free (see Section VI-A for setup details). While the exact improvement varies with the geometry and sparsity of each layer, leveraging 5× sparsity can yield up to 2.6× speedup with 2.3× less energy consumption over the entire network.

In practice, however, none of the existing sparse training methods can reach this potential. Most [8, 10, 49] require sorting all weights to determine the parameters to retain; with weight counts in the tens of millions, sorting is an expensive proposition. Several [33, 49] achieve only small pruning factors and suffer accuracy loss. Some [8, 49] prune the model very gradually; this implies (i) no peak memory footprint reduction, (ii) mediocre energy savings because the average sparsity is low during most of the training process, and (iii) the need to support two weight storage formats (dense and sparse) and switch formats mid-way during training. The remaining technique [10] maintains the target weight sparsity throughout training, but gives up computation sparsity — a significant drawback for training, where weights are usually 32-bit floating-point numbers that are energetically expensive to multiply.

In addition, existing accelerators that support sparse inference

[*]These authors contributed equally.
[‡]Work done while the author was with the University of British Columbia.

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00064



(a) forward: $x * W \rightarrow y$ (b) backward: $\frac{\partial \mathcal{L}}{\partial y} * W^{\circlearrowleft} \rightarrow \frac{\partial \mathcal{L}}{\partial x}$ (c) weight update: $x * \frac{\partial \mathcal{L}}{\partial y} \rightarrow \frac{\partial \mathcal{L}}{\partial W}$

Fig. 2: CNN training consists of (a) the forward pass, (b) the backward pass, and (c) the weight update pass; minibatch size adds a fourth dimension to the activations. Weights are accessed in different order during the forward and backward passes. Training FC layers is similar but uses multiplication instead of convolution and $W^{\top}$ instead of $W^{\circlearrowleft}$ in the backward pass. $\mathcal{L}$ = loss; x = iacts; y = oacts; W = weights; $\circlearrowleft$ = 180° filter-wise rotation.

are inadequate for sparse training. Weights are represented in formats that directly correspond to the dataflow being used [7, 11, 13, 36, 50, etc.]; this works well when weights are always accessed in the same order during inference, but does not support the different weight access patterns that arise in different phases of training (see Section II-A). Accelerators that perform load balancing (e.g., Sparten [11]) do this in software as a preprocessing step; this works for inference where weight sparsity is static, but not for training where weight sparsity changes dynamically. Finally, recent proposals like SCNN [36] and Sparten [11] use complex hardware to exploit two-sided sparsity (i.e., both weight and activation sparsity); this can be leveraged during the forward-pass phase of training, but usually does not exist in the backpropagation or weight update phases because the ubiquitous batch normalization destroys layer sparsity in the back-propagated gradient $\frac{\partial \mathcal{L}}{\partial y}$, so the additional hardware costs are not warranted for training. (We describe these challenges in more detail in Section II.)

In this paper, we tackle the challenges of accelerating sparse training by combining algorithmic adaptation with dataflow and hardware optimizations. The accelerator architecture we propose, Procrustes, relies on four key insights:

1) Two-sided sparsity can only be leveraged in the forward pass, but increases interconnect complexity [7, 36, etc.]. Procrustes therefore exploits one source of sparsity in each training phase: weight sparsity in the forward and backward passes, and activation sparsity in the weight update phase. This maximizes energy and latency improvements while minimizing hardware complexity.
2) While load-balancing a sparse workload across a 2D PE array can destroy spatial reuse, spatial reuse generally arises in only one hardware dimension (either row or column broadcast). Procrustes uses dataflows that distribute the non-sparse minibatch dimension (always available during training) across one hardware dimension and the sparse tensor dimension(s) across the other hardware dimension, load-balancing the workload across the minibatch dimension. This achieves good utilization and preserves spatial reuse without a complex interconnect.
3) While sparse training approaches generally rely on sorting to determine which weights to keep, it actually suffices to *partition* the weight set into two sets (retained and discarded). Procrustes replaces the sorting with a partitioning scheme based on dynamic quantile estimation [45], which avoids the computation and storage overheads of sorting.
4) Weight initialization values are only important during early phases of training and quickly outweighed by the accumulated gradients. Therefore, in sparse training algorithms where retained initial weight components prevent computation sparsity [10], the initial weights can be decayed to zero early in the training process.

In the remainder of the paper, we first show how to adapt an existing sparse training algorithm [10] to make it suitable for hardware accelerator implementation; the adapted algorithm achieves 3.9×–11.7× sparsity while maintaining unpruned accuracy on tasks like CIFAR10 and ImageNet.

We then propose a hardware architecture that adapts a standard 2D-PE-array inference accelerator to enable sparse training without incurring the dataflow limitations and interconnect complexity of the only prior sparse training accelerator proposal [49] and achieves much higher sparsity. Finally, we develop a sparse data representation suitable for training access patterns, and an inexpensive load-balancing technique that preserves maximum spatio-temporal reuse without complicating the on-chip interconnect. Most of the modifications are not specific to the sparse training method we adapt, but rather are necessary for accelerating any existing sparse training approach.

Compared to an equivalent accelerator that does not support training-time sparsity, Procrustes uses 2.27×–3.26× less energy and offers 2.28×–4× speedup without compromising accuracy on state-of-the art networks on ImageNet and CIFAR-10.

II. Sparse training considerations

A. DNN training

Stochastic gradient descent (SGD) — the de facto standard training algorithm for deep neural networks [26] — comprises three stages, illustrated in Figure 2:

1) The *forward pass* runs the inference algorithm to determine the model's predictions for training inputs and calculate the loss $\mathcal{L}$ (i.e., the training error). For a convolutional layer, this consists of convolving the input activation (iact) tensor x with a set of filters w to obtain the output activation (oact) tensor y (Figure 2a).
2) The *backward pass* back-propagates the loss gradient across the model's layers. For a convolutional layer, this is done by convolving the loss gradient with respect to

```
for n ∈ [0, N) do                          ▷ minibatch
  for r ∈ [0, R) do                        ▷ filter x-dim
    for s ∈ [0, S) do                      ▷ filter y-dim
      for p ∈ [0, P) do                    ▷ oacts x-dim
        for q ∈ [0, Q) do                  ▷ oacts y-dim
          for c ∈ [0, C) do                ▷ in channel
            for k ∈ [0, K) do              ▷ out channel
              y[p, q, k, n] += w[r, s, k, c]
                             × x[p+r, q+s, c, n]
```

Alg. 1: The computation of a CONV layer forward pass.

the oacts $\frac{\partial \mathcal{L}}{\partial y}$ with filters w; unlike in the forward pass, however, each filter is first rotated 180° (Figure 2b).

3) The *weight update pass* determines how much a weight w should be adjusted to decrease the loss by computing the gradient $\frac{\partial \mathcal{L}}{\partial w}$. For a convolutional layer, this consists of convolving the backpropagated loss gradient with respect to the oacts $\frac{\partial \mathcal{L}}{\partial y}$ with the input activations (iacts) x (Figure 2c).

In fully connected layers, x and y are 1D vectors, a weight matrix replaces the weight filters, inner product replaces convolution, and matrix transpose replaces the 180° rotation.

B. Sources of sparsity

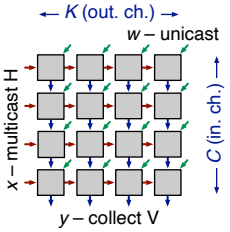
Inference accelerators that support sparsity [7, 11, 13, 36, 50, etc.] can leverage two sparsity sources: (a) zero-valued weights that result from pruning [15], and (b) zero-valued activations that result from the RELU activation function [2]. With suitable hardware support, multiply-accumulate (MAC) operations that involve zero weights or activations can be skipped, while zero-valued weights and activations need not be stored if a suitable sparse data format is used; some accelerators can take advantage of both sparsity sources simultaneously [7, 11, 36].

During training, weight sparsity can also be used in the backward gradient propagation phase, and input activation sparsity in the weight update phase (cf. Figure 2). However, the back-propagated gradient $\frac{\partial \mathcal{L}}{\partial y}$ does not exhibit sparsity because of the prevalent use of batch normalization [20]: batch normalization layers are commonly used between CONV and RELU layers, which means that the $\frac{\partial \mathcal{L}}{\partial y}$ sparsity generated from backpropagating through RELU is destroyed by backpropagating through the batch normalization layer.

