# A Gradient-Interleaved Scheduler for Energy-Efficient Backpropagation for Training Neural Networks

Nanda Unnikrishnan
*University of Minnesota*
Minneapolis, USA
unnik005@umn.edu

Keshab K. Parhi
*University of Minnesota*
Minneapolis, USA
parhi@umn.edu

***Abstract*—This paper addresses design of accelerators using systolic architectures for training of neural networks using a novel *gradient interleaving* approach. Training the neural network involves backpropagation of error and computation of gradients 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 to the same *configurable* systolic array. This results in reuse of the variables from one computation to the other and eliminates unnecessary memory accesses. The proposed approach leads to $1.4-2.2\times$ savings in terms of number of cycles and $1.9\times$ savings in terms of memory accesses. Thus, the proposed accelerator reduces latency and energy consumption.**

***Index Terms*—Neural Network, Deep learning, Accelerator architectures, Processor scheduling, Gradient interleaving, Systolic array**

## I. Introduction

Deep neural networks (DNNs) have permeated into all facets of our daily lives as seen in advances for recommender systems, automated photo recognition, and automatic text generations. Inference, in particular, has been extensively investigated because of its inherent advantages in data privacy, response time and bandwidth demand over cloud-based inference. Deep learning networks such as [1]–[5] have led to a massive surge in data center workloads. Fully connected layers in particular consume over 90% of the inference workloads currently in Google's datacenters [6].

The common understanding in the community is that machine learning algorithms are efficient because they have a large one time cost and subsequent inference cost is minimal. This mindset needs to be re-evaluated as we may be severely underestimating the cost of training. There have been numerous architectures that have reported energy consumption for inference [7]–[9] but very few for training [10], [11]. The energy consumption for training can be extrapolated from some energy-efficient architectures [8] as $54kJ$ for a single epoch considering training a simple network like Alexnet of Imagenet challenge. Thus we can see that even training a small neural network can consume significant energy.

Specifically, it has been shown that compared to the cost of execution, the cost of memory accesses dominates the energy consumption of these accelerators [12]. This has led to the development of memory-centric schedules that try to minimize the memory bandwidth to the DRAM. This can be achieved by treating it as a caching problem, selecting appropriate block sizes and blocks to maximize the reuse of the SRAM contents [13]. The majority of modern schedulers process the neural network in a layerwise manner and each layer is sequentially scheduled. Further approaches have tried to formulate it as an optimization problem/heuristics by careful partitioning [14]. Here, however, optimizations are only limited to intralayer optimizations, which may not exploit the benefits of advantages between layers. This is the case with layer fusion schedules that try to deconstruct the operations within the layer and try to maximize the reuse of variables across layers [15]. Finally, an important aspect is to formulate the dependence graph of the entire training flow and schedule it accordingly. This can maximize parallelism while reducing memory bandwidth [16]. However, there is a dearth of research that specifically targets the training of the fully-connected layer.

This research was supported in part by the National Science Foundation under grant number CCF-1814759.

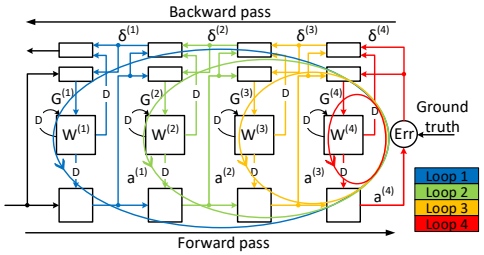



Fig. 1. Training loops for a 4-layer fully-connected neural network.

Research on the fully connected layer has been focused on exploiting sparsity during the inference phase [7], [17]. Unstructured pruning techniques have found some success at the training stage [18]; however, structured sparsity could be better suited for training [7]. Flexible architectures have shown promise at taking advantage of the relative strengths of the different flows at different stages of the CNN [8], [19], [20]. However, the overall impact is yet to be addressed completely [21]. The proposed *configurable systolic array* and interleaved scheduler maximize the use of a variable while eliminating intermediate results. The key contribution of this paper is a new scheduling approach for interleaving computations of multiple gradients, as opposed to scheduling these sequentially.

The rest of the paper is structured as follows. Section II focuses on the backpropagation equation and how systolic arrays and interleaving of gradients can be used to optimize the design. Section III evaluates the proposed methodology. Finally, in Section IV we summarize the main conclusions of the paper.

## II. Interleaving for Backpropagation Algorithm

The operation of a fully-connected layer can be easily written in terms of a matrix-matrix multiplication. Thus it is natural that many of the traditional scheduling approaches are derived from past implementations. These primarily involve developing blocks or tiles from the matrices and finding the optimal size and order to schedule these blocks. Backpropagation involves recursive feedback loops. The iteration period in recursive computing systems has a fundamental lower bound, referred as the *iteration bound* [22], [23].

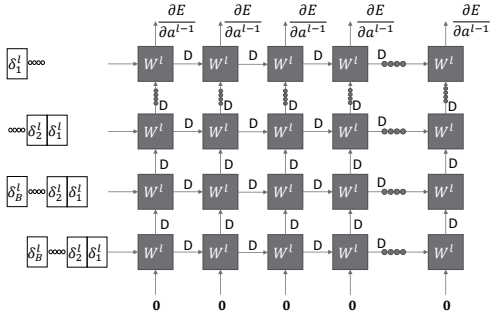


Fig. 2. Systolic array that computes $\delta^{(l-1)}$ in a weight-stationary mode.

