InterGrad: Energy-Efficient Training of Convolutional Neural Networks via Interleaved Gradient Scheduling

Nanda K. Unnikrishnan, *Graduate Student Member, IEEE*; and Keshab K. Parhi, *Fellow, IEEE*

***Abstract*—This paper addresses the design of accelerators using systolic architectures to train convolutional neural networks using a novel *gradient interleaving* approach. Training the neural network involves computation and backpropagation of gradients of error with respect to the activation functions and weights. It is shown that the gradient with respect to the activation function can be computed using a *weight-stationary* systolic array, while the gradient with respect to the weights can be computed using an *output-stationary* systolic array. The novelty of the proposed approach lies in *interleaving* the computations of these two gradients on the same *configurable* systolic array. This results in the reuse of the variables from one computation to the other and eliminates unnecessary memory accesses and energy consumption associated with these memory accesses. The proposed approach leads to $1.4-2.2\times$ savings in terms of the number of cycles and $1.9\times$ savings in terms of memory accesses in the fully-connected layer. Furthermore, the proposed method uses up to 25% fewer cycles and memory accesses, and 16% less energy than baseline implementations for state-of-the-art CNNs. Under iso-area comparisons, for Inception-v4, compared to weight-stationary (WS), Intergrad achieves 12% savings in energy, 17% savings in memory, and 4% savings in cycles. Savings for Densenet-264 are 18%, 26%, and 27% with respect to energy, memory, and cycles, respectively. Thus, the proposed novel accelerator architecture reduces the latency and energy consumption for training deep neural networks.**

***Index Terms*—Neural Network Training, Backpropagation, Convolutional Neural Networks, Accelerator Architectures, Gradient interleaving, Interleaved Scheduling, Systolic Array.**

I. Introduction

DEEP neural networks (DNNs) are brain-inspired models which have progressed significantly from their early days to make inroads into our daily lives [1]. DNNs [2]–[5] are used in applications such as recommender systems, automated photo recognition, and automatic text generation and have led to a massive surge in data center workloads. These networks operate in two distinct phases: the *training* phase, where the network *learns* underlying statistical relationships between inputs and outputs, and the *inference* phase, where the network predicts an output on a previously unseen input.

Thus, acceleration for on-device inference has been extensively investigated with numerous architectures and variants [6]–[8]. However, as DNN sizes have been continuously increasing [9], the one-time training cost is no longer insignificant. For example, even smaller networks such as Alexnet [2] can consume up to $54kJ$ for training a single epoch [10]. Moreover, the enormous energy consumption is exacerbated with larger state-of-the-art DNNs and has significant environmental impacts [11]. Although few accelerators support training [12], [13], most are neither dedicated nor optimized for training. This paper attempts to bridge this gap in the literature by exploring design and optimization techniques that can address some of the unique challenges of designing a *training* accelerator.

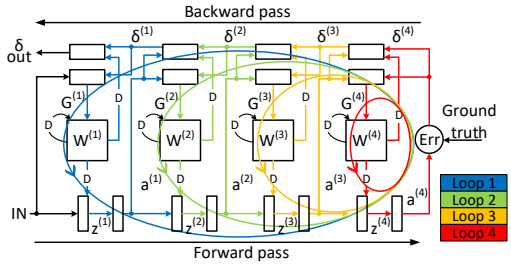

Fig. 1. Training loops for a 4-layer neural network [10].

The bottleneck in terms of energy consumption during training is the backpropagation algorithm. The algorithm computes the gradients of the loss function at each layer with respect to the current layer's weights and the previous layer's activation outputs. A key aspect of optimizing the backpropagation algorithm is understanding the dataflow between these operations. Relative tradeoffs between different dataflow have been studied in detail [14], with a general trend towards flexible architectures. Flexible architectures have shown promise at taking advantage of the relative strengths of the different flows at different stages or layers of the network [7], [15]. However, the granularity at which they can operate limits their flexibility. As the backpropagation step in a layer consists of multiple operations, changing the dataflow within the same operation can open new avenues for variable reuse.

In this paper, we propose a *configurable systolic array* that uses *interleaving* [16], [17] to combine gradient computations that share common variables in both the fully-connected (FC) and convolutional layers of the neural network. The proposed approach, which exploits reuse of common variables, has four distinct advantages: (i) it eliminates the need for multiple memory reads, which optimizes both throughput and energy, (ii) it allows fine-grained dataflow control in the array on a per-cycle and per-processing element (PE) basis, (iii) it avoids external image to column (im2col) operations in the

N. K. Unnikrishnan and K. K. Parhi are with the department of Electrical and Computer engineering at the University of Minnesota, Minneapolis, MN 55455 USA (e-mail: unnik005@umn.edu, parhi@umn.edu).

This work has been supported in part by the National Science Foundation under Grants CCF-1914749.

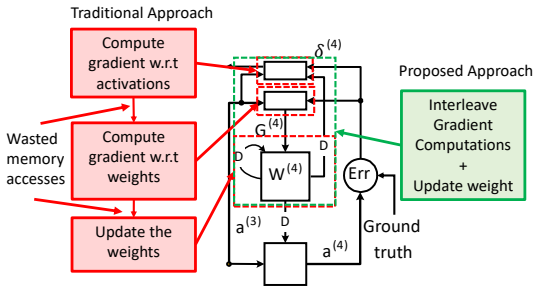

Fig. 2. Saving wasted memory accesses (red) by the proposed method of using an interleaved gradient computation and weight update (green).

convolutional layer during backpropagation, and (iv) it does not change any computed values, ensuring the accuracy of the neural network is unaffected.

This paper is an expanded version of [10] that proposed gradient interleaving within the FC layer. However, [10] was not directly applicable to the convolutional layers. Unlike the fully connected layers, the weights for the convolution layers correspond to tensors. These tensors are flattened and mapped to systolic arrays. The sizes of the systolic arrays for computing the gradients with respect to the weights and the activation functions are not the same. Furthermore, the input patterns to these arrays are also different. Interleaving these two computations in the same systolic array is impossible. The novelty of the paper lies in reformulating the computation of the gradients with respect to the weights in a different flattened form so that this can be mapped to a systolic array of the same size as the computation of the gradient with respect to the activation function. This reformulation is the key to achieving gradient interleaving for the convolutional neural network.

In addition to developing a novel architecture for interleaving the gradients for the convolutional layers, the following are the key contributions of this extension work:

1) We propose a new configurable systolic array with a modified general matrix multiplication (GEMM) algorithm to map the interleaved scheduler in the convolutional layer to hardware.
2) The proposed gradient interleaving approach is demonstrated for training common deep neural networks such as EfficientNet, VGG, ResNet, DenseNet, and Inception.
3) We show how interleaving in the convolutional layer eliminates the need for expensive im2col operations and provides greater processor utilization by exploiting null operations during striding.

The remainder of the paper is organized as follows. Section II focuses on the backpropagation equations and how gradient interleaving can be applied to the FC layers. Section III describes how to apply interleaved gradients to the convolutional layers. Section IV describes the changes to the hardware design of systolic arrays to support interleaved scheduling. Section V evaluates the proposed methodology. Section VI describes the related work in the field. Finally, in Section VII, we summarize the paper's main conclusions.

II. Backpropagation and Gradient Interleaving for FC Layers

The backpropagation algorithm, gradient descent, and derivatives are the foundations for training the most commonly available DNNs. However, these consume significantly more energy compared to inference. Therefore to optimize training, we examine the formulation of the backpropagation algorithm.

A. Backpropagation

Fig. 1 illustrates the dataflow graph on a four-layer fully-connected neural network. The lower half of the dataflow graph shows the forward pass computations, while the upper half shows the backward pass computations. As shown in Fig. 1, multiple nested feedback loops exist in the network. These feedback loops are the reason system-level techniques such as pipelining do not work, as delays *cannot* be introduced in a feedback loop system without affecting the output. The iteration period in recursive computing systems has a fundamental lower bound, referred to as the *iteration bound* [18].

Consider the computations in a single loop of Fig. 1. In the forward pass, the output of the linear function ($\boldsymbol{z}^{(l)}$) is obtained from a matrix-matrix multiplication between the activation output of the previous layer ($\boldsymbol{a}^{(l-1)}$) and the weights ($\boldsymbol{W}^{(l)}$) of the current layer (l). This is described by Eq. (1). Then, a nonlinear activation function $f(.)$ such as ReLU, sigmoid, or tanh is applied to compute the post-activation output of the layer ($\boldsymbol{a}^{(l)}$) as described by Eq. (2). These steps represent a single layer, and modern networks are created by stacking multiple layers. The final layer calculates the final value for regression or the confidence probabilities for classification.

$$\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)}\boldsymbol{a}^{(l-1)} \tag{1}$$

$$\boldsymbol{a}^{(l)} = f(\boldsymbol{z}^{(l)}) \tag{2}$$

The second part of the training loop is the backward pass, it is the process of propagating the gradient of the error backward through the network. The loss function is obtained by calculating the difference between the predicted output of the network and the *ground truth* with a loss metric such as mean squared error or cross-entropy loss. This loss is then *backpropagated* through the network with the help of the chain rule. This backpropagation step consists of three major operations: (i) computing the gradient of the loss function w.r.t. the activation function ($\boldsymbol{\delta}^{(l-1)}$), Eq. (3), (ii) computing the gradient of the loss function w.r.t. the weights ($\boldsymbol{W}^{(l)}$), Eq. (5), and (iii) updating the weights, Eq. (6). These three tasks are highlighted in red in Fig. 2. Traditional approaches treat these computations as independent operations, which is inefficient. We propose a single solution that can perform all operations together, as highlighted in green in Fig. 2. The backpropagation equations are given by:

$$\boldsymbol{\delta}^{(l-1)} = \frac{\partial E}{\partial \boldsymbol{z}^{(l-1)}} = \frac{\partial E}{\partial \boldsymbol{a}^{(l-1)}} \odot f'(z^{(l-1)}) \tag{3}$$

$$\frac{\partial E}{\partial a^{(l-1)}} = (\boldsymbol{W}^{(l)T}\boldsymbol{\delta}^{(l)}) \tag{4}$$

$$\boldsymbol{G}^{(l)} = \left(\frac{\partial E}{\partial \boldsymbol{W}^{(l)}}\right) = \boldsymbol{\delta}^{(l)}a^{(l-1)T} \tag{5}$$

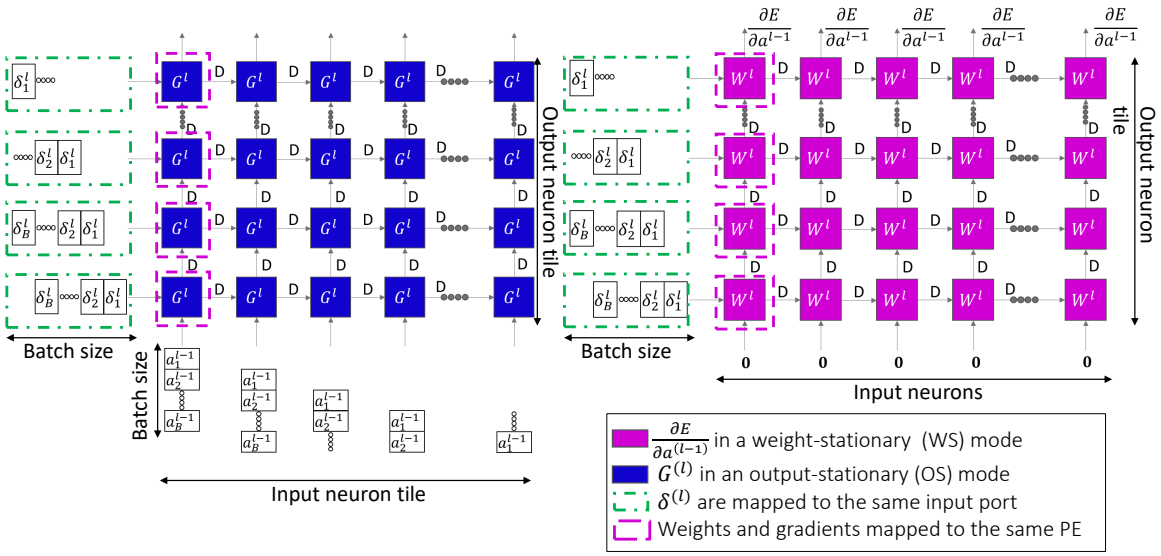

Fig. 3. Systolic arrays that compute $G^{(l)}$ in an OS mode and $\frac{\partial E}{\partial a^{(l-1)}}$ in a WS mode, modified from [10].

$$\boldsymbol{W}^{(l)}(k+1) = F\left(\boldsymbol{W}^{(l)}(k), \boldsymbol{G}^{(l)}(k)\right) \tag{6}$$

where E represents the loss function, $\boldsymbol{\delta}^{(l)}$ represents the gradient of the error backpropagated to layer l, f' represents the derivative of the activation function and $\boldsymbol{G}^{(l)}$ represents the gradient of the error w.r.t. weights. The notation $\odot$ represents the Hadamard product of matrices and ^{T} represents the transpose of the matrix. Although Eqs. (1) to (6) are defined using a FC layer, these can be extended to convolutional layers by replacing the linear multiplication operation with the convolution operation. An overview of the dataflow graph with the corresponding equations and their dimensions is described in Section S1 and Fig. S1 in the Supplementary Material.