Designers of sparse training accelerators, therefore, are faced with a choice: either spend additional hardware to accelerate one third of the training process, or reduce hardware complexity but give up on leveraging activation sparsity in the forward pass. In this paper, we focus on the latter approach.

C. Mappings, dataflows, and load balancing

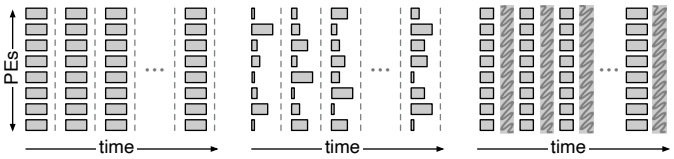
Algorithm 1 illustrates the operations required to evaluate a CONV layer (the forward pass is shown but the backward and weight update passes can be expressed in the same way, with only the innermost MAC operation different). The computation can be represented as a seven-dimensional nested loop, where each loop traverses a different dimension of the operation space [35] (single-sample inference accelerators may not have the N minibatch dimension). Regions of this operation space are then distributed as "work tiles" to different PEs by mapping two of the loops to the horizontal and vertical dimensions of a 2D PE array; together with exchanging the order in which loops are nested, this determines the dataflow [24, 35].



	fw	bw	wu
(H)orizontal	x	$\partial\mathcal{L}/\partial x$	x
(V)ertical	y	$\partial\mathcal{L}/\partial y$	$\partial\mathcal{L}/\partial y$
(U)nicast	w	w	$\partial\mathcal{L}/\partial w$

w weights x input activations
y output activation partial sums

Fig. 3: A weight-stationary mapping: input and output channel dimensions (C and K) are distributed spatially.



(a) dense (b) sparse +bcast (c) sparse –bcast

Fig. 4: DNN computation on a 2D PE array with a weight-stationary C, K mapping: (a) dense model, equal work and spatial reuse; (b) sparse model, unequal work but spatial iact/psum reuse; (c) sparse model, equal work but no spatial iact/psum reuse. ▭ = work tile; ▨ = overhead due to lack of reuse / complex interconnect. Each column is a full PE array's worth of work; a single layer's computation comprises many of these. ±bcast = with/without spatial weight reuse.

Figure 3 shows the ubiquitous weight-stationary dataflow [3, 6, 22, 34, 39, 40, 41, 48, etc.], which results from mapping the C, K dimensions across the PE array in the forward pass (mapping R, S is less common due to small filter sizes); the corresponding mappings for the backward and weight-update passes are shown in the adjacent table.

In this mapping, each workload (e.g., DNN layer) is first divided into PE-sized work tiles, all of which have the same number of weights. The tiles are mapped among the PEs; once PE receives one work-tile, the computation begins and runs until *all* work-tiles have finished. Finally, the next set of work-tiles is distributed among the PEs, and the process repeats until the entire layer has been evaluated (Figure 4a).

This mapping results in advantageous dataflow properties in a 2D PE array. Because all work tiles have the same amount of work, execution is naturally synchronized, and data can be spatially reused by broadcasting across multiple PEs. For example, in the forward pass in Figure 3, input activations are broadcast horizontally (read-only reuse), while partial sums are reduced vertically (read-write reuse). The dataflow patterns also

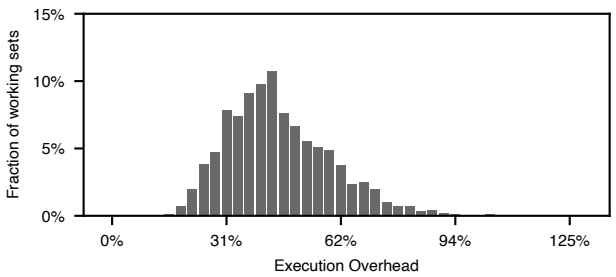


Fig. 5: Load imbalance histogram of full-PE-array working sets (columns in Figure 4b) when training VGG-S [46]/CIFAR-10 [23] using Dropback sparse training [10]. A perfectly load-balanced workload would have 100% of the sets at 0% overhead.

allow the on-chip network to be simple: our example requires two one-dimensional flows (for the activations and the partial sums) and one unicast flow (for the weights); typically, those would be three separate interconnects.

However, difficulties arise when the network is sparse. At reasonable pruning levels, on the order of 10% of the weights survive [15], with sparsity distributed unevenly among the worktiles (by chance and learning pressure). This leaves designers with two unpleasant alternatives:

1) Retain the same tiling and mapping of operations to the PE array as in the dense case. This preserves the single-dimensional dataflow patterns shown in Figure 3, allowing input activations and partial sums to be spatially reused. However, different amounts of work are distributed to different PEs, and utilization is low because latency is limited by the "slowest" PE (Figure 4b). Figure 5 shows how latency differs among full-PE-array sets of work tiles (i.e., columns in Figure 4b): frequently, the load imbalance causes execution time overheads in excess of 50%, and sometimes in excess of 100%.
2) Distribute an equal number of non-zero weights to each PEs. This balances the workload among the PEs (Figure 4b), but destroys the desirable single-dimensional on-chip traffic flow patterns of Figure 3 and severely reduces the benefits from spatial reuse. In addition, because related partial sums can be generated in any PE, a complex interconnect is required to reduce them [7, 49].

Choosing other dataflows also does not provide a panacea: for example, the activation-stationary dataflow used for some sparse accelerators [36] suffers similar issues in the weight update pass, requires two datatypes to be unicast, and suffers from low PE array utilization towards the tail of many networks where the activation tensors are small [7].

Section IV-C describes how Procrustes employs the additional minibatch dimension available during training to achieve effective load balancing while preserving a hardware-friendly dataflow and avoiding the need for a complex interconnect.

D. Sparse weight representation

Existing sparse-weight inference accelerators [7, 11, 13, 36, 50] employ a linear run-length encoding that is tightly coupled to the dataflow they use. For example, EIE [13] stores non-zero entries as an interleaved compressed sparse column (CSC) format, which permits a single column of an FC layer weight matrix W to be streamed to the PE array to interact with the same input activations. This layout matches the dataflow during the forward pass, but makes it impossible to calculate addresses within a column of $W^\top$ in the backward pass.

Similarly, the compressed format used for CONV filters in SCNN [36] organizes filter layers so that all sparse filters with the same input channel (and different output channels) are adjacent. In the forward pass, this corresponds to SCNN's input-stationary dataflow where a single input activation is multiplied by all filters from the same input channel and the partial sums are distributed to different output channels; however, in the backward pass the equivalent gradient-stationary dataflow would need to compute addresses for all filters from one *output* channel, which is not possible due to varying filter sparsity.

Procrustes instead uses a variant compressed block format [5] (Section IV-B) to ensure that weights can be compressed but still read efficiently during all relevant training phases.

E. Sparse training algorithms

Training of sparse networks relies on the observation that dense deep neural networks contain small subnetworks (~20% weights) that can be trained to match or exceed the original accuracy *provided* that the initial weight settings for the subnetwork are retained [8, 27]. In effect, most (~80%–90%) of the weights serve as a scaffolding necessary only to identify the weights that should survive in the final pruned subnetwork.

Most of the proposed sparse training algorithms work by gradually increasing sparsity during the training process. The lottery ticket algorithm [8] prunes 20% of the network every 50,000 training iterations by removing the lowest-magnitude weights; the authors report 5–10× model size reduction on CIFAR10 targets. Eager Pruning [49] follows a similar magnitude-based approach, but adds a feedback loop and a checkpoint-based rollback scheme to avoid overpruning; maintaining top-1 accuracy on ImageNet, it can prune ResNet50 2.4× (25.6M→10.8M weights) by removing 0.8% of the weights every 24,000 iterations. Both approaches rely on sorting all weight values to select which weights to keep.

Dynamic sparse reparametrization [33] starts by randomly distributing zero weights at the desired sparsity level, but allows the zeros to redistribute across the weight tensor during training. For ResNet50, for example, ~200,000 additional parameters are set to zero every 1,000–8,000 iterations, but an equal number of weights are allowed to regrow after each pruning step. It avoids the need to sort all weights by using a value threshold adjusted via a set-point feedback loop whenever the network is pruned; however, the initial value of this threshold becomes a hyperparameter. ResNet50 can be pruned 3.5× (25.6M→7.3M) with some top-1 accuracy loss on ImageNet (−1.6%).

In contrast to the gradual pruning approaches [8, 33, 49], the Dropback algorithm [10] prunes the network from the beginning: only a fixed percentage of the parameters (e.g., 10%) are ever allowed to change. In every iteration, only

init: $W^{(0)}$ with $W^{(0)} \sim N(0, \sigma)$
output: $W^{(t)}$
while *not converged* **do**
  $T = \left\{ \left| \sum_{i=0}^{t-1} \frac{\eta \partial f\left(W^{(i-1)}; x^{(i-1)}\right)}{\partial w} \right| \text{ s.t. } w \in W_{trk} \right\}$
  $P = \left\{ \left| \frac{\eta \partial f\left(W^{(i-1)}; x^{(i-1)}\right)}{\partial w} \right| \text{ s.t. } w \in W_{prn} \right\}$
  $S = \text{sort}(T \cup P)$
  $mask = \mathbb{1}(S > S[k])$
  $W^{(t)} =$
   $mask \cdot \left(W^{(t-1)} - \eta \nabla f\left(W^{(t-1)}; x^{(t-1)}\right)\right) + \overline{mask} \cdot W^{(0)}$
  $t = t + 1$

Alg. 2: Dropback algorithm [10]. W_{trk} and W_{prn} = tracked and pruned weights; T and P = tracked and pruned accumulated gradients; S = sorted accumulated gradients; k = number of gradients to keep; η = learning rate. *mask* is a boolean matrix indicating which weights to keep and $\overline{mask}$ is its logical inverse.

the weights with the highest accumulated gradient survive (which again requires sorting), on the theory that this represents learning better than magnitude during early iterations; the pruned weights are reset to their initial values rather than to 0. Dropback prunes ResNet18 11.7× (11.7M→1M) while maintaining top-1 accuracy on ImageNet.

In this paper, we focus on Dropback algorithm (Algorithm 2), which offers by far the highest compression ratios and introduces only one additional parameter (the sparsity factor) during training. Unfortunately, two aspects stand in the way of hardware acceleration: (a) pruned weights are not set to 0, and so MAC energy is not saved; and (b) millions of gradients must be sorted to determine which weights should be pruned. We demonstrate how to overcome these drawbacks and make Dropback algorithm hardware-friendly in Section III.

III. Adapting sparse training algorithms to hardware

To adapt Dropback algorithm to the requirements of an efficient hardware implementation, Procrustes

(i) creates computation sparsity by decaying initial weight values $W^{(0)}$ over the first 1,000 iterations, and
(ii) avoids the need to sort all gradients by using dynamic quantile estimation to continuously determine a threshold value that tracks the target sparsity.

We discuss the details below.

A. Creating computation sparsity

A key challenge in using Dropback algorithm [10] to enable energy-efficient training is the fact that it never entirely removes pruned weights: instead, pruned weights have their values returned to their initialization-time values. These are generally non-zero, so no MAC operations are saved, and, because MAC computation accounts for much of the energy during training (cf. Figures 1 and 17), energy savings are also limited.

To determine how to recover computation sparsity, we first considered the function of the weights during training. We

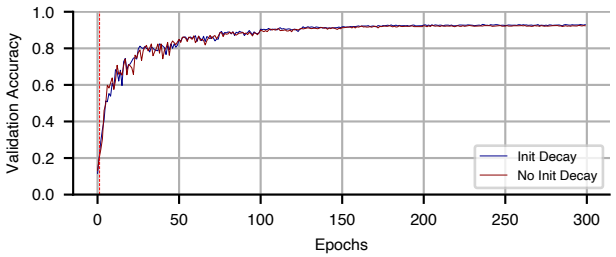


Fig. 6: Validation accuracy over the course of training when initial weights decay 0.9× every iteration, compared to a baseline without decay (VGG-S on CIFAR-10). The dashed vertical line indicates the point at which all initial weights have decayed to zero (1,000 iterations, or early in the second epoch as epochs are 800 iterations each).

hypothesized that the initial weight values are important during the early iterations when weights have not moved far from their initial state and the accumulated gradients are small compared to the initial weights. Later on, we reasoned, the accumulated gradients are much larger than the initial weight values, and the initial scaffolding could safely be removed.

We therefore examined whether the initial weight values could be gradually decayed to zero so that eventually only the accumulated gradients remain and all pruned weights become zero. We decayed the initial weight values 10% every iteration (decay parameter $\lambda = 0.9$), eventually zeroing them; the resulting training scheme is detailed in Algorithm 3. Figure 6 shows how validation accuracy evolves over the course of training compared to a baseline where weights do not decay: neither accuracy nor convergence time are affected.

init: $W^{(0)}$ with $W^{(0)} \sim N(0, \sigma)$
output: $W^{(t)}$
while *not converged* **do**
  $T = \left\{ \left| \sum_{i=0}^{t-1} \frac{\eta \partial f\left(W^{(i-1)}; x^{(i-1)}\right)}{\partial w} \right| \text{ s.t. } w \in W_{trk} \right\}$
  $P = \left\{ \left| \frac{\eta \partial f\left(W^{(i-1)}; x^{(i-1)}\right)}{\partial w} \right| \text{ s.t. } w \in W_{prn} \right\}$
  $S = \text{sort}(T \cup P)$
  $mask = \mathbb{1}(S > S[k])$
  $W^{(t)} =$
   $mask \cdot \left(W^{(t-1)} - \eta \nabla f\left(W^{(t-1)}; x^{(t-1)}\right)\right) + \overline{mask} \cdot \lambda^t W^{(0)}$
  $t = t + 1$

Alg. 3: Dropback algorithm with initial weight decay. W_{trk} and W_{prn} = tracked and pruned weights; T and P = tracked and pruned accumulated gradients; S = sorted accumulated gradients; k = number of gradients to keep; η = learning rate; λ = decay parameter (we used 0.9). *mask* is a boolean matrix indicating which weights to keep and $\overline{mask}$ is its logical inverse.

In this experiment, the initial weight decay scheme results in 80% weights set to zero by iteration 1000 (out of 234,400 total iterations). This means that 60% of computation in 99.5%

of iterations can be entirely skipped, potentially resulting in significant energy savings.

B. Choosing which weights to keep

The second key challenge of the original Dropback algorithm is the need to sort all accumulated gradients to determine which weights should be kept and which should be reset to their initial values. A comparison-based sort requires a minimum of $\log_2(n!)$ comparisons in the worst case — 336M comparisons for the relatively compact VGG-S with 15M weights, compared to the 4.3G MACs required for one training iteration with batch 16. Even if the DNN accelerator were modified to support sorting (i.e., to return both indices and values), sorting would take in excess of 1.3M cycles on a 256-PE device.

To overcome this challenge, we considered replacing the target sparsity factor (such as 10×) with a global value threshold ϑ. In this scheme, every computed gradient is tested whether it should be added to the tracked set T, and added to T only if it exceeds ϑ. This would reduce the number of comparisons to one per produced gradient (15M for VGG-S).

The question is how to determine ϑ for each iteration. Dynamic sparse reparametrization [33] accomplishes this via a set-point feedback scheme that adjusts ϑ every 1,000–8,000 iterations, but this introduces an additional hyperparameter, the initial value of ϑ. Instead, we determine ϑ dynamically via a streaming quantile estimation technique [45], shown in Algorithm 4. To allow for peak update rate (up to 4 per cycle in the last VGG-S CONV layer), we extended the technique to process four updates at once.

The tracking process proceeds as follows:

- If the gradient dimension δ_w is *not* in the tracked set T, $|\delta_w|$ is compared against ϑ. If it is higher, δ_w evicts and replaces the lowest entry in T; otherwise, it is discarded. In either case, $|\delta_w|$ is used to update the quantile estimate (Algorithm 4).
- If δ_w is tracked, it is added to the stored accumulated gradient δ_w^{acc}. The quantile estimate is updated with $|\delta_w^{\text{acc}} + \delta_w|$.

In our experiments, we found that the tracking accuracy sensitivity to the values of $\widehat{Q}_q(0)$ and ϱ is negligible, so we use the same values for all experiments (see Algorithm 4) rather than treating them as hyperparameters.

To determine the accuracy of this estimate, we trained VGG-S using a sparsity target of 7.5× and streamed the computed accumulated gradients to the estimator. Figure 7 shows that while the quantile estimation exhibits minor deviations from ground truth (because different layers have different amounts of sparsity), these estimation errors have no detrimental effect on the validation accuracy of the trained network. Overall, the quantile estimation error results in extra weights being tracked, and reduces the sparsity factor slightly from 7.5× to 5.2×; however, this overhead is much lower than that required to sort all weights or to train a dense network.