When implemented in hardware, these equations are mapped to matrix-matrix or matrix-vector multiplication operations. Systolic arrays [19] and other spatial architectures have emerged as an alternative to general-purpose computing and are highly optimized for matrix-matrix operations. These architectures have inherent advantages with their spatial dataflow pattern allowing for more efficient data reuse. A dataflow for this array describes which data remain stationary and which data flow through the array. These dataflow patterns can broadly be classified based on which data is stored within the systolic array, namely, *input-stationary* (input feature map), *output-stationary* (OS) (partial output sums), *weight-stationary* (WS) (layer weights), and *row-stationary* (filter rows) [20]. Row-stationary (RS) is a specialized case of weight-stationary where the array stores the convolutional filter row rather than individual weights. Matrix-matrix operations are the focus of this paper as they constitute a significant portion of the computation time [21]. The computations of the activation function, its derivative, the associated Hadamard product, and pooling are often assigned to special blocks or special function units (SFUs) [22]–[24]. A detailed discussion of the SFU for the backward pass can be found in Section S1.1 of the Supplementary Material.

Further subsections will contrast the proposed approach against the traditional mapping of the backpropagation algorithm on systolic arrays. Although we do not directly explore the forward pass, the accelerator maps the matrix-matrix computation of the forward pass as a WS or OS operation in the systolic array.

B. Computation of $\boldsymbol{G}^{(l)}$ in an FC layer

The architecture shown in Fig. 3(left) is considered for the operation in Eq. (5). The figure shows an array consisting of $Mtile \times Ntile$ processing elements (PEs), where $Mtile$ and $Ntile$, respectively, represent the systolic array's horizontal and vertical dimensions. The markings for input neurons, output neurons, and batch size show how those dimensions map to the systolic array through tiling. output The array operates in an OS mode, where the output partial sums ($\boldsymbol{G}^{(l)}$) accumulate in the PE. It takes $Ntile$ cycles where the systolic array loads $Mtile$ words per cycle. The PEs are interconnected in horizontal and vertical directions.

$\delta^{(l)}$ is passed into the array along the left edge, and the activation outputs $a^{(l-1)}$ are input to the bottom edge of the array. The gradients with respect to the weights are accumulated *in-place* for the minibatch of B. After the matrix multiplication completes, the array contents are shifted out, requiring additional $Ntile$ cycles where $Mtile$ gradients are unloaded per cycle from the systolic array.

C. Computation of $\delta^{(l-1)}$ in an FC layer

To compute Eq. (4) the architecture shown in Fig. 3(right) is considered. The markings for input neurons, output neurons, and batch size show how those dimensions map to the systolic array through tiling. The array is set up in a WS mode, where one of the inputs, $\boldsymbol{W}$, is held constant inside the cell's local memory. The weights are first loaded into the array from the edge. Although we only consider a single weight stored in each processing element in this example, the same technique can be extended to process multiple weights simultaneously. Once the weights are loaded, $\delta^{(l)}$ is loaded along the left edge, staggered by a clock cycle for each row. Once the results are calculated

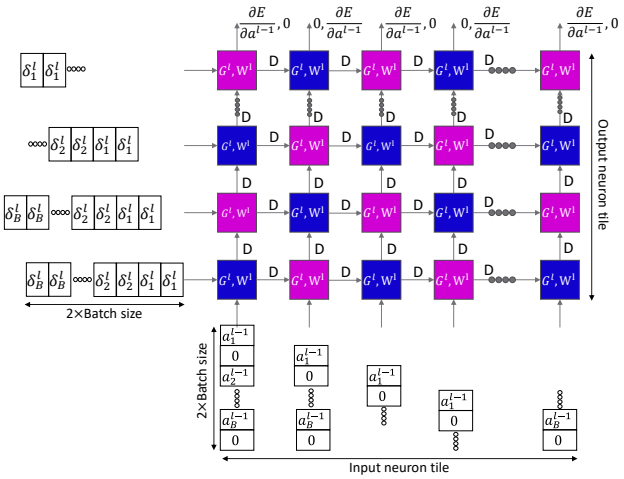

Fig. 4. Systolic array operation in the proposed interleaved mode.

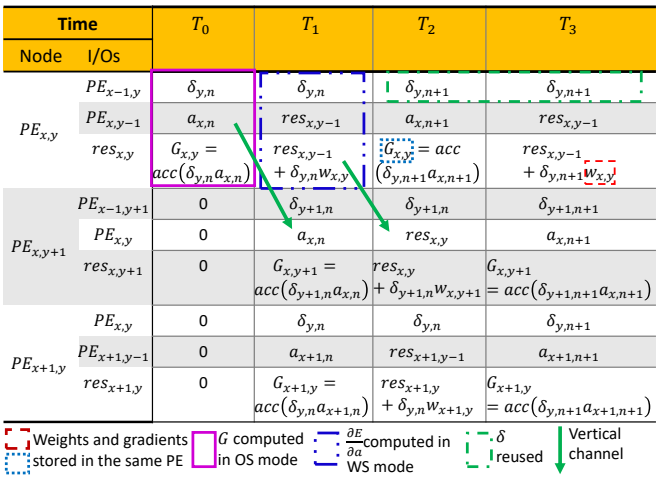

Fig. 5. Dataflow in the systolic array in both x and y directions, modified from [10]. The figure highlights the reuse of the vertical communication channel between $a_{x,n}$ and $res_{x,y}$. Each PE toggles between WS and OS modes every cycle.

in each PE, the partial sums are accumulated vertically in the array. A pipelined architecture allows for a simple PE design with a low critical path delay. The entire process requires B cycles to process the entire minibatch of size B. Once these calculations are complete, the array contents are no longer required and are discarded.

D. Interleaved gradient scheduler

Traditionally hardware accelerators implement Eqs. (4) to (6) sequentially. However, multiple avenues of data reuse among the equations can be exploited to reduce memory access and energy consumption. One such inefficiency is that prior approaches do not exploit the reuse of $\boldsymbol{\delta}^{(l-1)}$ between Eqs. (4) and (5), $\boldsymbol{W}^{(l)}$ between Eqs. (4) and (6), and $\boldsymbol{G}^{(l)}$ between Eqs. (5) and (6). Note that most high-level synthesis [25]–[27] systems cannot automate this reformulation.

Interleaving has been extensively studied in signal processing systems to reuse hardware and computations across different data points [16], [17], [28]. Based on requirements in Sections II-B and II-C, this paper proposes to interleave the gradient computation of Eqs. (4) and (5) enabled by a novel *configurable systolic arrays* that can switch between OS and WS modes every cycle. In both cases, $\delta^{(l-1)}$ is loaded through the left edge of the array allowing reuse of $\delta^{(l-1)}$ across both equations.

E. Interleaved gradients for the FC layer

The proposed interleaving of the systolic array is shown in Fig. 4. The dimensions in the figure indicate how the input data are mapped to the systolic array. The figure shows that the computation performed is identical to Fig. 3 with the PEs alternating between WS and OS modes. Fig. 5 shows the data flow and operations in the array along the vertical (y) and horizontal (x) directions. Note that $PE_{x,y}$ and $PE_{x,y+1}$ are adjacent along the vertical direction. Similarly, $PE_{x,y}$ and $PE_{x+1,y}$ are adjacent along the horizontal direction. T_0, T_1 *etc*. represent consecutive time steps. The reuse of δ effectively reduces the number of accesses to the on-chip memory by $B \times \lfloor \frac{N}{Ntile} \rfloor \times \lfloor \frac{M}{Mtile} \rfloor \times Ntile$. Here, $N \times M$ is the dimension of the weight matrix.

Finally, once the gradients are obtained, the weights can be updated as per Eq. (6). From a careful observation of the computations of Eq. (4) and *reformulated* Eq. (5), it can be seen that each element of the gradient matrix that is generated is *located* in the same PE as the corresponding element of the weight matrix. The gradient $\boldsymbol{G}^{(l)}$ is a temporary variable that is generated and must ultimately update the weight matrix; however, due to the conventional approach, it must be stored after creation and recalled from the memory to update the weight as per Eq. (6). However, with the proposed *configurable systolic arrays*, the weight matrix can be updated *in-place* without the need to store or retrieve the temporary variable $\boldsymbol{G}^{(l)}$. Thus, a further $3 \times \lfloor \frac{N}{Ntile} \rfloor \times \lfloor \frac{M}{Mtile} \rfloor \times Ntile \times Mtile$ accesses are saved. The overall reduction in memory accesses is given by:

$$\lfloor \frac{N}{Ntile} \rfloor \times \lfloor \frac{M}{Mtile} \rfloor \times Ntile \times (3 \times Mtile + B) \quad (7)$$

The in-place weight update adapts the transpose of the weight matrix, $W^{(l)T}$. However, we may desire to return the original variable and not the transposed form. This is easily handled in the case of systolic arrays as it can implicitly perform a transpose operation when unloading data. $W^{(l)T}$ that is normally read out through the vertical channel can instead be read out through a horizontal channel to get $W^{(l)}$.

III. Gradient Interleaving for Convolutional Layers

This section describes the operations in the convolutional layers, their mapping to systolic arrays, and the interleaving of the gradients.

A. Convolutional layer computations

The equations for the convolutional layer can be written in a similar manner to Eqs. (4) to (6) with the matrix multiplication operations replaced by convolution operations. We explore different ways to optimize training by exploring the reuse of variables between different computations. Fig. 6 describes

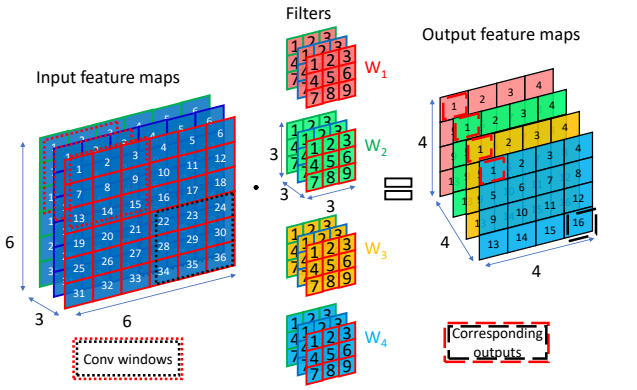

Fig. 6. A 6×6 input map with three channels is convolved with four $3 \times 3 \times 3$ filters in the forward pass to generate four output feature maps. The highlighted boxes show the convolution window in the input feature map and the corresponding calculated output pixel. The border color of the cell indicates the input feature map dimension, and the color fill of the cell indicates the output feature map dimension.

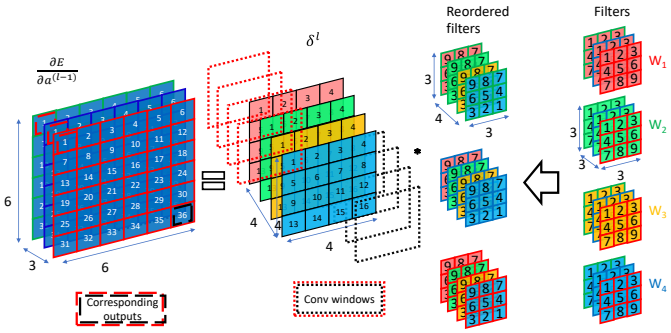

Fig. 7. Convolution between the filters and a 4×4 $\delta^{(l)}$ map to calculate $\delta^{(l-1)}$. There are 4 $\delta^{(l)}$ channels corresponding to the four filters. Reordering results in three filters corresponding to the number of channels for $\delta^{(l-1)}$. The highlighted red box shows the convolution window and corresponding output pixel. The border color of the cell indicates the input feature map dimension, and the color fill of the cell indicates the output feature map dimension.

the forward pass of the convolutional layer. Each convolution operation is an element-wise multiplication between the filter, $w^{(l)}$ (W_1 to W_4) and the convolution window of the input, $a^{(l-1)}$, followed by a summation that corresponds to a single output pixel. The red and black dotted lines represent example convolutional windows and their corresponding output pixel.

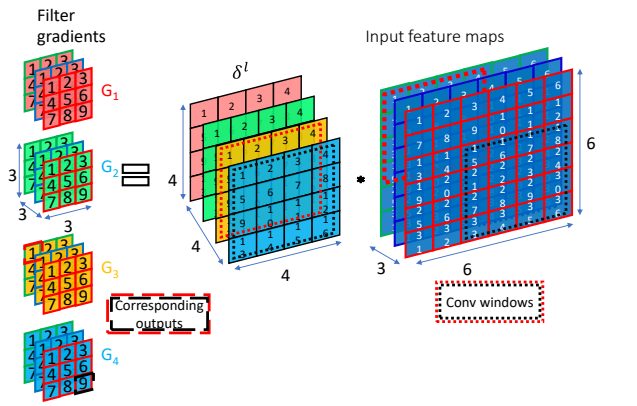

Fig. 8. Each channel from the input feature map is convolved with each channel of $\delta^{(l)}$ to calculate the gradients of the filter weights $G^{(l)}$. The border color of the cell indicates the input feature map dimension, and the color fill of the cell indicates the output feature map dimension.

The output feature maps are color-coded as per the weights that generated them. The convolution window is then stridden across the input image until all output pixels are calculated and repeated for all filters and input images.

The gradient of the error propagated to the previous layer, $\delta^{(l-1)}$, is calculated by a convolution operation between $\delta^{(l)}$ and the 180° rotated filter channel, i.e., the upper right element moves to lower left and upper left element to lower right. Additionally, the same input channel from all filters are merged to form the equivalent filter for backpropagation. This mapping reduces the problem to a simple convolution-like operation like the forward pass as detailed by the reordering step in Fig. 7. The calculation of the gradient filter, $G^{(l)}$, is described in Fig. 8. The error from the subsequent layer, $\delta^{(l)}$, is convolved with each channel of the input from the previous layer, $a^{(l-1)}$.

Most general-purpose processors and accelerators convert the convolution operation into a matrix-matrix operation to exploit the hardware optimizations for the general matrix multiplication (GEMM) algorithm. This operation, called image to column (*im2col*), takes the convolution windows, unrolls the different windows, and arranges them in a Toeplitz matrix [20] format. The filters are also unrolled similarly and stacked to form a matrix. The convolution operation can be mapped to a matrix-matrix multiplication operation like the fully-connected layer. Thus each convolution operation also requires an expensive call to an im2col block or transpose, either in the host CPU or a dedicated memory manipulation block [24].