Note that selecting weights through quantile estimation is not specific to the Dropback algorithm: separating some fraction of the highest-value or highest-gradient weights is needed by all sparse training algorithms [8, 10, 32, 33].

init: $\widehat{Q}_q(0) = 10^{-6}$; $\varrho = 10^{-3}$
input: $\delta(n), \widehat{Q}_q(n)$
output: $\widehat{Q}_q(n+1)$
if $\widehat{Q}_q(n) < \delta(n)$ **then**
  $\widehat{Q}_q(n+1) = (1 + \varrho q)\,\widehat{Q}_q(n+1)$
else
  $\widehat{Q}_q(n+1) = (1 - \varrho\,(1-q))\,\widehat{Q}_q(n+1)$

Alg. 4: The quantile estimation algorithm DUMIQUE [45]. $\delta(n)$ = the n^{th} accumulated gradient value computed; $\widehat{Q}_q(n)$ = the q^{th} quantile estimate at step n; ϱ = adjustment rate hyperparameter. Procrustes uses a modified, parallelized variant which treats the average of four incoming accumulated gradients as a single $\delta(n)$.

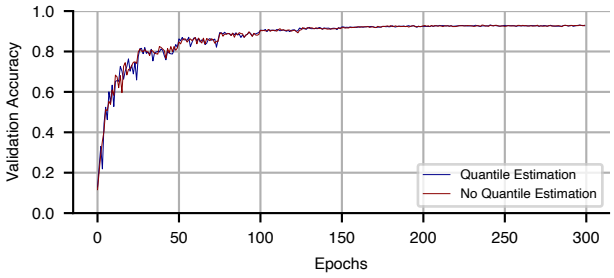


Fig. 7: Validation accuracy over training epochs when sparse training is used and quantile estimation (Algorithm 4) is used to determine the value threshold ϑ under which accumulated gradients are discarded, compared to a baseline with initial weight decay and exact sorting (VGG-S on CIFAR-10).

IV. Dataflow & sparse data format

A. Storage and sparsity during training

Weights (or, more precisely, accumulated gradients) are always stored compressed using the format described in Section IV-B. Typically, all weight gradients are produced, but most gradients that are not already tracked will not survive the comparison with existing accumulated gradients.

Activations are stored uncompressed for immediate reuse and in a compressed format for long-term reuse. The forward pass reads sparse weight tensor, and produces a dense output activation tensor, which is then immediately reused as inputs to the next layer; the activations are then compressed using a sparse, zero-free format, and reused in the weight update stage. This technique is similar in spirit to Gist [21].

B. Compressed sparse weight representation

To avoid the challenges discussed in Section II-D, a sparse weight storage format designed for training must support:

(i) iterating through 2D convolution filters across different dimensions in different stages (for CONV layers), and across both rows and columns of weight matrices (for FC layers),

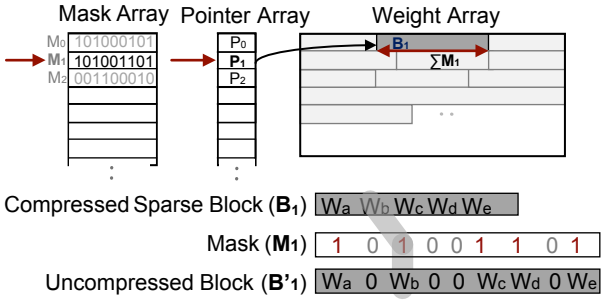


Fig. 8: The compressed sparse block (CSB) weight representation in Procrustes.

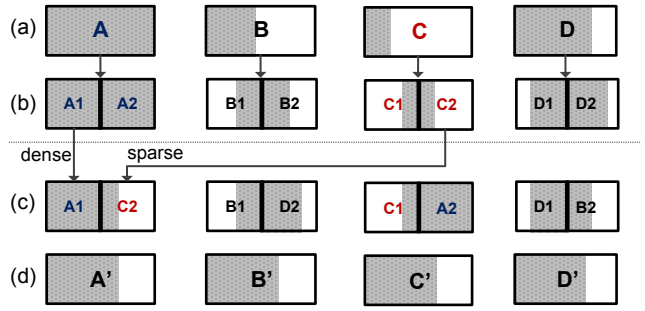


Fig. 9: Load balancing: work tiles are cut in half (b) and the halves rearranged in dense-sparse pairs (c).

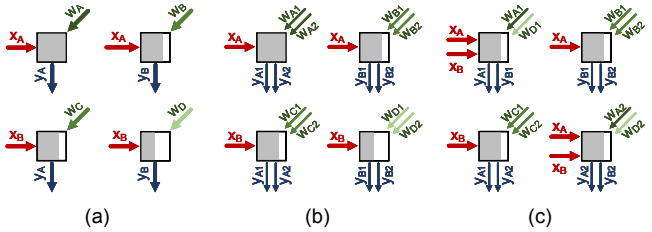


Fig. 10: Load balancing in the weight-stationary C, K dataflow in a four-PE array: (a) PE workload imbalance (shaded PEs) due to different weight sparsities (shaded arrows); (b) PE workloads and the corresponding weight (w) and partial sum (y) tiles split in half across the K dimension (note the thinner arrows); (c) half-tiles exchanged between the top-left and bottom-right PEs for load balancing. Activations must be sent on both rows and columns, and require twice the buffer space in the PEs.

(ii) rotating kernels (for CONV layers) or transposing weight matrices (for FC layers), and
(iii) different kernel sizes in CONV layers.

Procrustes uses a modified compressed sparse block (CSB) format [5] shown in Figure 8 to store weights in the on-chip global buffer and external DRAM. Blocks store non-zero values and are variable in size because of sparsity, but correspond to fixed-size regions in the corresponding dense weight space — kernels for CONV layers, square fragments of the weight matrix in FC layers, etc. The region size can vary on layer granularity to support different kernel sizes.

The Procrustes CSB format comprises three components, illustrated in Figure 8:

(a) the weight array, which stores variable-size packed weight blocks corresponding to kernels, etc.;
(b) the pointer array, indexed by tensor coordinates, which identifies the weight array location that stores the relevant weight values; and
(c) the mask array, also indexed by tensor coordinates, which stores a mask identifying non-zero value locations in the unpacked block (and therefore also the packed size).

The pointer and mask arrays are decoupled to support different mask lengths for each layer (e.g., different kernel sizes in CONV layers, flexible block sizes in FC layers and during weight update, and so on); in all of our simulations, mask arrays fit in the on-chip GLB.

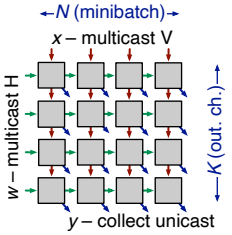
Because the pointer array is indexed by coordinates in the original (dense) operation space and is decoupled from the compressed contents, the format makes computing kernel addresses straightforward while adapting cleanly to different kernel dimensions. The indirection also makes it easy to determine the density of working sets assigned to each PE: it suffices to subtracting pointers of adjacent work tiles. In addition, because blocks are sized to and retrieved on filter granularity, they can be rotated (to be used in the backprop pass) while being fetched from the global buffer to the per-PE register files; similarly, transposition of the weight matrix for the FC layers can be done by transposing subtensors piecewise.

Activations are stored uncompressed for short term reuse (as activations in the next layer) and compressed in CSB format for long-term reuse (forward pass to weight update).

C. Load balancing and dataflow

Figure 9 illustrates the load balancing process used in Procrustes. First, every work tile (a) is cut into two halves along one of the tile dimensions (b); because sparsity is almost certainly uneven within the tile, the two halves will likely have different densities. Next, the halves are sorted according to density, and half-tiles are matched starting from opposite ends (c): the sparsest half-tile is matched with the densest half-tile, and so on. This ensures that each newly formed tile is as close as possible to the average density across all PE work tiles (d).