B. Mapping backpropgation convolutions to systolic arrays

A major difference in the convolution operation over the fully connected layers is mapping the operations to the array. To compute $\delta^{(l-1)}$ as shown in Fig. 7, consider the weight stationary mapping shown in Fig. 9. The array consists of $X \times Y$ processing elements (PEs) where X and Y are the dimensions of the systolic array in the horizontal and vertical directions, 3×36 in this example. D is a register delay between the PEs in the horizontal direction. In this convolution operation, $\delta^{(l)}$ can be visualized as a feature map and its corresponding convolution windows are passed along the left edge of the systolic array. In this example, $\delta^{(l)}$ is a 36×36 matrix. The weights, $\boldsymbol{W}$, are held constant within the PE's local memory. In this example, $W^{(l)}$ is a 3×36 matrix. The complete input matrices used in this example are shown in Fig. S5 of the Supplementary Material. The partial sums of $\frac{\partial E}{\partial a^{(l-1)}}$ are accumulated vertically in the array, resulting in a 3×36 matrix for this example. The entire process requires $B \times \#ConvolutionWindows$ of $\delta^{(l)}$ cycles to process the entire minibatch of size B.

Similarly, to compute $G^{(l)}$ as shown in Fig. 8, consider the output stationary mapping shown in Fig. 10. The array consists of 27×4 processing elements (PEs) in this example, corresponding to the size of the output $G^{(l)}$. In this convolution operation, $\delta^{(l)}$ is flattened and passed as the filter along the left edge of the systolic array. Each row corresponds to one flattened channel of $\delta^{(l)}$. This represents a 16×4 matrix in the above example. The convolution windows of the input

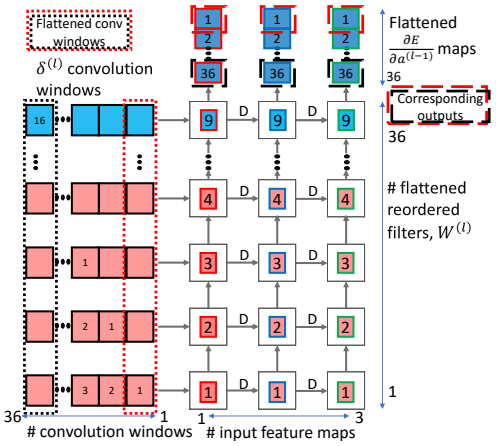

Fig. 9. Mapping of the $\delta^{(l-1)}$ computation to a systolic array in WS mode. The flattened filters are stored as columns in the array and convolution windows of $\delta^{(l)}$ are passed into the array from the left edge. The computed result is taken from the top edge. The red and black dotted column boxes on the left show the mapping of the convolution windows from Fig. 7.

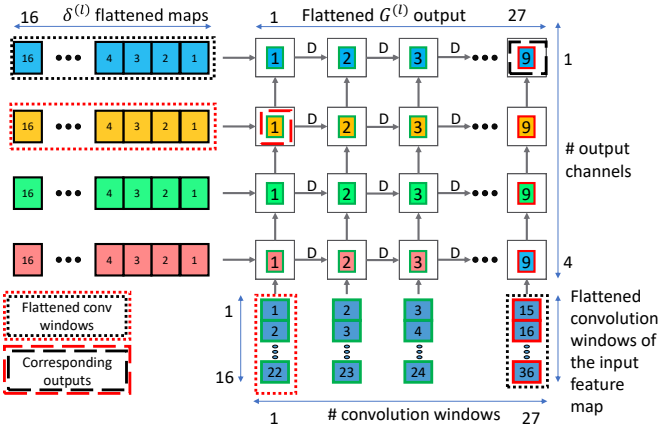

Fig. 10. Mapping of the $G^{(l)}$ computation to a systolic array in OS mode. The flattened map of $\delta^{(l)}$ is passed into the array from the left edge as a filter. The convolution windows of the input feature maps are passed in from the bottom edge. The computed result is stored within the PE. The red and black dotted column boxes on the left and bottom show the mapping of the convolution windows from Fig. 8.

feature map are flattened and passed along the bottom edge of the systolic array. This represents a 27×16 matrix in the above example. The complete data matrices used as inputs in the example can be found in Fig. S6 of the Supplementary Material. The input feature maps are staggered by a clock cycle per column and the matrix computations require 16 cycles, the size of a flattened delta map, per map to complete. The computed results are accumulated in place for the minibatch of B after which the array contents are shifted out.

C. Interleaved gradients for the convolutional layer

While the initial mapping of the backpropagation computations in Section III-B is reminiscent of the form used to interleave gradients in the fully connected layers, this is not straightforward in the convolution case. This is understood by observing how $\delta^{(l)}$ is incident to the systolic arrays in the two operations. When calculating $G^{(l)}$ in Fig. 8, $\delta^{(l)}$ is treated as a filter however, when calculating $\delta^{(l-1)}$ in Fig. 7, $\delta^{(l)}$ is treated as a feature map.

The challenge of interleaving Fig. 9 for computing $\delta^{(l-1)}$ and computing $G^{(l)}$ from Fig. 10 to the same array is because the systolic arrays that are used in both computations are of different sizes. For example, $\delta^{(l-1)}$ requires an array of size 3×36 and $G^{(l)}$ requires an array of size 27×4. The second challenge lies in the fact that the flattened $\delta^{(l)}$ inputs on the left side of the systolic arrays are also of different dimensions for the two computations. For example, in the computation of $\delta^{(l-1)}$ the $\delta^{(l)}$ input is of dimension 36×36 whereas that for computing $G^{(l)}$ is 16×4. Thus, interleaving the computation of the two gradients into one systolic array is impossible. However, this impossibility can be overcome by reformulating the computation of $G^{(l)}$ such that this can be computed using the same systolic array used to compute $\delta^{(l-1)}$. Furthermore, the input pattern for $\delta^{(l)}$ should also be in the same format as for $\delta^{(l-1)}$. We observe that this is possible by computing the gradients of the weights in a reordered form instead of the original form. In the original form, the gradients correspond to 4 filters, where each filter corresponds to a $3 \times 3 \times 3$ tensor. They are computed as 3 filters in the reordered form, where each filter corresponds to a $3 \times 3 \times 4$ tensor. Fortunately, the $\delta^{(l)}$ used in computing $\delta^{(l-1)}$ shown in Fig. 9 is the same pattern needed to compute the weight gradients in the reformulated form. Fig. 11 describes the input pattern of $\delta^{(l)}$; this is also the same as used in Fig. 9 (also the same format as shown in Fig. S5). Now it is possible to interleave the computations of the two gradients in the same systolic array, as shown in Fig. 12. No other approach has explored how to fuse these two computations. Note that, like the fully connected layer architecture shown in Fig. 4, the $\delta^{(l)}$'s are held for two clock cycles in Fig. 12. Each PE alternates between the WS mode for the $\delta^{(l-1)}$ computation and OS mode for the $G^{(l)}$ computation.

The proposed architecture has three advantages. First, as we reuse the $\delta^{(l)}$ with no additional overhead required to preprocess the input, this eliminates the need to reload $\delta^{(l)}$. The input $\delta^{(l)}$ for computing G as shown in Fig. 11 is identical to the input $\delta^{(l)}$ shown in Fig. 9. Furthermore, we also eliminate the need to generate $\delta^{(l)}$ in two different formats. Second, we can load a flattened $a^{(l-1)}$ directly from memory without creating convolutions windows. This eliminates any calls to im2col for the $a^{(l-1)}$, arguably simplifying the control of the architecture. Third, the output from the $G^{(l)}$ computation exists in the same PE as its corresponding $W^{(l)}$ matrix element.

Use of the common input $\delta^{(l)}$ leads to significant memory access savings as the two gradient computations can be interleaved without the need to load the data twice from the local memory. This also leads to a reduction in the number of cycles required due to the reuse of inputs form loading and unloading overheads. As with the case of the FC layer, the reuse of δ effectively reduces the number of accesses to the on-chip memory by $B \times \lfloor \frac{F}{X} \rfloor \times \lfloor \frac{N_F}{Y} \rfloor \times X \times N$, where $X \times Y$ is the systolic array size, B is the batch size, N is the number of convolution windows, F is the flattened filter size and N_F is the number of filters. Similarly, the in-place update of the weight matrix further reduces $3 \times \lfloor \frac{F}{X} \rfloor \times \lfloor \frac{N_F}{Y} \rfloor \times X \times Y$ memory accesses. The overall reduction in memory accesses

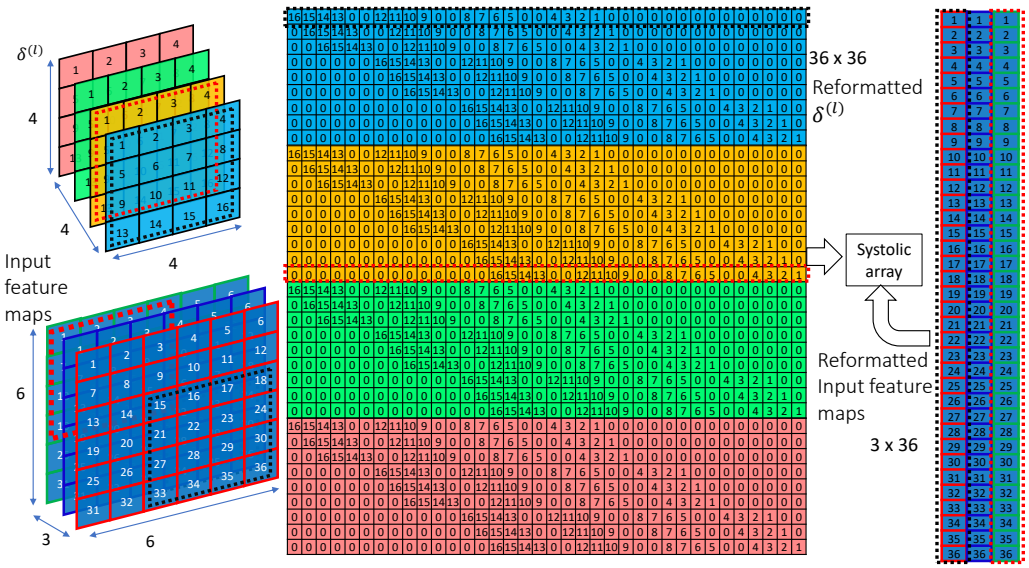

Fig. 11. Modified input matrices for the output stationary calculation of $G^{(l)}$ in Fig. 10. The 36×36 matrix in the center is the modified representation of the $\delta^{(l)}$ map shown on the top left. This is the same format as used in Fig. 9 (also the same format as shown in Fig. S5). The 3×36 matrix on the right is the modified representation of the input feature map shown on the bottom left that is created by flattening. The red and black dotted boxes show how the convolution windows on the left are mapped to the input matrices.

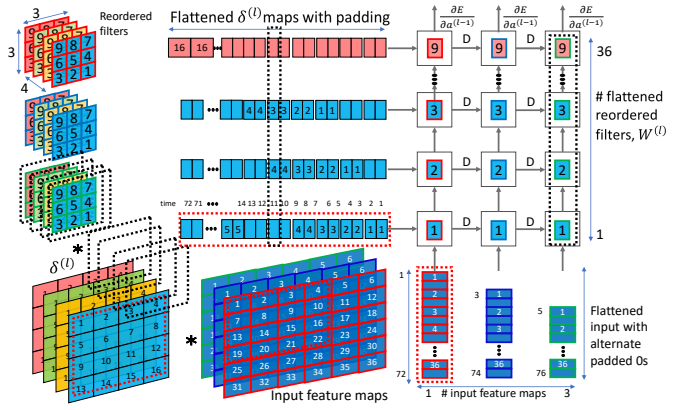

Fig. 12. Interleaving of operations for the convolutional layer. The black dotted boxes on the reordered filters and $\delta^{(l)}$ represent one of the convolution windows to compute $\delta^{(l-1)}$. The corresponding weight stationary mapping of that window is shown on the flattened $\delta^{(l)}$ and reordered filters $W^{(l)}$. Similarly, the red dotted boxes represent $\delta^{(l)}$ and the input feature map of one convolution window for computing $G^{(l)}$. The corresponding output stationary mapping of that window is shown on the flattened $\delta^{(l)}$ and flattened input feature maps.

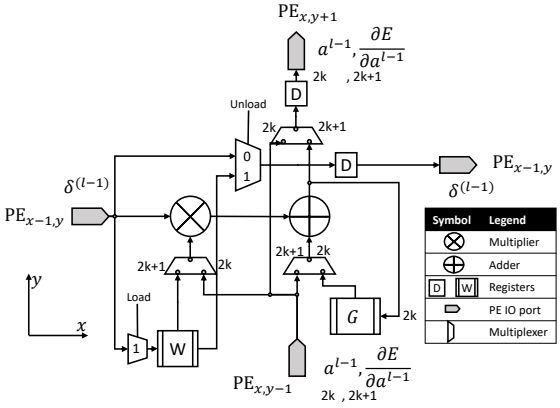

Fig. 13. Architecture of the configurable systolic array processing element.

is given by:

$$\lfloor \frac{F}{X} \rfloor \times \lfloor \frac{N_F}{Y} \rfloor \times X \times (3 \times Y + B \times N) \tag{8}$$

Additionally, for both the input features, $a^{(l-1)}$, and error gradients, $\delta^{(l)}$, we do not need to compute the convolution windows. This eliminates the im2col step in the backward pass, as we only need to store the flattened map. For information on the handling of stride and padding, refer to Section S2 in the Supplementary Material.