However, naively applying this rebalancing scheme to the entire PE array without changing the dataflow would impact on-chip communications patterns and require a complex interconnect. Figure 10 demonstrates this on the weight-stationary C, K dataflow on a 4-PE array. In pane (a), input activations are broadcast horizontally (x_A in the top row and x_B in the bottom row), partial sums are accumulated vertically (y_A in the left column and y_B in the right column), while the weights are unicast (as in Figure 3); however, because the weights have different levels of sparsity (shaded arrows), the PEs have different amount of computation (shaded PEs). In pane (b), each PE's workload is cut in half as discussed above; each weights tile (w_A and w_B) is also split in half (e.g., into $w_{A1} + w_{A2}$ and $w_{B1} + w_{B2}$, note the thinner arrows), as are the corresponding partial sums (y_A and y_B). Finally, in pane (c), the workload



	fw	bw	wu
(H)orizontal	w	w	$\partial\mathcal{L}/\partial w$
(V)ertical	x	$\partial\mathcal{L}/\partial x$	x
(U)nicast	y	$\partial\mathcal{L}/\partial y$	$\partial\mathcal{L}/\partial y$

w weights x input activations
y output activation partial sums

Fig. 11: Mappings and dataflows that spatially distribute the minibatch across one dimension of the PE array.

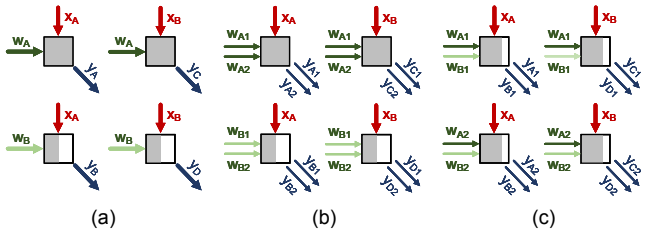


Fig. 12: Load balancing in the proposed K, N dataflow in a four-PE array: (a) PE workload imbalance (shaded PEs) due to different weight sparsities (shaded arrows); (b) PE workloads and the corresponding weight (w) and partial sum (y) tiles split in half across the K dimension (note the thinner arrows); (c) half-tiles exchanged between the top-left and bottom-right PEs to load-balance across K. Each input activation tile is still sent to only one column.

halves are balanced across the PE array, so that the top-left and bottom-right PEs swap half their workloads; this, however, means that all input activations (x_A and x_B) must now be sent to both columns and rows, requiring more bandwidth and a more complex interconnect, and double the activations must be buffered at the target PEs. The P, Q input-stationary dataflow faces similar challenges in the weight update pass (cf. Figure 2) and requires unicasting two of the three datatypes.

Procrustes addresses both of these problems by leveraging a simple observation: training is typically done across a minibatch of 32–64 samples rather than on single items [4, 31].†

Because a training accelerator does not need to support single-sample inference, the minibatch dimension (N in Algorithm 1) can be used to distribute work tiles across one dimension of the PE array. The other dimension can then be safely chosen to be a dimension where sparsity exists — e.g., the input or output channel dimensions (C or K) with weight sparsity. Because only one dimension is sparse, and that dimension corresponds to spatial reuse, the load balancing process needs to be applied only to one dimension of the PE array (i.e., the dimension opposite to the spatial reuse pattern, here N).

Figure 11 illustrates how a K, N mapping (output channel, minibatch) with load balancing across the output channel (K) dimension preserves the single-dimension dataflow properties

†Minibatches in the 1,000s allow faster training on large multi-GPU clusters but can incur some accuracy cost [1, 12].

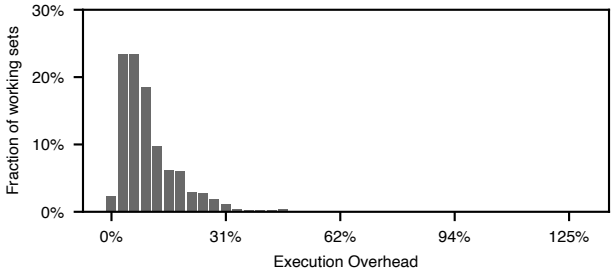


Fig. 13: Load imbalance histogram of full-chip working sets (columns in Figure 4b) after load-balancing half-tiles (VGG-S/CIFAR-10). Compare with Figure 5. A perfectly load-balanced workload would have 100% of the sets at 0% overhead.

(cf. Figure 3) during the forward pass. Weights are now the same across the minibatch and are multicast across the horizontal dimension of the PE array, partial sums are collected across the vertical dimension, and input activations vary across both dimensions and so are unicast.

A detailed example is shown in Figure 12. As in Figure 10, pane (a) shows the unbalanced workload, pane (b) shows each PE's workload (and consequently the weight and partial sum tiles) cut into half, and pane (c) shows the PE array after load-balancing along the K dimension. Observe that, in contrast to Figure 10, the load-balanced dataflow in pane (c) has the same on-chip interconnect communication patterns and requires the same interconnect bandwidth as the unbalanced dataflow in pane (c).

Finally, Figure 13 demonstrates that this technique results in effective load balancing. While the balance is not 100% perfect, the execution time overheads for most full-PE-array working sets are small at <10%, with the worst imbalance at 30% — a vast improvement to the common 40%–50% overheads and up to 2× slowdown without load balancing (see Figure 5).

V. Hardware Architecture

The overall hardware architecture of Procrustes is based on 2D PE array where each PE has a local register file (RF) and all PEs share an on-chip global buffer (GLB); an off-chip DRAM completes the memory hierarchy. PEs are interconnected via three simple interconnects: two support one-dimensional traffic flows in the horizontal and vertical directions, and one supports unicast traffic to any PE in the array. Because Procrustes focuses on training, we use 32-bit floating point MAC units in the PE datapath, but the design can be used with any datatype.

The design is illustrated in Figure 14, with differences from the baseline accelerator dashed. Procrustes places one global quantile estimation unit (QE) between global buffer and the external DRAM; the QE unit monitors accumulated gradients flowing from the GLB to DRAM and discards all except those above the target sparsity quantile.

In addition, each PE contains a weight recomputation unit (WR) responsible for generating the initial weight values. The WR accepts a weight index and generates a 32-bit integer initial

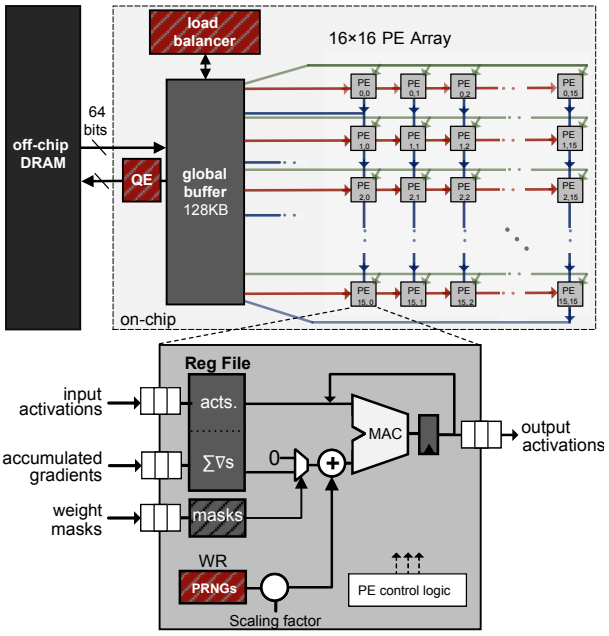


Fig. 14: Procrustes system architecture. The WR module is added to recreate initial weights for Dropback-style training, the QE unit is added to support quantile estimation, and the load balancer is added to support work tile re-balancing.

Baseline dense accelerator	
PEs	256 (16×16)
datatype	32-bit floating-point
pruning type	none
interconnect	3× 1D-flow interconnect
global buffer	128 KB
local buffer (RF)	1 KB per PE
dataflow	optimal (via Timeloop+Accelergy)
Procrustes modifications	
pruning type	lowest accumulated gradients
pseudo-RNG	xorshift [30], one per PE
quantile estimator	DUMIQUE [45], max 4 requests / cycle
dataflow	optimal spatial-minibatch dimension

TABLE I: Hardware configurations for the baseline dense training accelerator and Procrustes sparse training accelerator.

value for the relevant weight. It consists of 3 xorshift [30] pseudo-random generators (RNGs) whose outputs are added to produce an approximately Gaussian output. Note that, unlike conventional RNG, the WR unit does not contain hidden state, and is purely a function of its seed and the weight index. The "RNG" output is then scaled using an integer multiplier; this this enables popular initialization formulæ like Xavier [9] or Kaiming [17], and allows the initial weights to be decayed as per Algorithm 3. Finally, the scaled value is converted to FP32 and added to the accumulated gradient retrieved from weight storage if the weight is tracked, or to zero if the weight has been pruned.