IV. Implementing Interleaved Gradient and Configurable Systolic Arrays

This section describes how gradient interleaving for the FC and the convolutional layers can be incorporated into a modified GEMM algorithm and the hardware modifications for the PEs to support the proposed method.

A. Configurable systolic array processing elements

A new configurable PE is designed to support the proposed interleaved architecture. There are two basic modes of operation that the PE must support: a WS mode to compute $\delta^{(l-1)}$ and an OS mode to compute $\boldsymbol{G}^{(l)}$ as shown in Fig. 13. An additional requirement is that the system should switch between the two modes every cycle. Therefore, any control logic to the circuit must be straightforward and flexible to support both modes.

We can interleave the two modes without additional overhead for arithmetic units. Furthermore, to enable fast switching, we use a one-bit counter to alternate the mode every clock cycle to enable fast switching. This counter simplifies the control circuit to only choose between the odd or even phases for alternating rows or columns of the array. This is implemented as switches or multiplexers in the circuit that indicate which path is selected in the $2k$ or even phase, and

$2k+1$ or odd phase. In the even phase, the PE operates in an OS mode, and in the odd phase, the PE operates in a WS mode. Therefore, the proposed PE architecture in Fig. 13 can achieve the above goal with a minor overhead of 4 multiplexers. At the array level, we can determine which PEs are in phase with each other for the fully connected layer as shown in Fig. 4. The pattern is derived by analyzing the presence of delay elements or registers on horizontal or vertical edges. A delay element toggles the phase of the adjacent PE. Thus, once an operation starts, we only need a 1-bit counter to drive all the inputs of the array, directly for one set and as the complement for the other set. We can derive a similar pattern for the convolution layers by analyzing Fig. 12.

In terms of the inter-PE communication, the horizontal connection need not be modified as, with both modes, only $\delta^{(l)}$ is transmitted. However, for the vertical connections, extra hardware is needed to multiplex between transmitting $a^{(l-1)}$ and the partial sum of Eq. (4). Nevertheless, given that these will only be transmitted in alternate cycles, the only requirement is that the bus is designed to be sufficiently large to accommodate the larger of the two.

The interleaving techniques were designed not to introduce any new complex control logic at the system level. In terms of the input, the new systolic array will support inputs on the left and bottom edges. This is the same requirement as an OS architecture. The only additional support is one level of multiplexing to preload the weights into the array. The outputs are streamed out continuously while it's processing, and the internally stored partial sums are read out at the end of processing a tile. For the fully connected layers, the data is not modified in any way, and interleaving is possible by latching the value of $\delta^{(l)}$ for two cycles. Thus no additional overhead is required.

B. Mapping and tiling computations

The proposed hardware can operate in two different modes: either a direct matrix-matrix multiplication mode or an interleaved scheduler mode. The proposed interleaved scheduler at the system level is summarized in Algorithm 1. The inner computational loops are fused at the algorithm level to interleave all three computations. These structures follow a hierarchical GEMM format to exploit parallelism and maximize cache efficiency. This can serve as a baseline on how to integrate the proposed hardware into existing software frameworks like PyTorch or Tensorflow. Here the tile size for this algorithm is chosen as $Mtile$ and $Ntile$ to match the dimensions of the systolic array. The proposed approach will fit into a hierarchical system as it only optimizes the inner loops dedicated to specialized hardware such as tensor cores [22], [23]. At a tile level, the algorithm will generate the result for $G_{m,n}$ for the entire tile after processing all elements in the $Otile$ loop, where $Otile$ represents tiling in the input channel dimension O. Similarly, it can generate δ_n after processing all elements in the $Ntile$ loop.

C. Architecture

The goal of the proposed approach is to be viewed as an enhancement to existing systolic arrays with minimal overhead. Therefore, we design a system-level baseline architecture to evaluate the effectiveness of the proposed approach. We use [24] to determine the overall architecture of the systolic array. The design itself is not optimized for area or throughput but rather serves as a basis to evaluate the overheads associated with the proposed approach. Fig. 14 shows the overview of the baseline architecture. First, the architecture assumes four local memories to store the weights, activations, deltas, and results. Second, the input and output buffers process the inputs of the systolic array and the memory controller interfaces the buffers, local memory, and external memory. Finally, the PE control determines the operations within the PEs such as loading, unloading, or mode of computations like WS or OS. For the proposed approach we modify the PE array to the new proposed configurable PE and we modify the PE control to support the new InterGrad mode for the backward pass.

Algorithm 1 Integration of interleaved scheduling for back-propagation in a modified GEMM algorithm.

Input: $\boldsymbol{W}^{(l)}$; $\boldsymbol{\delta}^{(l)}$; $\boldsymbol{a}^{(l-1)}$ $\boldsymbol{G}^{(l)}_{init}$
for $m \leftarrow 0$ to $M-1$ by $Mtile$ **do**
  for $n \leftarrow 0$ to $N-1$ by $Ntile$ **do**
    for $o \leftarrow 0$ to $O-1$ by $Otile$ **do**
      // Proposed new interleaved Scheduling block
      InterGrad($\boldsymbol{W}^{(l)}_{m,n,o}$, $\boldsymbol{\delta}^{(l)}_{m,n,o}$; $\boldsymbol{a}^{(l-1)}_{m,n,o}$,$\boldsymbol{G}^{(l)}_{init\ m,n,o}$,"compute")
    end for
    if UPDATE WEIGHTS **then**
      $\boldsymbol{W}_{m,n}$ = InterGrad("update and unload weights")
    end if
    if UNLOAD INTERMEDIATE GRADIENTS **then**
      $\boldsymbol{G}_{m,n}$ = InterGrad("unload gradients")
    end if
  end for
  $\boldsymbol{\delta}^{(l-1)}_{n}$ = InterGrad("unload δ")
end for
Output: W

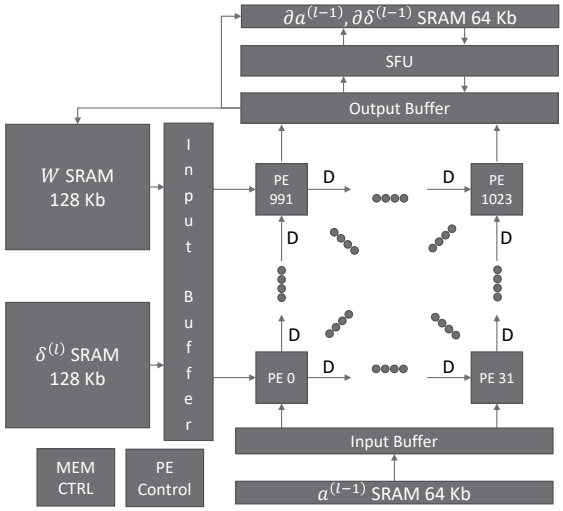

Fig. 14. Architecture to evaluate effectiveness of the proposed enhancements.

V. Evaluation of the Proposed Approach

A. Methodology

We evaluate the advantages of the proposed methodology for the fully-connected (FC) and convolution layers and compare them against traditional dataflow. For the FC layer, we use the structure shown in Fig. 1 as a reference. Each experiment varies the number of neurons or network size of the FC layer,

the batch size of the inputs, and the systolic array size. For the convolutional layers, we use the convolutional layers of VGG16, ResNet, and InceptionV1. For the forward pass, the proposed dataflow is identical to the baseline in terms of both memory accesses and computation cycles. Though the forward pass is not optimized, it is included in the evaluation of the entire system's performance. The conversion from images to a matrix-matrix multiplication is handled by the im2col block. This is detailed in [14], [20], [24] and is common to the baseline and proposed approach.

We evaluate the performance while varying array size and batch size. We developed a simulation tool over the open-source python-based NN simulation framework SCALE-sim [14] to evaluate the proposed method. The tool is built to calculate statistics like the number of cycles and on-chip SRAM memory access for the inference and backpropagation operations. The tool supports both traditional dataflows as well as the proposed dataflow. We evaluate the proposed method and compare it with two baselines: weight stationary (WS) and output stationary (OS). We model the WS approach by mapping all the backpropagation computations to the systolic array. The tool determines the number of cycles, memory accesses, and computations based on Fig. 3(right) for the FC layer and Fig. 9 for the convolution layer. Similarly, the tool models the OS approach based on Fig. 3(left) for the FC layer and Fig. 10 for the convolution layer.

The tool is designed to model the proposed dataflow as follows. The computation is first split into tiles and allocated to the PEs based on Algorithm 1. The tool processes each tile determining the number of cycles and memory accesses based on Fig. 4 for an FC layer or Fig. 12 for a convolution layer. Finally, this is processed cycle-wise to determine the number of multiply and accumulate operations (MACs), internal register reads/writes, inter-PE communication, and on-chip SRAM reads/writes from the array.

All results that report memory accesses and memory accesses savings refer to accesses to the SRAM unless stated otherwise. For the convolutional layers, we additionally report the energy consumption based on a normalized energy cost [20], [29]. The referenced energy model is normalized to a multiply and accumulate (MAC) computation. As a result, the energy consumption for communication, internal register reads/writes, and accessing the on-chip SRAM are $2\times$, $1\times$, and $6\times$ of a MAC operation, respectively. Additionally, The energy model accounts for the overheads associated with the new configurable PE.

The proposed work aims to reduce the overall number of accesses to the memory as measured in SRAM access without regard to DRAM access and latency. Though the proposed method does not directly target DRAM access, it will significantly reduce the DRAM access, especially in cases where all the data does not fit in the SRAM. Furthermore, reusing variables within the systolic array ensures that even partially loaded contents to the SRAM are used completely before being ejected. We evaluate the backpropagation algorithm at two granularities: a single layer and as part of a larger DNN. This evaluation is performed first on the FC layers and then validated on the convolutional layers.

B. Single-layer scheduling for FC layer

For a single layer evaluation, the innermost loop in Fig. 1 (loop 4) is used. We evaluate the three traditional dataflow models for comparison, i.e., WS and OS. Additionally, we use a third baseline called flex, which picks the best metric between OS and WS. This is only done to highlight the benefits of InterGrad as, in a multi-layer scenario, the best approach for one metric did not always lead to the best results in others. In traditional dataflow, we compute the backward pass equations, Eqs. (4) to (6), as separate matrix-matrix operations. In the proposed interleaved scheduler methodology, a single equivalent time is stated for processing all equations in an interleaved manner. Computations for Eq. (2) are not shown but are assumed to be processed elementwise separately.

TABLE I
INTERLEAVED GRADIENT SCHEDULER (INTERGRAD) VS. TRADITIONAL DATAFLOW FOR EACH EQUATION OF BACKPROPAGATION IN FC LAYERS. FLEX REFERS TO FLEXIBLE SCHEDULE THAT CAN PICK THE BEST DATAFLOW FOR EACH EQUATION.

EQ.	Cycles ($\times 10^3$)				Memory Accesses ($\times 10^6$)			
	OS	WS	Flex	Inter Sched	OS	WS	Flex	Inter Grad
(1)	35.8	114.7	35.8	35.8	6.5	8.4	6.5	6.5
(4)	81.7	18.9	18.9		8.4	6.4	6.4	
(5)	65.7	65.7	65.7	98.2	12.6	12.6	12.6	12.6
(6)	35.8	114.7	35.8		6.5	8.4	6.5	
All	219.0	314.0	156.3	134.0	33.9	35.9	31.9	19.0

Table I breaks down the computation time and memory accesses for the proposed method and traditional dataflow by equation number. The single loop is evaluated on a 128×128 systolic array with a batch size of 64 and a 4096-neuron FC layer. It is observed that the backward pass dominates the overall computation time and memory requirement for the backpropagation loop. For example, looking just at the backward operations Eqs. (4) to (6), we see that the proposed method takes 18.5% fewer cycles than flexible architectures and over 46.4% fewer cycles compared to traditional dataflow. Similarly, the proposed method reduces the number of memory accesses by 50% in backward pass computations. Overall, for the entire loop, including the forward pass, the proposed method reduces 14.2% and 40.3% of cycles and memory accesses, respectively. While the latency of each backpropagation computation has increased, there is an overall increase in throughput as the total time to perform both computation is reduced. Additionally, the overall latency or layer level latency looking at both operations together is reduced.

TABLE II
THE NUMBER OF CYCLES AND MEMORY ACCESSES FOR AN $N \times N$ MLP LAYER WHILE VARYING N, MODIFIED FROM [10]. VALUES ARE NORMALIZED TO WS. INTERGRAD OUTPERFORMS THE BASELINE FOR ALL NETWORK SIZES.

N	**Cycles**			**Memory**		
	WS	**OS**	**InterGrad**	**WS**	**OS**	**InterGrad**
128	1.00	0.81	0.57	1.00	1.00	0.58
256	1.00	0.78	0.51	1.00	0.98	0.55
512	1.00	0.75	0.46	1.00	0.96	0.54
1024	1.00	0.72	0.44	1.00	0.96	0.53
2048	1.00	0.71	0.42	1.00	0.95	0.53
4096	1.00	0.70	0.42	1.00	0.95	0.53

Table II analyzes the effect of the network size on the proposed method's performance in terms of normalized number of cycles and memory accesses to the local on-chip SRAM. This is obtained by sweeping and evaluating different network sizes and batch sizes. For fixed network sizes, the values obtained are averaged across batch sizes and normalized to the value for WS. The proposed methodology reduces the number of cycles to compute this loop by 30%. Also, the proposed method reduces the number of single-loop memory accesses by 42%. This corresponds closely to Eq. (7) as the increase in memory accesses and savings for δ is proportional to the array size. Thus the savings are constant across all network sizes. Similarly, both the accesses and savings for the weight update step are proportional to the square of the array size.

TABLE III
SINGLE LOOP PERFORMANCE OF THE INTERLEAVED GRADIENT SCHEDULER (INTERGRAD) ON FC LAYERS VS. TRADITIONAL DATAFLOW APPROACHES WHILE VARYING BATCH SIZE (B)

	Batch Size	WS	OS	InterGrad	% Improve
Cycles ($\times 10^3$)	8	207.35	155.61	81.36	39.0%
	16	212.11	157.34	84.60	37.4%
	32	221.84	160.80	91.06	34.5%
	64	241.58	167.73	103.99	29.1%
	128	281.58	181.58	129.86	20.2%
Memory Accesses ($\times 10^6$)	8	20.47	20.27	10.34	48.9%
	16	21.49	21.10	10.96	47.5%
	32	23.54	22.76	12.20	45.0%
	64	27.64	26.08	14.69	40.7%
	128	35.84	32.71	19.73	34.5%

Table III analyzes the effect of the batch size on the performance. The values obtained are averaged for a different number of neurons. The proposed method uses between 20% and 39% fewer cycles across all batch sizes. The proposed method reduces the number of single-loop memory accesses between 34% and 48%. This matches the expected savings as the savings for δ scales with B, but the weight update step does not. At larger batch sizes, the savings from the weight update step remain constant; therefore, savings as a fraction of the overall memory accesses decrease. In general, for the fully connected layers, a small batch size can lead to under utilization of the systolic array. To ensure high utilization, the batch size must be chosen such that it is comparable to or greater than the dimensions of the systolic array. Furthermore, as the proposed approach interleaves the computations along the batch size dimension as shown in Fig. 4, this mitigates the effects of smaller batch sizes to some degree.

C. Application to the convolutional layers in CNNs

We compare the proposed approach against a baseline computation that perform WS/OS only. We analyze the effect of the batch size on the proposed method's performance on VGG16 for a single iteration of the forward and backward pass, shown in Table IV. It provides the average number of cycles and memory accesses to the local on-chip SRAM by sweeping and evaluating different batch sizes. This shows that the savings in cycles and memory accesses are consistent and do not vary with batch size. This follows closely with Eq. (8) as memory accesses and saving are proportional to the batch size, leading to a constant improvement across different batch sizes. The effect of batch size on the general performance of the systolic array is less prominent in the convolution layers as the batch size dimension also has the convolutions windows.

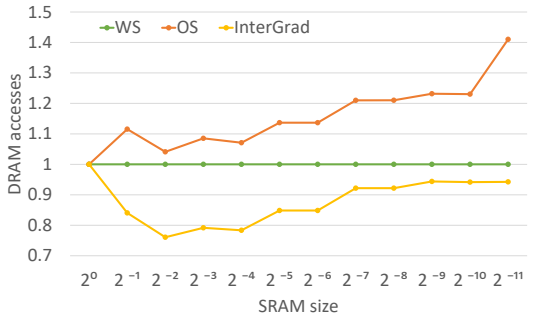

Fig. 15. Accesses to the DRAM versus the reduction in SRAM cache size. The proposed InterGrad causes fewer DRAM accesses even as the size of the SRAM is reduced. Results are normalized to WS.

TABLE IV
PERFORMANCE OF THE INTERLEAVED SCHEDULER (INTERGRAD) APPROACHES ON THE CONVOLUTIONAL LAYERS OF VGG16 WHILE VARYING BATCH SIZE. INTERGRAD CONSISTENTLY PERFORMS BETTER THAN THE BASELINE FOR ALL BATCH SIZES.

B	Cycles ($\times 10^6$)			Memory Access ($\times 10^9$)			Energy ($\times 10^{12}$)		
	WS	OS	Inter Grad	WS	OS	Inter Grad	WS	OS	Inter Grad
1	4.3	4.3	4.0	0.8	0.8	0.7	0.15	0.14	0.13
4	14.9	15.6	14.0	3.2	3.2	2.6	0.58	0.57	0.51
8	29.0	30.7	27.3	6.3	6.3	5.1	1.15	1.15	1.01
16	57.3	61.0	53.9	12.5	12.6	10.2	2.30	2.29	2.02
32	113.9	121.5	107.2	25.0	25.2	20.5	4.59	4.59	4.03
64	227.2	242.6	213.6	50.0	50.4	40.9	9.18	9.17	8.06
128	453.7	484.8	426.6	99.9	100.8	81.7	18.35	18.34	16.12

TABLE V
THE NUMBER OF CYCLES, MEMORY ACCESSES AND ENERGY FOR THE CONVOLUTIONAL LAYERS OF VGG WITH VARYING SYSTOLIC ARRAY DIMENSIONS, NORMALIZED W.R.T. WS. INTERGRAD CONSISTENTLY PERFORMS BETTER THAN THE BASELINE FOR ALL SYSTOLIC ARRAY DIMENSIONS.

Size	Cycles			Memory Accesses			Energy		
	WS	OS	InterGrad	WS	OS	InterGrad	WS	OS	InterGrad
16	1.00	1.01	0.99	1.00	1.00	0.83	1	1.00	0.88
32	1.00	1.02	0.98	1.00	1.00	0.83	1	1.00	0.88
64	1.00	1.03	0.96	1.00	1.01	0.83	1	1.00	0.88
128	1.00	1.07	0.94	1.00	1.01	0.82	1	1.00	0.88
256	1.00	1.11	0.91	1.00	1.01	0.80	1	1.00	0.87

As convolutional layers' utilization is very sensitive to the systolic array size, we evaluate the proposed method across arrays of different sizes. Table V summarizes this experiment and shows the normalized number of cycles, memory accesses, and energy consumption. The values obtained are averaged across batch sizes and normalized to WS for different array sizes. The proposed method consistently reduces the number of memory accesses by 17% and total energy by 12% when tested on the convolutional layers of the VGG network. This follows closely with the equation in Section III-C as both memory accesses and saving are directly proportional to the square of the array size, leading to a constant improvement across array sizes. The proposed method shows a more significant improvement in the number of cycles as we increase the systolic array size.

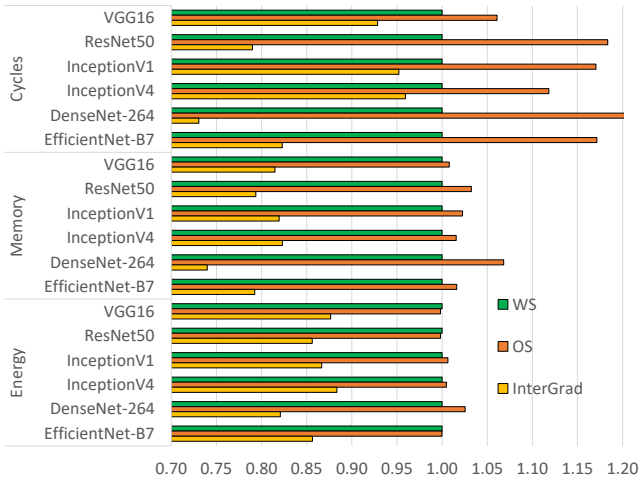

Fig. 16. Summary of performance improvement of the InterGrad mode versus other stationary modes. All results are normalized to WS.

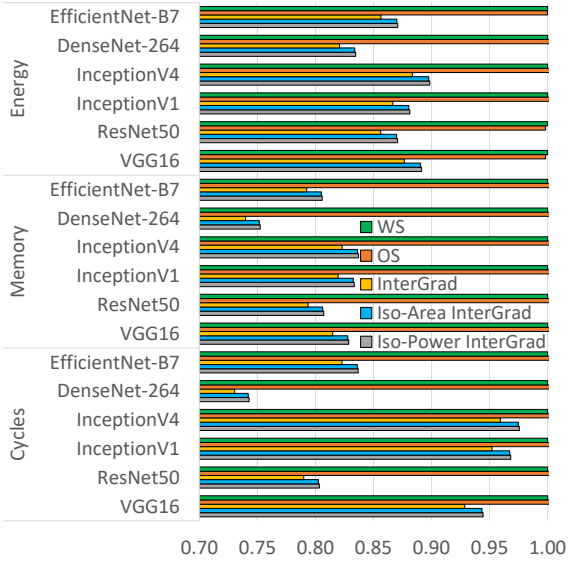

Fig. 17. Summary of iso-area and iso-power performance of InterGrad. The proposed method InterGrad, even when accounting for the area (iso-area InterGrad) and power (iso-power InterGrad) overheads, outperforms the baseline dataflows. All results are normalized to WS.

D. Impact of SRAM cache size and DRAM efficiency

All results, thus far, have assumed that the entire DNN fits within the SRAM cache. Thus, all reports for memory accesses were to the SRAM only. As modern DNN accelerators and GPUs have large caches [22], [23], the paper focused on computational comparisons without regard to cache efficiency. However, this may not be true as neural networks are getting larger. Therefore, we modified the simulator to emulate a finite-sized fully-associative cache with the least recently used policy to examine this effect. When the cache is insufficient to store the entire contents of the inputs and outputs, it needs to fetch the data from the next higher level of memory. As we tile the systolic array operations, if the data is not present, a new tile is loaded into the cache, and the least recently used tile is evicted. Thus, any cache miss results in a DRAM read/write. In Fig. 15, we evaluate the performance of the proposed architecture by starting with an SRAM cache that can just fit the entire contents of the network (2^0) and then progressively reducing the size of the SRAM cache. Fig. 15 shows the normalized DRAM accesses when the SRAM size is progressively lowered. The results reported include both the convolution and FC layers. As shown in Fig. 15, the proposed system is more robust to a reduction in cache size, even up to a $64\times$ reduction, (2^{-6}), in the SRAM size with 15% less DRAM accesses. By interleaving the backpropagation operations, the proposed method better reuses the δ variable, which would otherwise first be thrashed out due to its large size.

E. Implementation analysis

The proposed configurable PE was synthesized using a 65nm technology node operating at 100MHz and a 1V supply. The PE was parametrized so it could be synthesized with any floating-point datatype. Table VI compares the area and power characteristics of the proposed PE versus a standard WS/OS PE across common data types used for training neural networks. bf16 refers to bfloat-16, Google's brain floating-point format, and tf32 refers to Nvidia's TensorFloat-32 format. We choose a simple baseline PE that supports both WS and OS modes of operation. The baseline PE consists of a single multiplier and adder similar to the proposed PE. The baseline PE has only one memory register to store the weights during WS mode and the accumulated result during OS mode, compared to two memory registers required for the proposed PE. The baseline PE also has fewer multiplexers and a simpler control compared to the proposed PE.

TABLE VI
AREA AND POWER CHARACTERISTICS OF THE PROPOSED CONFIGURABLE PE VERSUS A SIMPLE WS/OS PE. ANALYSIS WAS PERFORMED USING DIFFERENT FLOATING POINT FORMATS FOR THE MAC BLOCK.

N	Area		Power	
	WS/OS PE	**InterGrad**	**WS/OS PE**	**InterGrad**
bf16	1.00	1.087	1.00	1.092
tf32	1.00	1.078	1.00	1.107
fp16	1.00	1.079	1.00	1.078
fp32	1.00	1.055	1.00	1.051
fp64	1.00	1.029	1.00	1.039

The proposed PE only marginally increases the area by 3-9% and power by 4-10%. The overhead depends on the ratio of the contribution by the MAC block versus the rest of the circuit. In larger data formats, the MAC unit dominates the characteristics of the block. However, the newly introduced register and control logic have a more significant impact in smaller data formats. The energy model used accounts for the increase in energy from the new registers and inter-PE communication. Furthermore, when considering all energy sources, memory reads/writes, register reads/writes, and inter-PE communication, the energy consumed by the mac operation itself is on average 30% of the total energy consumption.

To understand the implementation implications on the results, we implement the architecture described in Section IV-C and Fig. 14 in System Verilog for a 32×32 array with the BF16 format. The design is synthesized using Synopsys Design Compiler with ST-65 technology node 1V supply

and 100MHz clock. For the baseline, we choose a similar architecture to Fig. 14 that is based on implementation in [24]. The differences between the baseline and proposed approach can be described by their functionality. While the proposed architecture can support OS mode, WS mode, SGD weight update, and the new InterGrad dataflow mode, the baseline architecture only supports WS and OS modes. Flexible dataflows [15], [21], [24] that can support multiple dataflows have shown promise and the simple flexible baseline chosen has limited overhead compared to the WS-only approach. We implement and compare the baseline architecture with the proposed enhanced architecture with modified PEs and control. Additionally, there is no overhead to perform the reformulation of the input $\delta^{(l)}$ for Fig. 12, as we simply use the input version shown in Fig. 9 that is used for the baseline architecture.

Table VII shows the overhead of the proposed method over the baseline architecture. These results include all possible overheads, including the new PE design, additional data routing/ multiplexing, and new control logic.

TABLE VII
AREA AND POWER OVERHEADS OF THE PROPOSED APPROACH COMPARED TO A BASELINE SYSTOLIC ARRAY IMPLEMENTATION.

Parameter	Area		Power	
	Baseline	**InterGrad**	**Baseline**	**InterGrad**
PE array	1.00	1.079	1.00	1.089
Controller	1.00	1.035	1.00	1.041
Overall	1.00	1.016	1.00	1.017

F. System level performance of InterGrad

We evaluated the proposed method on well-known convolutional neural networks VGG16 [3], Resnet50 [4], inceptionV1 [5], Inceptionv4 [30], DenseNet-264 [31], and EfficientNet-B7 [32]. To understand the system-level performance, we show the results for the complete network, including its convolution and FC layers. It is shown in Fig. 16 that the proposed interleaved scheduler requires between 3% and 27% fewer cycles when compared to the best dataflow. The proposed method reduces the number of memory accesses between 15% and 26% while decreasing the total energy consumption between 11% and 18%. We use the implementation results from the previous subsection to generate iso-area and iso-power results. This is then used to generate the results shown in Fig. 17. The area overheads of the proposed approach do not significantly reduce the performance of the proposed approach. Even when performing an iso-area or iso-power constraints, the proposed approach achieves between 2% and 26% fewer cycles, between 16% and 25% less memory accesses, and between 10% and 17% less energy. Under iso-area comparisons, for Inception-v4, compared to WS, Intergrad achieves 12% savings in energy, 16% savings in memory and 3% savings in cycles. Savings for Densenet-264 are 17%, 25% and 26% with respect to energy, memory and cycles, respectively.