VI. Evaluation

A. Methods

We re-implemented the baseline Dropback training algorithm [10] using PyTorch [37] and verified the reported training sparsity levels and accuracy results; we then implemented the initial weight decay (IC) and quantile-estimation extensions needed for Procrustes.

We evaluated Procrustes on five CNNs: ResNet18 [18] (11.7M weights) and MobileNet v2 (3.5M weights) applied to the ImageNet image classification task [38], as well VGG-S [46] (a 9.2× reduced version of VGG-16 with 15M weights), WRN-28-10 [47] (36.5M weights), and a small Densenet [19] (growth rate 24, 3 blocks × 10 layers, for a total of 2.7M weights), all CIFAR-10 [23].

To determine optimal mappings and dataflows, we extended Timeloop [35] to support sparse weight masks (retrieved from our PyTorch model), model sparse computation, account for sparse encoding overheads, and accurately reflect latency due to load imbalances. We also used Timeloop to determine cycle-level latency; to determine energy costs, we use energy access cost provided in Accelergy [42] with its default 40nm library. We modelled all layers of all networks and all stages of training (forward, backward, and weight update).

As a dense baseline, we used a 2D PE array architecture with 16×16 PEs, adapted to the 32-bit floating-point precision commonly used in training; we used Timeloop to determine the optimal tiling and dataflow. Hardware modules not present in the baseline were implemented in Verilog RTL and synthesized using Synopsys DC in the 45nm FreePDK process. Accelerator configuration details are shown in Table I.

B. Pruning ratios and accuracy

Table II shows the sparsity factors achieved while maintaining the same accuracy as the corresponding dense (unpruned) network using the Procrustes sparse training algorithm. Depending on the network, our training scheme achieves 3.9×–11.7× weight sparsity without compromising accuracy.

Importantly, achieving unpruned-level accuracy does not require additional convergence time. Figure 15 demonstrates this on the VGG-S, DenseNet, and WRN, all on CIFAR-10. Figure 16 demonstrates the same effect on ResNet18 trained on ImageNet at various weight pruning ratios. Overall, Procrustes converges reaches state-of-the-art accuracy as quickly (or faster) than the baseline unpruned network.

C. Energy savings and speedup

Figure 17 shows the energy savings obtained by training with Procrustes across several CNNs. Most of the energy is saved by performing fewer MAC operations; because training is most often done on FP32 values, MACs dominate the energy usage. Intra-PE register file (RF), global buffer (GLB), and DRAM access energies are also substantially reduced, but account for less of the baseline energy expenditure, and therefore contribute less to the overall savings.

The figure also illustrates that Procrustes can transform higher sparsity ratios into bigger energy savings: ResNet18,

model	dataset	dense size	dense MACs	sparse size	sparse MACs	sparsity	# epochs	dense accuracy	pruned accuracy
Densenet	CIFAR-10	2.7M	528M	692k	157M	3.9×	340	94.2%	93.7%
WRN-28-10	CIFAR-10	36M	4G	8.3M	863M	4.3×	462	96.0%	96.1%
VGG-S	CIFAR-10	15M	269M	2.9M	113M	5.2×	236	93.0%	93.1%
MobileNet v2	ImageNet	3.5M	301M	0.35M	75M	10×	131	70.98%	71.13%
ResNet18	ImageNet	11.7M	1.8G	1M	359M	11.7×	81	69.17%	69.31%

TABLE II: Sparsity achieved using the Procrustes training scheme for the CNNs tested, together with weight footprint and MAC reduction and the final accuracy compared to the dense baseline.

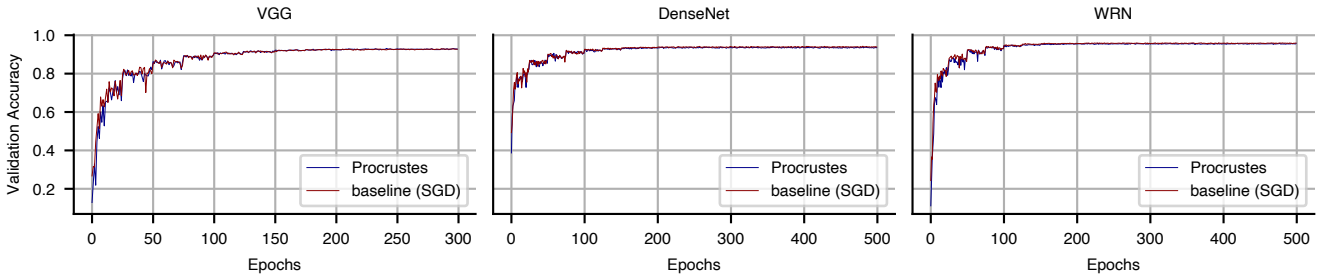


Fig. 15: Validation accuracy over training time for Procrustes and the unpruned baseline (SGD) on CIFAR-10: (left) VGG-S, (centre) DenseNet, and (right) WRN-10-28.

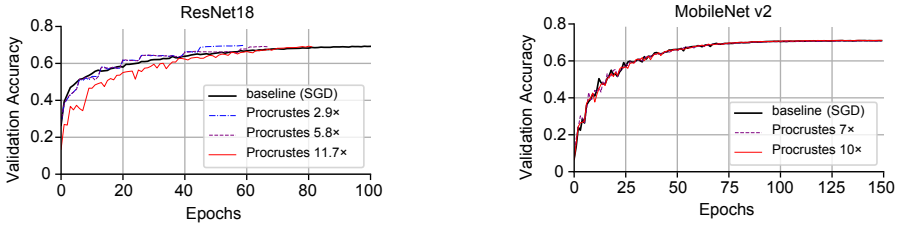


Fig. 16: Validation accuracy over training time for Procrustes and the unpruned baseline for ResNet18 (left) and MobileNet v2 (right) on ImageNet.

which has the highest pruning factor (11×), saves the most energy compared to the dense baseline (3.26×), while WRN has the best speedup (4×). MobileNet v2 benefits less in energy because its depth-separable convolutions limit reuse and so comparatively more energy is spent on DRAM accesses; however, Procrustes still trains it with 2.39× less energy than the dense baseline, and almost as much speedup as WRN (3.88× faster than dense).

For most networks, the forward and back-propagation passes offer more energy savings; this is because those passes can take advantage of weight sparsity, which is generally higher than activation sparsity. VGG-S demonstrates a less common case where the weight sparsity is concentrated in the layers that perform relatively few MACs, so the activation sparsity leveraged by the weight-update phase actually saves more operations.

Overall, Procrustes is effective in converting training-time sparsity to energy savings.

D. Mapping and dataflow choice

Figure 18 shows how energy expenditure varies with different spatial partitioning schemes. Sparsity enables energy improvement across all phases and all mappings. Because the number of MAC operations and the memory hierarchy are the same across the different mappings, the lion's share of the energy use is the same across the different dataflows, and variations are negligible. This is in agreement with prior work that also reported negligible impact of the chosen dataflow on the energy during inference [44].

This finding enables us to select spatial partitioning that results in the best performance (i.e., shortest execution time).

Figure 19 shows how execution times vary when the working set is mapped to the PE array using different spatial partitioning schemes; all schemes can be implemented using the simple network topology shown in Figure 14 except the weight-stationary C, K scheme, which requires a complex network to load-balance PE working sets across the entire chip. The partitioning schemes that distribute the minibatch dimension along one of the PE array dimensions (C, N and K, N) are the fastest mappings because they are able to achieve effective load balancing and good utilization across all layers of the CNNs; K, N performs slightly better because it offers slightly higher utilization in the first network layer. The C, K scheme performs less well even though it requires a more complex

Component	Power (mW)	Area (μm^2)
Per-PE area: Procrustes overheads *italicized*		
FP32 MAC	7.29	18,875.72
Register File	15.61	198,004.71
PRNG	*0.35*	*1,920.84*
Mask Memory	*2.65*	*44,932.66*
System area: Procrustes overheads *italicized*		
Global Buffer	73.74	17,109,596.5
Quantile Engine	*1.38*	*9,861.4*
Load Balancer	*2.05*	*8,725.23*

TABLE III: Silicon area costs and overheads (synthesis using Synopsys DC with the FreePDK 45nm library). For fairness, the power estimates assume the same dense computation (i.e., no sparsity).

interconnect, largely because it is inefficient on layers that have few channels. The activation-stationary P, Q scheme does not require load-balancing in the forward and back-propagation phases, is hard to load-balance during the weight update phase, and has low utilization when activation tensors are relatively small; it is overall the slowest mapping.