VI. RELATED WORK

Though there has been an abundance of AI/ML hardware accelerators however, very few works target training [33]. Most DNN training accelerators optimize matrix-matrix multiplication, whether it be dense matrices [12], [34] or sparse matrices [6]–[8], [21], [35], [36]. Existing training optimizations exploit the sparse nature of the computations to reduce complexity and communication overhead [37], [38]. Several papers also consider flexible dataflows or interconnect to handle variable size matrices [7], [15], [21]. Additionally, heterogeneous clusters of processors [24] optimized for different computations in the CNN have been used. Dataflows like the row stationary, though shown to work exceptionally well with inference, have challenges in training. In the case of training, the filter "row" used is the size of the feature matrix rows, not just filter sizes. This exhibits a significant degree of variation in the neural network depending on the feature map size at each layer, requiring a large scratch pad which is hard to keep fully utilized. Sparse training accelerators [39] can significantly reduce the number of computations by exploiting the zeros and dynamic pruning. However, the above approaches target computations in a generic sense and are orthogonal to the architectural exploitation of the backpropagation algorithm proposed in this paper.

Although there has been an abundance of inference-based convolution accelerators, these do not translate well to the training task. Quantization or approximation-based approaches [40]–[42] that use reduced precision arithmetic, assume that training and retraining are possible in higher precision. Accelerators targeted towards convolution [43], [44] are optimized for a range of input filter and feature map sizes. These assumptions of sizes do not hold true when performing the backward pass and are sub-optimal.

Many algorithmic or co-design implementations of the backpropagation algorithm have focused on ways to speed up the backpropagation algorithm. Pipeline parallelism, where the backpropagation is split into multiple processors, is one such optimization [45]–[48]. However, unlike the proposed method, the above approaches are primarily based on partitioning and multiprocessing rather than on-device optimizations. A prior approach [49] to eliminate the inefficiency of the im2col, with inference in mind, provides an alternative solution to frame the im2col as multiple 1×1 convolutions. However, the proposed method is optimized for training by designing a solution that eliminates the im2col operation and exploits interleaving. Also, the reuse of variables between layers and the different operations in the backpropagation algorithm has been proposed recently to maximize memory efficiency [50]–[52]. However, these approaches exploit memory reuse within the nearest memory block (cache/SRAM). The proposed work is the first to reuse variables at the hardware level within the computations of the processor.

VII. CONCLUSION

This paper proposes a novel scheduling scheme by interleaving various computations to reduce the latency and memory accesses of the design. This is the first approach to exploit multi-operation variable reuse within the matrix-multiplication block without additional caches or scratchpads. It has been shown that the proposed method outperforms the

best traditional dataflow schemes by a factor of $1.4\times \sim 2.2\times$ in terms of the number of cycles and by a factor of up to $1.9\times$ in terms of memory accesses in fully-connected layers found in common CNNs. Furthermore, the proposed method uses up to 25% fewer cycles and memory accesses, and 16% less energy than baseline implementation for state-of-the-art CNNs. Future work will be directed towards three avenues. First, we will explore adapting this work to consider the effect of structured sparsity. Second, we will extend this work for training recurrent neural networks (RNNs) via backpropagation through time (BPTT), and graph neural networks. Third, with the rising cost of transformers [53], the proposed method might lead to significant improvements in the fully-connected layers within the transformer.

REFERENCES

[1] K. K. Parhi and N. K. Unnikrishnan, "Brain-inspired computing: Models and architectures," *IEEE Open Journal of Circuits and Systems*, vol. 1, pp. 185–204, 2020.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, 2012.

[3] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

[4] K. He *et al.*, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[5] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[6] C. Deng *et al.*, "PermDNN: Efficient compressed DNN architecture with permuted diagonal matrices," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[7] Y. Chen *et al.*, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, June 2019.

[8] S. Han *et al.*, "EIE: Efficient inference engine on compressed deep neural network," in *isca*, June 2016, pp. 243–254.

[9] J. Devlin *et al.*, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 2019.

[10] N. Unnikrishnan and K. K. Parhi, "A gradient-interleaved scheduler for energy-efficient backpropagation for training neural networks," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, 2020.

[11] E. Strubell, A. Ganesh, and A. McCallum, "Energy and policy considerations for deep learning in NLP," in *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2019.

[12] Y. Chen *et al.*, "DaDianNao: A machine-learning supercomputer," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014, pp. 609–622.

[13] B. Fleischer *et al.*, "A scalable multi- TeraOPS deep learning processor core for AI training and inference," in *2018 IEEE Symposium on VLSI Circuits*, 2018.

[14] A. Samajdar *et al.*, "SCALE-Sim: Systolic CNN accelerator simulator," *arXiv preprint arXiv:1811.02883*, 2018.

[15] H. Kwon, A. Samajdar, and T. Krishna, "MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[16] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*. Hoboken, NJ, USA: Wiley, 1999.

[17] K. K. Parhi, "Hierarchical folding and synthesis of iterative data flow graphs," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 60, no. 9, pp. 597–601, Sep. 2013.

[18] K. Ito and K. K. Parhi, "Determining the minimum iteration period of an algorithm," *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, vol. 11, no. 3, pp. 229–244, Dec 1995.

[19] H. T. Kung and C. E. Leiserson, "Systolic arrays (for VLSI)," in *Sparse Matrix Proceedings*, vol. 1, 1979, pp. 256–282.

[20] V. Sze *et al.*, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec 2017.

[21] E. Qin *et al.*, "SIGMA: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 58–70.

[22] NVIDIA, "NVIDIA A100 Tensor Core GPU Architecture," Tech. Rep., 2020.

[23] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2017.

[24] H. Genc *et al.*, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *Proceedings of the Design Automation Conference (DAC)*, 2021.

[25] J. Cong *et al.*, "Latte: Locality aware transformation for high-level synthesis," in *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2018, pp. 125–128.

[26] Y.-H. Lai *et al.*, "HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing," in *Proceedings of the ACM International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019.

[27] C.-Y. Wang and K. K. Parhi, "High-level DSP synthesis using concurrent transformations, scheduling, and allocation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 3, pp. 274–295, March 1995.

[28] N. K. Unnikrishnan and K. K. Parhi, "Multi-channel FFT architectures designed via folding and interleaving," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, 2022, pp. 142–146.

[29] Y. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 367–379.

[30] C. Szegedy *et al.*, "Inception-v4, Inception-ResNet and the impact of residual connections on learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2017.

[31] G. Huang *et al.*, "Convolutional networks with dense connectivity," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2019.

[32] M. Tan and Q. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *Proceedings of the International Conference on Machine Learning*, Jun. 2019, pp. 6105–6114.

[33] Y. Chen *et al.*, "A survey of accelerator architectures for deep neural networks," *Engineering*, vol. 6, no. 3, pp. 264–274, 2020.

[34] L. Song *et al.*, "PipeLayer: A pipelined ReRAM-based accelerator for deep learning," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 541–552.

[35] N. Kim *et al.*, "ComPreEND: Computation pruning through predictive early negative detection for ReLU in a deep neural network accelerator," *IEEE Transactions on Computers*, pp. 1–1, 2021.

[36] J. Yang *et al.*, "S2Engine: A novel systolic architecture for sparse convolutional neural networks," *IEEE Transactions on Computers*, pp. 1–1, 2021.

[37] Y. Yu and N. K. Jha, "SPRING: A sparsity-aware reduced-precision monolithic 3D CNN accelerator architecture for training and inference," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–1, 2020.

[38] H. Wang *et al.*, "Error-compensated sparsification for communication-efficient decentralized training in edge environment," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 1, pp. 14–25, 2022.

[39] S. Kim *et al.*, "TSUNAMI: Triple sparsity-aware ultra energy-efficient neural network training accelerator with multi-modal iterative pruning," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 69, no. 4, pp. 1494–1506, 2022.

[40] Z.-G. Tasoulas *et al.*, "Weight-oriented approximation for energy-efficient neural network inference accelerators," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 12, pp. 4670–4683, 2020.

[41] W. Liu, J. Lin, and Z. Wang, "A precision-scalable energy-efficient convolutional neural network accelerator," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 10, pp. 3484–3497, 2020.

[42] L. R. Juracy *et al.*, "A high-level modeling framework for estimating hardware metrics of cnn accelerators," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 11, pp. 4783–4795, 2021.

[43] J. Jo, S. Kim, and I.-C. Park, "Energy-efficient convolution architecture based on rescheduled dataflow," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 12, pp. 4196–4207, 2018.

[44] K.-W. Chang and T.-S. Chang, "VWA: Hardware efficient vectorwise accelerator for convolutional neural network," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 1, pp. 145–154, 2020.

[45] N. K. Unnikrishnan and K. K. Parhi, "LayerPipe: Accelerating deep neural network training by intra-layer and inter-layer gradient pipelining and multiprocessor scheduling," in *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, 2021.

[46] D. Narayanan *et al.*, "PipeDream: Generalized pipeline parallelism for DNN training," in *Proceedings of the ACM Symposium on Operating Systems Principles*, 2019.

[47] Y. Huang *et al.*, "GPipe: Efficient training of giant neural networks using pipeline parallelism," in *Advances in Neural Information Processing Systems*, 2019.

[48] S. Zhao *et al.*, "vPipe: A virtualized acceleration system for achieving efficient and scalable pipeline parallel DNN training," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 3, pp. 489–506, 2022.

[49] A. Vasudevan, A. Anderson, and D. Gregg, "Parallel multi channel convolution using general matrix multiplication," in *Proceedings of the IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2017, pp. 19–24.

[50] S. Venkataramani *et al.*, "Memory and interconnect optimizations for peta-scale deep learning systems," in *Proceedings of the IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2019, pp. 225–234.

[51] S. Lym *et al.*, "Mini-batch serialization: CNN training with inter-layer data reuse," in *Proceedings of Machine Learning and Systems*, 2019.

[52] H. Jin *et al.*, "Layer-centric memory reuse and data migration for extreme-scale deep learning on many-core architectures," *ACM Transactions on Architecture and Code Optimization*, vol. 15, no. 3, Sep. 2018.

[53] O. Sharir, B. Peleg, and Y. Shoham, "The cost of training NLP models: A concise overview," 2020. [Online]. Available: https://arxiv.org/abs/2004.08900

Nanda K. Unnikrishnan (Graduate student member, IEEE) is currently pursuing a Ph.D. degree in electrical engineering at the University of Minnesota, Minneapolis, USA. He received his B.Tech in electronics and communication from the National Institute of Technology, Karnataka (NITK), Suratkal, India in 2014, and his M.S. in electrical engineering from the University of Minnesota in 2018.

He worked at SilabTech, India (now Synopsys) from 2014 to 2016 as a design verification engineer for mixed-signal designs. He interned at Qualcomm Technologies Inc, San Diego in the Summer of 2017, with Intel Labs in the Summer of 2018, and Facebook in the Summer of 2021. He currently works as an ASIC architecture engineer at Meta. His research interests lie in the design of VLSI architectures for machine learning and deep learning systems.

Keshab K. Parhi (S'85–M'88–SM'91–F'96) received the B.Tech. degree from the Indian Institute of Technolgy, Kharagpur, India, in 1982; the M.S.E.E. degree from the University of Pennsylvania, Philadelphia, PA, USA; in 1984, and the Ph.D. degree from the University of California at Berkeley, Berkeley, CA, USA, in 1988. He has been with the University of Minnesota, Minneapolis, MN, USA, since 1988, where he currently is the Erwin A. Kelen Chair and Distinguished McKnight University Professor in the Department of Electrical and Computer Engineering. He has published over 700 papers, is the inventor or co-inventor of 34 patents, has authored the textbook VLSI Digital Signal Processing Systems (New York, NY, USA: Wiley, 1999), and coedited the reference book Digital Signal Processing for Multimedia Systems (Boca Raton, FL, USA: CRC Press, 1999). His current research interests include the VLSI architecture design and implementation of artificial intelligence and machine learning systems, data-driven computational neuroscience and psychiatry, and hardware security. Dr. Parhi has served on the Editorial Boards of the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS PART I AND PART II, the IEEE TRANSACTIONS ON VLSI SYSTEMS, IEEE TRANSACTIONS ON SIGNAL PROCESSING, the IEEE SIGNAL PROCESSING LETTERS, and the IEEE Signal Processing Magazine, and served as the Editor-in-Chief of the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS PART I from 2004 to 2005. He currently serves on the Editorial Board of the Journal of Signal Processing Systems (Springer). He was the Distinguished Lecturer of the IEEE Circuits and Systems Society during 1996-1998 and 2019-2021. He served as a Board of Governors Elected Member of the IEEE Circuits and Systems Society from 2005 to 2007. He is the recipient of numerous awards including the 2017 Mac Van Valkenburg award, the 2012 Charles A. Desoer Technical Achievement award, the 2021 John Choma Education Award and the 1999 Golden Jubilee medal, from the IEEE Circuits and Systems Society, the 2013 Distinguished Alumnus Award from IIT Kharagpur (India), the 2013 Graduate/Professional Teaching Award from the University of Minnesota, the 2004 F. E. Terman award from the American Society of Engineering Education, the 2003 IEEE Kiyo Tomiyasu Technical Field Award, and the 2001 IEEE W. R. G. Baker Prize Paper Award. He is a Fellow of ACM, AIMBE, AAAS and NAI.

Supplementary Information: InterGrad: Energy-Efficient Training of Convolutional Neural Networks via Interleaved Gradient Scheduling

Nanda K. Unnikrishnan, *Graduate Student Member, IEEE*; and Keshab K. Parhi, *Fellow, IEEE*

S1. Dataflow representation of the backpropagation algorithm

Fig. S1 shows a dataflow representation of backpropagation in the fully connected layer. The figure comprises two parts: the forward pass (top) and the backward pass (bottom). Each node of the dataflow graph in the forward pass is accompanied by the corresponding equation that shows the computation at that node. W^l is a $N^l \times N^{l-1}$ matrix representing the layer weights, and N^l represents the number of output neurons. The dimensions at the output of layer l. z^l and a^l, are the outputs of the fully connected layer and activation functions, respectively, each of size $N^l \times B$, where B represents the batch size of the input data. Each node in the forward pass also highlights the dimensions of the output of that node. In the backward pass, we illustrate the gradients that propagate through each edge of the dataflow graph with a colored dashed line. First, the green lines highlight the gradient of the error w.r.t. the output of a layer ($\frac{\partial E}{\partial z}$) or δ, calculated by Eq. (3) in the paper. Second, the red lines highlight the gradient of the error w.r.t. the output of the activation function of a layer ($\frac{\partial E}{\partial a}$), calculated by Eq. (4) in the paper. Last, the blue lines highlight the gradient of the error w.r.t. the weight of a layer ($\frac{\partial E}{\partial W}$) or G, calculated by Eq. (5) in the paper. The backward pass also highlights the dimensions of all the matrices.

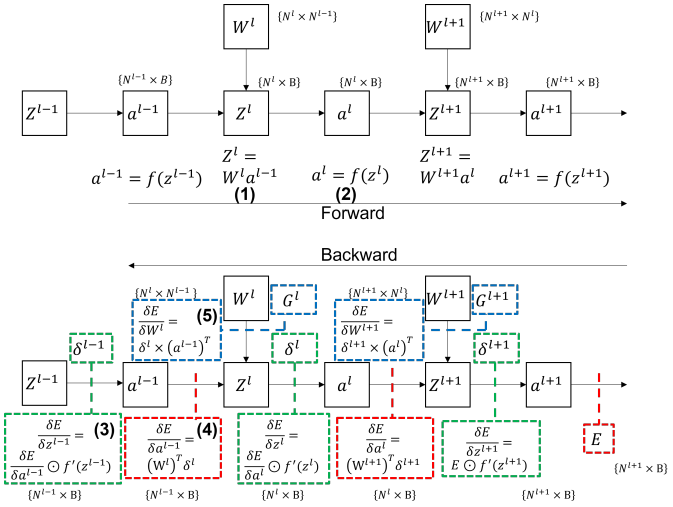

Fig. S1. Equations and derivatives at every stage of a sample dataflow of a 3-layer neural network. The dimensions of the variables are also included.

A. Calculating the derivatives of activation

The computations of the activation function, its derivative, and the associated Hadamard product are often designated to special blocks or special function units (SFUs). In the forward pass, these functions are computed with the help of look-up tables to approximate the different types of activation functions. However, in the backward pass, these can exploit some of the characteristics of the derivatives to minimize computations. We illustrate this below on a set of popular activation functions: Sigmoid, hyperbolic tangent (tanh), rectified linear unit (ReLU), Leaky Rectified linear unit (LReLU), exponential linear unit (ELU), and scaled exponential linear unit (SELU).

$$f(x) = Sigmoid(x) = \sigma(x) = \frac{1}{1+e-x} \tag{S1}$$

$$f'(x) = \frac{\partial(\sigma(x))}{\partial x} = f(x)(1-f(x)) \tag{S2}$$

$$f(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e-x} \tag{S3}$$

$$f'(x) = \frac{\partial(tanh(x))}{\partial x} = (1-f^2(x)) \tag{S4}$$

$$f(x) = ReLU(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \tag{S5}$$

$$f'(x) = \frac{\partial(ReLU(x))}{\partial x} = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \tag{S6}$$

$$f(x) = LReLU(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases} \tag{S7}$$

$$f'(x) = \frac{\partial(LReLU(x))}{\partial x} = \begin{cases} 1 & \text{if } x > 0 \\ \alpha & \text{if } x \leq 0 \end{cases} \tag{S8}$$

$$f(x) = ELU(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases} \tag{S9}$$

$$f'(x) = \frac{\partial(ELU(x))}{\partial x} = \begin{cases} 1 & \text{if } x > 0 \\ f(x) + \alpha & \text{if } x \leq 0 \end{cases} \tag{S10}$$

K. K Parhi and N. K. Unnikrishnan are with the department of Electrical and Computer engineering at the University of Minnesota, Minneapolis, MN 55455 USA (e-mail: unnik005@umn.edu, parhi@umn.edu).

This work has been supported in part by the National Science Foundation under Grants CCF-1914749.

$$f(x) = SELU(x) = \begin{cases} \lambda x & \text{if } x > 0 \\ \lambda\alpha(e^x - 1) & \text{if } x \leq 0 \end{cases} \tag{S11}$$

$$f'(x) = \frac{\partial(SELU(x))}{\partial x} = \begin{cases} 1 & \text{if } x > 0 \\ f(x) + \lambda\alpha & \text{if } x \leq 0 \end{cases} \tag{S12}$$

There are two general ways to compute the activation function for the forward pass. First, simple activations like ReLU can be implemented with multiplexers without additional computation. Second, complex activations using exponent functions can be approximated with look-up table-based solutions. We can implement some of the characteristics in the backward pass and generate the derivatives from the existing outputs of the forward pass. Simple derivatives like the ReLU and LReLU can be implemented with multiplexers. Derivatives of activation functions like ELU and SELU can use the existing outputs and vector addition to add a constant to all values. Functions like sigmoid and tanh can be similarly derived with vector addition and multiplication blocks. Thus an SFU with a forward and backward pass can be implemented with a vector version of look-up tables, multiplexers, adders, and multipliers. This vector multiplier can also be used to implement the Hadamard product required in backpropagation equations.

Though the SFU handles many operations, SFUs are only a tiny percentage of the overall number of computations in a large DNN. These SFU functions make up less than 0.01% (calculated from VGG16) of the total number of multiply and accumulate (MAC) operations required to train a CNN. This can be understood by looking at the computation of a single convolution layer. A convolution filter performs $F * F * C$ MAC operations for each call to the activation function, where $F * F$ is the filter size, and C is the number of input feature maps. For the deeper layers of CNNs like ResNet and VGG, this can be of the order of $3 * 3 * 512$, which is a factor of 4608. So over the entire span of deep networks, the number of activation MACs becomes insignificant compared to the main convolution computations. In a backward pass, for each evaluation of the derivative of the activation function, other operations include computing gradient with respect to the weights, gradient with respect to the activation, and a weight update step. Even when accounting for multiple operations in the backward pass like vector multiplications additions and Hadamard products, the derivative evaluation and Hadamard product operations make up less than 0.01% of the backward pass (calculated for the VGG16). Hence the SFU forms a very tiny percentage of the DNN computations.

B. Handling different weight update optimizers

One advantage of the gradient interleaving approach is that it ensures that the gradient and its associated weight exist within the same PE. This is advantageous as it allows us to perform an in-place memory update without needing to unload the gradients or reload the weights for a weight update operation. This was explained in the context of an SGD operation. However, other optimizers bring in new challenges. The

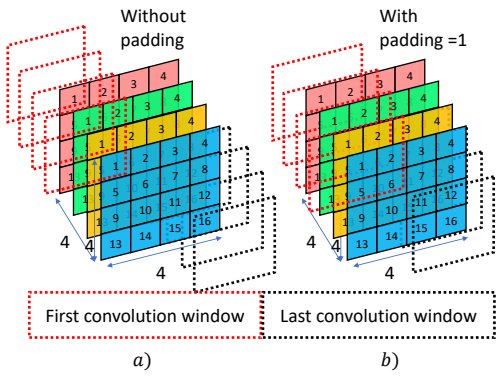

Fig. S2. Effect of padding inputs in the forward pass on $\delta^{(l-1)}$ computations. a) $\delta^{(l-1)}$ computation without padding. b) $\delta^{(l-1)}$ computation with padding of 1.

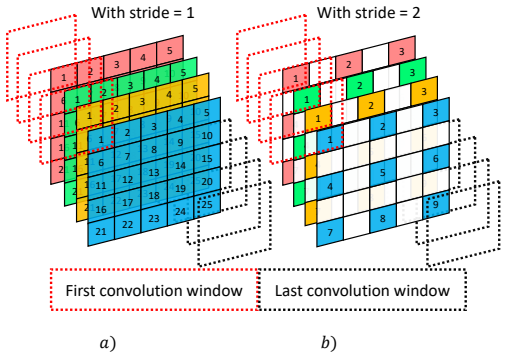

Fig. S3. Effect of stride the inputs in the forward pass on $\delta^{(l-1)}$ computations. A stride (S) in the forward pass can be mapped to a sparse output filter map with $S-1$ null operations inserted between each data point. a) $\delta^{(l-1)}$ computation with stride=1. b) $\delta^{(l-1)}$ computation with stride=2.

simplest extension is the introduction of momentum, either in the standard form or as Nesterov accelerated gradients.

$$\boldsymbol{m}(t) = \gamma\boldsymbol{m}(t-1) + \eta\nabla L(\boldsymbol{W}(t-1)) \tag{S13}$$

$$\boldsymbol{W}(t) = \boldsymbol{W}(t-1) - \boldsymbol{m}(t) \tag{S14}$$

$$\boldsymbol{m}(t) = \gamma\boldsymbol{m}(t-1) + \eta\nabla L(\boldsymbol{W}(t-1) - \gamma\boldsymbol{m}(t-1)) \tag{S15}$$

$$\boldsymbol{W}(t) = \boldsymbol{W}(t-1) - \boldsymbol{m}(t) \tag{S16}$$

Eqs. (S13) and (S14) represent the update equations for momentum SGD and Eqs. (S15) and (S16) represent the update equations for Nestrov accelerated gradients. Here $\boldsymbol{W}(t)$ represents the weight matrix, $\boldsymbol{m}(t)$ the momentum of the parameter, t the unit step interval between weight updates, η the learning rate, γ a tunable hyperparameter and $\nabla L()$ the gradient of the loss function with respect to the weight used.

The introduction of momentum requires an additional register in the PE to store the parameters. With respect to the system's energy, the number of MAC operations remains the same. At the same time, the proposed method continues to save memory access by reading and writing the gradients and the weights, thus saving energy. The only additional energy of the system will be the inter-PE communication to load and unload

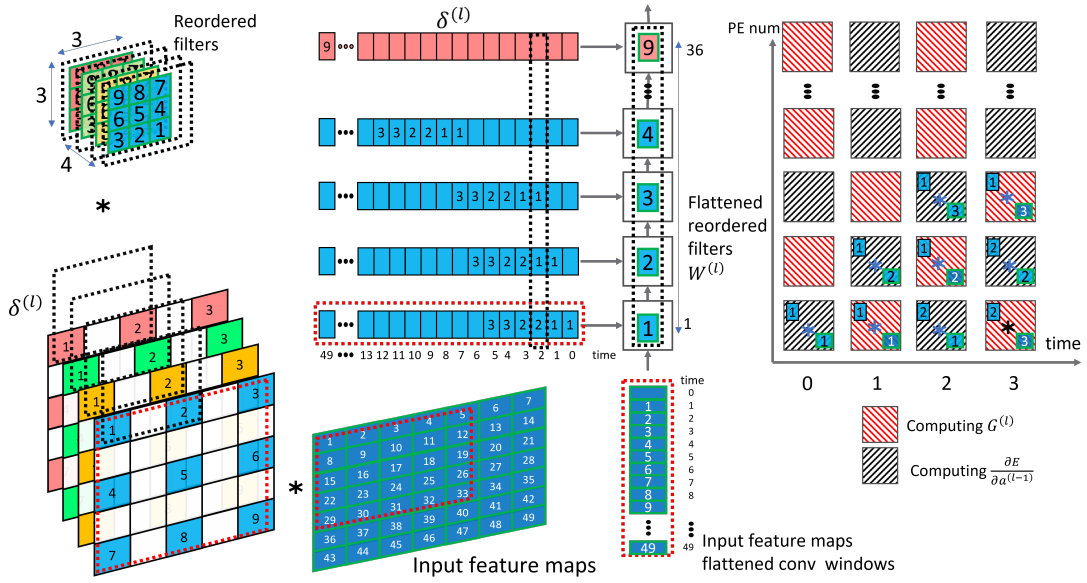

Fig. S4. Modified interleaved scheduling to handle stride. The null operations introduced in $\delta^{(l-1)}$ calculation are reused for computing G by tactically broadcasting the input as shown.

the momentum parameters. This is an acceptable increase as inter-PE communication is limited to only the start and end of the computations. Additionally, the inter-PE communication is proportional to the size of the systolic array. Thus, a smaller array size will see a lesser impact than a larger array. The evaluation section explores this further to analyze the impact on performance if the weight update step is skipped.

S2. Handling Stride and Padding

The introduction of convolution layers requires special considerations for striding and padding. Padding defines how many extra padded elements are added to each side of the input feature map. Padding of one indicates that the dimensions of the input feature map are expanded by 1 in each direction. The expanded dimensions are filled in with 0 or a user-defined value. When considering the backward pass, the changes to the computations for $\delta^{(l-1)}$ are shown in Fig. S2. The introduction of padding changes the convolution window's starting and end locations as we are not required to compute the gradients to the padding locations. For the interleaved gradients case, the algorithm does not need to compute $\delta^{(l-1)}$ in the cycles corresponding to the padding values. The G computation with padding is the same, except that the input feature map is padded.