Procrustes uses the overall fastest K, N scheme for all phases of training.

E. Scalability

Figure 20 shows how Procrustes scales when the PE array size is quadrupled from 256 cores (16×16) to 1024 cores (32×32); the global buffer size is doubled over the 256-core size (a factor of $\sqrt{2}$). Overall, energy is very similar same for all dataflows / passes because the number of MAC operations is the same. Latency scales near ideally (3.9× on 4× the cores) in the K, N mapping used by Procrustes. Other mappings (especially activation-stationary P, Q) do not scale as well since they trade off PE array utilization to retain spatial reuse.

F. Silicon area overheads

The silicon area and power overheads of Procrustes are detailed in Table III. Despite the RNG initial weight recomputation module being included in every PE, its area and power pale in comparison to the FP32 MAC unit which all PEs include.

Overall, the Procrustes accelerator has an area overhead of 14% over an equivalent dense accelerator, and consumes 11% more power when executing the same *dense* workloads. Both are a small price to pay for the 2.27×–3.26× energy savings offered by sparse training.

G. Generality

Procrustes is the first sparse training accelerator to combine substantial sparsity ratios, 2.27×–3.26× energy savings, and up to 4× speedups while maintaining state-of-the-art accuracy of the trained networks. While in this paper we use Procrustes to extend the Dropback training algorithm, the quantile estimation and spatial-minibatch dataflow insights apply to all existing — and likely many future — sparse training algorithms.

VII. Related work

A. Sparse accelerators

Eager Pruning [49] is the only extant proposal for a sparse training accelerator. It works by starting with a dense network and very gradually pruning the lowest-magnitude weights, with fewer than 1% of weights removed every tens of thousands of training iterations; maintaining accuracy limits pruning to comparatively low factors of 1.5–3.5×. The accelerator uses on a weight-stationary dataflow where denser filters are distributed over more PEs than sparser filters; to manage the resulting irregularity in collecting partial sums, the authors propose a module that connects the PEs and can either accumulate or route partial sums. Although the Eager Pruning algorithm relies on sorting weights, this does not appear to be considered in the hardware or the latency and energy measurements. In contrast, Procrustes achieves higher pruning factors, does not rely on sorting weights, and avoids the need for a complex interconnect via a novel load balanced dataflow.

All other sparse accelerators only support inference. EIE [13] and CambriconX [50] use a variants of the compressed sparse column format, which prevents them from efficiently accessing weights during the backward pass. SCNN [36] and SparTen [11] use an input-stationary dataflow to enable both weight and activation sparsity; however, both use a CSC-like format to encode sparse weights, and neither can be used to accelerate training.

B. Sparse training algorithms

Most proposed sparse training algorithms very slowly increase sparsity during the training process. The lottery ticket algorithm [8] prunes 20% of the network every 50,000 training iterations by removing the lowest-magnitude weights; the authors report 5–10× model size reduction on CIFAR10 targets. Eager Pruning [49] follows a similar magnitude-based approach, but adds a feedback loop and a checkpoint-based rollback scheme to avoid overpruning; maintaining top-1 accuracy on ImageNet, it can prune ResNet50 2.4× (25.6M→10.8M weights) by removing 0.8% of the weights every 24,000 iterations. Unlike Procrustes, both approaches rely on sorting all weight values to determine which weights to keep.

Dynamic sparse reparametrization [33] starts by randomly distributing zero weights at the desired sparsity level, but allows the zeros to redistribute across the weight tensor during training. For ResNet50, for example, ~200,000 additional parameters are set to zero every 1,000–8,000 iterations, but an equal number of weights are allowed to regrow after each pruning step. It avoids the need to sort all weights by using a value threshold adjusted via a set-point feedback loop whenever the network is pruned; however, the initial value of this threshold becomes a hyperparameter. This method prunes ResNet50 3.5× (25.6M→7.3M) with some top-1 accuracy loss on ImageNet (−1.6%). Procrustes offers higher sparsity factors and prunes the network much more quickly, which translates to substantial energy savings.

Dropback [10] prunes the network from the beginning: only a fixed percentage of the parameters (e.g., 10%) are ever

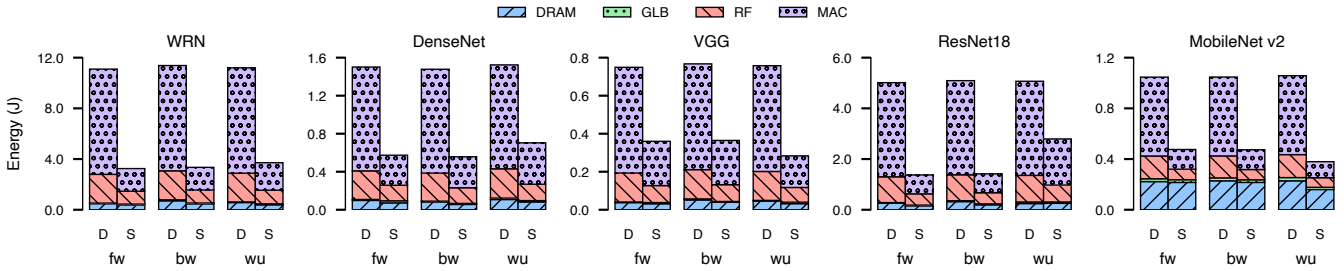


Fig. 17: Energy breakdown of using KN dataflow for (left) WRN-10-28, (middle left) DenseNet, (middle) VGG-S, (middle right) ResNet18, and (right) MobileNet v2. Lower is better. K = output channel dimension; N = minibatch dimension. S = sparse; D = dense. fw = forward pass; bw = backward pass; wu = weight update phase.

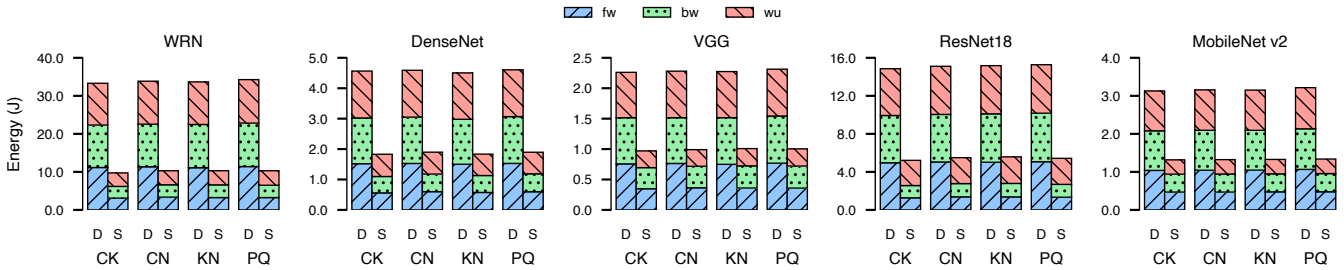


Fig. 18: Energy Comparison across different dataflows for (left) WRN-10-28, (middle left) DenseNet, (middle) VGG-S, (middle right) ResNet18, and (right) MobileNet v2. Lower is better. C = input channel dimension; K = output channel dimension; P and Q = output activation dimensions; N = minibatch dimension. S = sparse; D = dense. fw = forward pass; bw = backward pass; wu = weight update phase.

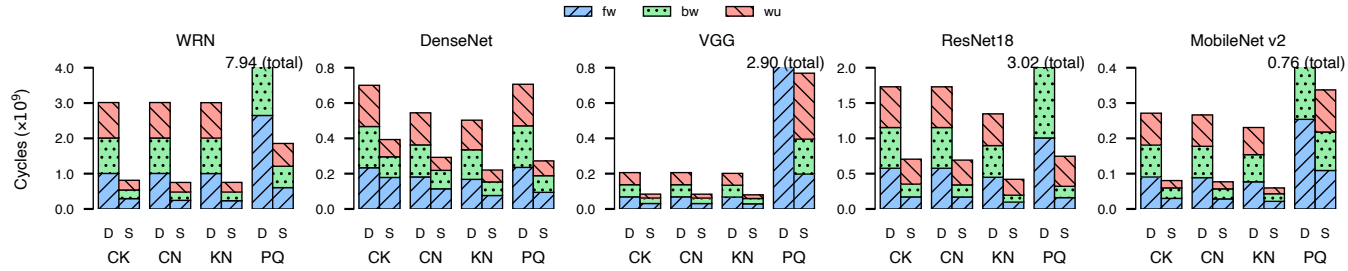


Fig. 19: Training latency across different dataflows for (left) WRN-10-28, (middle left) DenseNet, (middle) VGG-S, (middle right) ResNet18, and (right) MobileNet v2. Lower is better. C = input channel dimension; K = output channel dimension; P and Q = output activation dimensions; N = minibatch dimension. S = sparse; D = dense. fw = forward pass; bw = backward pass; wu = weight update phase.