Striding defines at what interval to perform the convolution operation. A simple convolution operation with a stride of 1 moves the convolution window by one pixel after each computation. A stride of S indicates that the convolution window is moved by S pixels to perform the new computation. Thus performing a stride greater than one in the forward pass results in fewer outputs. The effect of striding on the backward $\delta^{(l-1)}$ computation is shown in Fig. S3. Conceptually, the output feature space can be visualized as a sparse representation with $S-1$ null operations between the data points. Thus, we use this sparse data representation for the $\delta^{(l-1)}$ and $G^{(l)}$ computations.

A direct way to extend gradient interleaving to cases with stride greater than one is to treat these null operations as 0s as shown in Fig. S3 and map it to the existing approach. However, we can leverage these null operations to perform interleaving. This is demonstrated in the simple example shown in Fig. S3. This section will only explore how to modify InterGrad to support striding and the basics of gradient interleaving for convolutions have been covered in Section 3.3. The black dotted lines represent the convolution window required to compute $\delta^{(l-1)}$ from the reordered filters and $\delta^{(l)}$ maps shown on the left. The corresponding mapping to the systolic array is shown by the black dotted lines over the array and the $\delta^{(l)}$ input to the array. The computation for $G^{(l)}$ is shown by the red dotted lines on the $\delta^{(l)}$ and input feature maps at the bottom. The mapping of this computation to the systolic array is shown by red dotted boxes on the inputs of the array.

We develop a static timing chart to indicate which cycles the corresponding PE is computing $G^{(l)}$ or $\frac{\partial E}{\partial a^{(l-1)}}$, as shown on the right of Fig. S4. The contents in each box indicate the multiplication at that time instance. For example, in time instance 3, PEs 1 and 3 are computing $\frac{\partial E}{\partial a^{(l-1)}}$ for the convolution shown in the black dotted box and PE 2 is computing $G^{(l)}$ for the convolution shown by the red dotted box. Thus with selective broadcasting and bypassing of inputs we can effectively interleave the two operations without the need to double the number of cycles. The proposed method is advantageous compared to the traditional im2col approach as we do not need to compute each stridden convolution window for the input feature map.

S3. Limitations of Interleaved Scheduling

A limitation of the proposed method is that it could potentially lead to underutilization of the systolic array. This

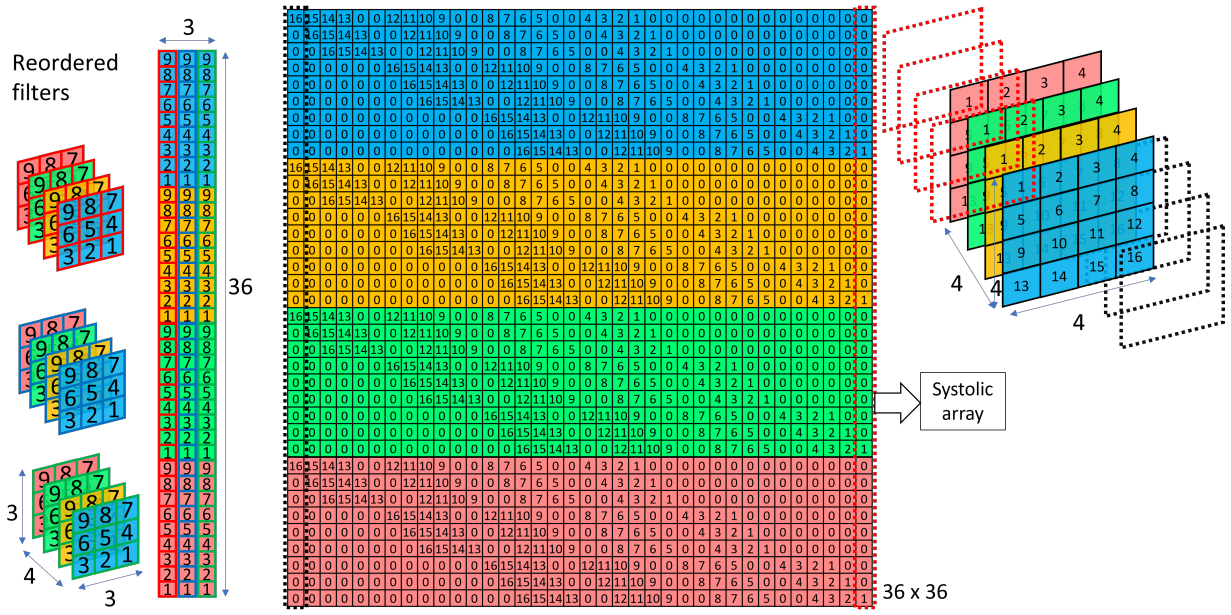

Fig. S5. Input matrices for the weight stationary calculation of $\delta^{(l-1)}$ in Fig. 9. The 3×36 matrix on the left represents the re-ordered flattened reordered filters stored in the systolic array. The 36×36 matrix in the center represents the $\delta^{(l)}$ input to the systolic array. The red and black dotted boxes show how the convolution window of the $\delta^{(l)}$ map shown on the right is mapped to the matrix.

can be understood by looking at Eqs. (4) and (5). When we apply the convolution operation described in Section III-C, the base application uses the systolic array in a weight-stationary manner for δ calculation. The convolution between the filter and $\delta^{(l)}$, to calculate $\delta^{(l-1)}$, is inherently parallel along the input channel dimension as illustrated in Fig. 7. However, the convolution between the input feature map and $\delta^{(l)}$, to calculate G, is computed in parallel in the output channel dimension.

The issue lies in the cases where the input dimensions and output dimensions are such that they are significantly less than the dimensions of the systolic array. For example, a 128×128 systolic array with 16 input channels and 32 filters will be utilized 12.5% if the dataflow is parallel in the input channels dimension and 25% if the dataflow is parallel in the output filters dimension. When interleaved scheduling changes the calculation of G from output parallel to input parallel, this could lead to better utilization if the number of input channels is greater than the number of output channels or *vice versa*. This is seen in the case of the inception past the dimensionality reduction layers. This underutilization is less prominent in networks like VGG and Resnet, where the input and output layer dimensions are significantly larger than the systolic array size. The filter size, F, plays a small role in determining utilization. This is shown in Fig. S7 in the mapping on the top right corner, where F is never mapped to a dimension of the array independently but always combined with another variable. The characterization is less dependent on the input feature map size as the input image is broken into convolution windows, and the number of convolution windows is combined with either the input channels or output channels. In all the cases explored, the main bottleneck for the array is when the number of input or output channels is small. Fig. S7 illustrates the underutilization of the different dataflows for the first five layers of the VGG16 network. The top right table in Fig. S7 indicates the number of operations mapped to the X and Y dimensions of the array and the number of cycles (N) to compute one array result. The table highlights which mappings would lead to underutilization for different systolic array sizes. For example, the table entries highlighted in grey would lead to underutilization in all array sizes greater than and equal to 128×128.

This underutilization leads to lower throughput in the InterGrad case, as shown in Fig. S7 for layers 1 and 3. Despite this drawback, InterGrad has a considerably lower memory footprint in all cases. This leads to InterGrad outperforming all other dataflows in terms of energy consumption.

S4. Characterization of the Weight Update Step

Implementing the weight update step in hardware may not be practical in the context of interleaved gradient scheduling for complex optimizers or operation tiling. Thus, it is essential to characterize the energy-saving from the weight update step compared to the savings from reusing δ.

TABLE S1
ENERGY SAVINGS FROM THE WEIGHT UPDATE STEP AS A FRACTION OF THE OVERALL SAVINGS WITH VARYING SYSTOLIC ARRAY SIZES.

Size	VGG16	ResNet50	InceptionV1
32	1.39%	0.74%	0.35%
64	2.67%	1.35%	0.61%
128	5.02%	2.52%	1.11%

Table S1 summarizes the percentage energy savings of the SGD weight update step compared to the overall energy savings of the InterGrad ($\frac{Savings\ from\ weight\ update}{Overall\ InterGrad\ savings}$). The results were simulated by varying the systolic array size for

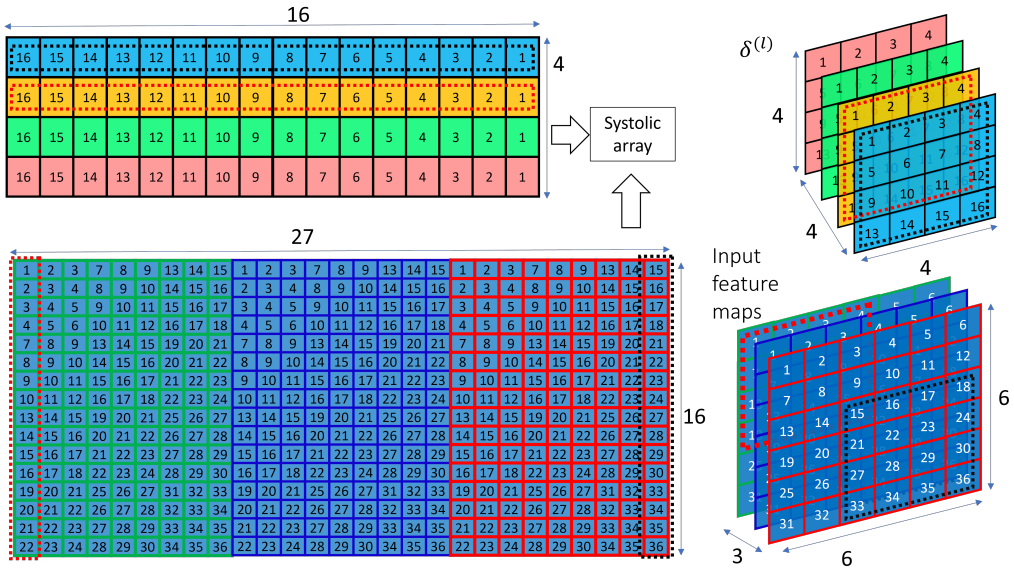

Fig. S6. Input matrices for the output stationary calculation of $G^{(l)}$ in Fig. 10. The 16×4 matrix at the top shows how the $\delta^{(l)}$ map on the top right is mapped to systolic array input. The 27×16 matrix at the bottom shows how the input features maps at the bottom right is mapped to systolic array input. The red and black dotted boxes show how the convolution windows shown on the right is mapped to the matrix.

VGG	Input maps size/ $\delta^{(l-1)}$ size (A)	Input channels (IC)	Filter size (F)	Output channels (OC)	Output maps size/ δ^{l} size (B)	Mode inGrad (IG)	Delta calculation $\delta^{(l-1)}=W^{l}*\delta^{l}$			G calculation $G^{(l-1)}=IF*\delta^{l}$			Comparison		
							X	Y	N	X	Y	N	Cycles	Mem	Energy
1	224x224	3	3x3	64	224x224	WS	3	576	50176	50176	64	27	1.00	1.00	1.00
						OS	3	50176	576	27	64	50176	1.17	1.00	0.86
						IG	3	576	50176	3	576	50176	1.26	0.89	0.63
2	224x224	64	3x3	64	224x224	WS	64	576	50176	50176	64	576	1.00	1.00	1.00
						OS	64	50176	576	576	64	50176	1.12	1.01	1.00
						IG	64	576	50176	64	576	50176	0.88	0.78	0.86
3	112x112	64	3x3	128	112x112	WS	64	1152	12544	12544	128	576	1.00	1.00	1.00
						OS	64	12544	1152	576	128	12544	1.12	1.01	0.99
						IG	64	1152	12544	64	1152	12544	1.07	0.83	0.87
4	112x112	128	3x3	128	112x112	WS	128	1152	12544	12544	128	1152	1.00	1.00	1.00
						OS	128	12544	1152	1152	128	12544	1.10	1.02	1.00
						IG	128	1152	12544	128	1152	12544	0.90	0.82	0.88
5	56x56	128	3x3	256	56x56	WS	128	2304	3136	3136	256	1152	1.00	1.00	1.00
						OS	128	3136	2304	1152	256	3136	1.05	1.01	1.00
						IG	128	2304	3136	128	2304	3136	0.90	0.82	0.88

		WS	OS	IS
δ	X	IC	IC	IC
	Y	F*OC	A	F*OC
	N	A	F*OC	A
	P	IC*OC*F	IC*A	IC*OC*F
G	X	B	F*IC	IC
	Y	OC	OC	F*OC
	N	F*IC	B	A
	P	B*OC	F*IC*OC	F*IC*OC

legend	Under utilization array size
	32
	64
	128
	256

Fig. S7. Analysis of utilization after mapping the first five layers of the VGG16 network. X and Y refer to the dimensions of the computation that need to be mapped to the systolic array. The column headers define the dimensions of the variables. The table on the top right shows how the computation dimensions are mapped to the systolic array. The underutilization array size table defines at what array size the mapping is underutilized ($\leq 80\%$ utilization). Underutilization in smaller arrays always leads to underutilization in larger arrays. The comparison column compares the different approaches normalized to WS. WS is Weight-stationary, OS is Output-stationary, and IG is the proposed interleaved scheduling (InterGrad).

multiple popular CNNs. As shown in Table S1, the savings from the weight update step are only a small fraction of the overall savings. There are two main contributing factors. First, the weight update step has a more prominent role in the FC layer than the convolution layers. Thus, networks where the FC layer plays a more significant role, like VGG16, have a more considerable weight update step. Second, with the increase in array size, there is less inefficiency in communication and fewer memory accesses. Thus, the contribution from the reuse of δ reduces, but it still accounts for most of the energy savings. Thus a detailed tradeoff analysis is required to evaluate supporting complex optimizers within the PE-array. The weight update step could be offloaded to the main CPU, or a dedicated hardware unit within the SFU. This should be explored in future work.