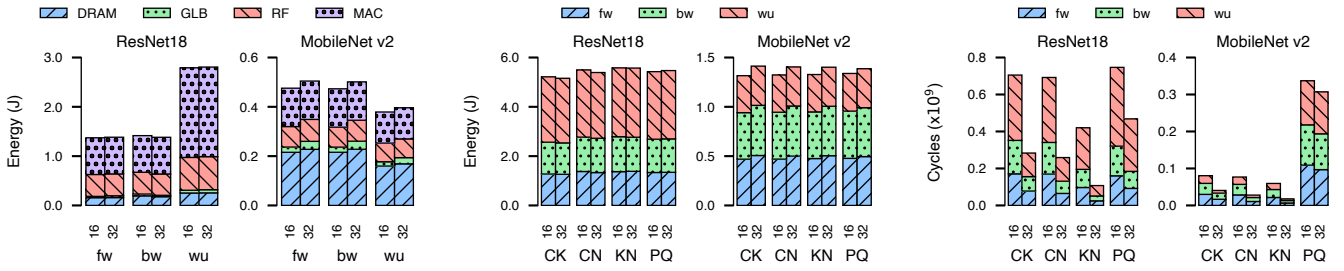


Fig. 20: Scalability of Procrustes on 16×16 (256) to 32×32 (1024) cores on ResNet-18 and MobileNet v2 classifying ImageNet configured as in Figs. 17–19. Energy differences are is negligible as the workload is the same. Speedup scales best for the Procrustes mappings (CN and KN) because other mappings trade off utilization for reuse.

allowed to change. In every iteration, only the weights with the highest accumulated gradient survive (which again requires sorting), on the theory that this represents learning better than magnitude during early iterations; the pruned weights are reset to their initial values rather than to 0. With Dropback, ResNet18 can be pruned 11.7× (11.7M→1M) while maintaining top-1 accuracy on ImageNet. Procrustes adapts Dropback to the needs of an efficient hardware implementation, removing the requirement for sorting and decaying initial weights to 0 to create computation sparsity.

VIII. Summary

This paper introduces Procrustes, a sparse DNN training accelerator that produces pruned models with the same accuracy as dense models without first training, then pruning, and finally retraining, a dense model.

Procrustes relies on three key techniques. First, it adapts an existing training algorithm to create computation sparsity that can be converted into energy savings. Next, it replaces the sorting step present in nearly all sparse training algorithms with hardware-friendly, computationally simple quantile estimation. Finally, it leverages a novel load-balancing scheme that converts sparsity into speedup, and proposes a novel dataflow that enables load balancing without significant changes to the on-chip interconnect.

IX. Acknowledgements

The authors are grateful to the anonymous reviewers for insightful feedback and helpful suggestions.

This material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Project Agency (DARPA) under agreement number FA8650-20-2-7007, and by the Natural Sciences and Engineering Research Council of Canada (NSERC) under award number NETGP 485577-15. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL), Defense Advanced Research Project Agency (DARPA), the U.S. Government, the Natural Sciences and Engineering Research Council of Canada (NSERC), or the Government of Canada.

References

[1] T. Akiba, S. Suzuki, and K. Fukuda, "Extremely large minibatch SGD: training ResNet-50 on ImageNet in 15 minutes," *arXiv preprint arXiv:1711.04325*, 2017.

[2] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: ineffectual-neuron-free deep neural network computing," in *ISCA*, 2016.

[3] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *MICRO*, 2016.

[4] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," in *Neural networks: Tricks of the trade*. Springer, 2012, pp. 437–478.

[5] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *SPAA*, 2009.

[6] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ASPLOS*, 2014.

[7] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2019.

[8] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," in *ICLR*, 2019.

[9] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *AISTATS*, 2010.

[10] M. Golub, G. Lemieux, and M. Lis, "Full Deep Neural Network Training on a Pruned Weight Budget," in *SysML*, 2019.

[11] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. Vijaykumar, "SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks," in *MICRO*, 2019.

[12] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: Training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.

[13] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: efficient inference engine on compressed deep neural network," in *ISCA*, 2016.

[14] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," in *ICLR*, 2016.

[15] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *NIPS*, 2015.

[16] B. Hassibi and D. G. Stork, "Second order derivatives for network pruning: Optimal brain surgeon," in *NIPS*, 1993.

[17] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification," in *ICCV*, 2015.

[18] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *CVPR*, Jun. 2016.

[19] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *CVPR*, 2017.

[20] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," in *ICML*, 2015.

[21] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, "Gist: Efficient data encoding for deep neural network training," in *ISCA*, 2018.

[22] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *ISCA*, 2017.

[23] A. Krizhevsky, "Learning multiple layers of features from tiny images," Master's thesis, University of Toronto, 2009.

[24] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, "Understanding Reuse, Performance, and Hardware Cost of DNN Dataflow: A Data-Centric Approach," in *MICRO*, 2019.

[25] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *NIPS*, 1990.

[26] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient backprop," in *Neural networks: Tricks of the trade*. Springer, 2012, pp. 9–48.

[27] C. Li, H. Farkhoor, R. Liu, and J. Yosinski, "Measuring the intrinsic dimension of objective landscapes," in *ICLR*, 2018.

[28] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," in *ICLR*, 2017.

[29] J.-H. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," in *ICCV*, 2017.

[30] G. Marsaglia *et al.*, "Xorshift RNGs," *Journal of Statistical Software*, vol. 8, no. 14, pp. 1–6, 2003.

[31] D. Masters and C. Luschi, "Revisiting small batch training for deep neural networks," *arXiv preprint arXiv:1804.07612*, 2018.

[32] D. C. Mocanu, E. Mocanu, P. Stone, P. H. Nguyen, M. Gibescu, and A. Liotta, "Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science," *Nature Communications*, vol. 9, p. 2383, 2018.

[33] H. Mostafa and X. Wang, "Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization," in *ICML*, 2019.

[34] NVIDIA, "NVIDIA Deep Learning Accelerator (NVDLA)," 2017. [Online]. Available: http://nvdla.org/

[35] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A Systematic Approach to DNN Accelerator Evaluation," in *ISPASS*, 2019.

[36] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *ISCA*, 2017.

[37] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *NIPS*, 2019.

[38] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[39] Y. Shen, M. Ferdman, and P. Milder, "Overcoming resource underutilization in spatial CNN accelerators," in *FPL*, 2016.

[40] Y. Shen, M. Ferdman, and P. Milder, "Maximizing CNN accelerator efficiency through resource partitioning," in *ISCA*, 2017.

[41] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks," in *FPGA*, 2016.

[42] Y. N. Wu and V. Sze, "Accelergy: An architecture-level energy estimation methodology for accelerator designs," 2019.

[43] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing energy-efficient convolutional neural networks using energy-aware pruning," in *CVPR*, 2017.

[44] X. Yang, M. Gao, J. Pu, A. Nayak, Q. Liu, S. E. Bell, J. O. Setter, K. Cao, H. Ha, C. Kozyrakis *et al.*, "Dnn dataflow choice is overrated," *arXiv preprint arXiv:1809.04070*, 2018.

[45] A. Yazidi and H. Hammer, "Multiplicative update methods for incremental quantile estimation," *IEEE Transactions on Cybernetics*, vol. 49, pp. 746–756, 2017.

[46] S. Zagoruyko. (2015) Torch | 92.45% on CIFAR-10 in Torch. [Online]. Available: http://torch.ch/blog/2015/07/30/cifar.html

[47] S. Zagoruyko and N. Komodakis, "Wide residual networks," *arXiv preprint arXiv:1605.07146*, 2016.

[48] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *FPGA*, 2015.

[49] J. Zhang, X. Chen, M. Song, and T. Li, "Eager pruning: algorithm and architecture support for fast training of deep neural networks," in *ISCA*, 2019.

[50] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-X: An accelerator for sparse neural networks," in *MICRO*, 2016.