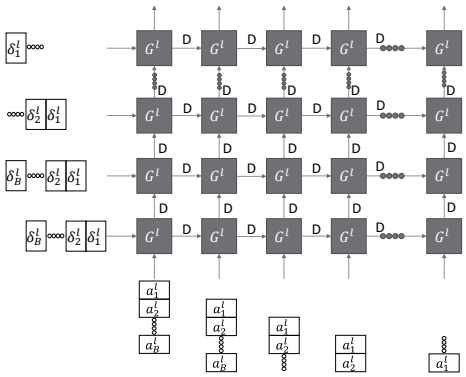


Fig. 3. Systolic array that computes $G^{(l)}$ in an output-stationary mode.

Systolic arrays [24] and other spatial architectures have inherent advantages when it comes to data reuse. Their spatial dataflow pattern allows for more efficient reuse of data. Although current implementations achieve high speedups, these do not exploit all the possible avenues of reuse available. To illustrate this, we focus on the backpropagation algorithm which is the backbone in the training of neural networks.

The forward pass or inference for the fully-connected layers is well understood and can easily be represented as a matrix-vector multiplication as follows:

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

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

where $l$ represents the layer being processed, $W^{(l)}$ is the weight matrix of the fully-connected layer, and $f$ is the activation function for that layer. This paper does not attempt to optimize the forward pass computations and Eqs. (1) and (2) are no longer discussed.

For the fully-connected layer, there are two sets of gradients that need to be computed. The first is to calculate the gradients for each weight matrix. The latter requires computation of the gradient with respect to the activation function. These can be summarized by the following set of equations:

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

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

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

where $E$ represents the loss function, $\delta^{(l)}$ represents the training error gradient backpropagated to layer $l$, and $f'$ represents the derivative of the activation function. The notation $\odot$ represents the Hadamard product of matrices.

### A. Computation of $\delta^{(l-1)}$

To compute (4) the architecture shown in Fig. 2 is considered. The array consists of $P \times Q$ processing elements (PEs), where $P$ and $Q$ represent the horizontal and vertical dimensions of the systolic array. The PEs are interconnected along the horizontal and vertical directions and each contains a pipeline register. The array is set up in a weight-stationary mode, where one of the inputs, $W$, is held constant inside the local memory of the cell. The weights are first loaded into the array from the edge. It takes $Q$ cycles where $P$ words are loaded per cycle into the systolic array. Though in this example only a single weight stored in each processing element is considered, the same technique can be extended to process multiple weights simultaneously. Once the weights are loaded, $\delta^{(l)}$ is loaded at the Western edge, and is staggered by a clock cycle with each row of the design. Once the results are calculated in each PE the partial sums are accumulated vertically in the array. The pipelined architecture is used to assume a simple PE design with a low critical path delay. This proceeds for $B$ more cycles where $B$ is the number of data points in the mini-batch for training. In a traditional design once these calculations are complete the contents of the array are no longer required and are discarded.

### B. Computation of $G^{(l)}$

To compute (5) the architecture shown in Fig. 3 is considered. To enable a more useful computation, rather than computing (5) directly, the *transpose* of the gradient is computed as follows:

$$G^{(l)T} = \left( \frac{\partial E}{\partial W^{(l)}} \right)^T = a^{(l-1)} \delta^{(l)T} \tag{6}$$

Rewriting (5) to (6) is the *key* to reducing memory access for computing the gradients in the configurable systolic array. Note that most high-level synthesis [25]–[28] systems cannot automate this reformulation. As with the earlier case, for Fig. 3, the array consists of $P \times Q$ processing elements. The array is set up in an output-stationary mode, where the partial sums $G^{(l)}$ are held constant inside the local memory of the cell.

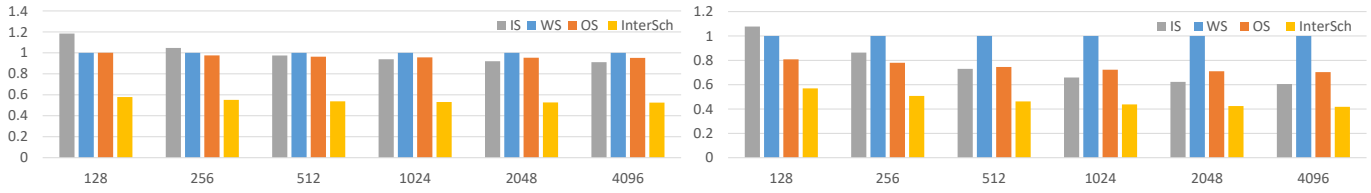
Once again, $\delta^{(l)}$ are passed into the array along the Western edge and the new input $a^{(l-1)}$ is passed into the Southern edge of the array. Both inputs are staggered by a clock cycle and passed into the array and the array processes the inputs in a wavefront manner. The gradients thus generated are accumulated *in-place* for the mini-batch of $B$. Once these calculations are complete the contents of the array have to be shifted out. That requires an additional $Q$ cycles where $P$ words are unloaded per cycle from the systolic array.

### C. Interleaving of gradients

To improve performance or reduce the number of memory accesses, traditional approaches have often looked at (4) to (6) in isolation. However, there are multiple avenues of data reuse to exploit looking at the commonalities between the equations. One way to exploit this is through interleaving, and this has been extensively studied in signal processing systems to reuse hardware and computations across different data points [29]. The key approach is to treat the entire backpropagation as a form of a feedback system that can be modeled as a data-flow graph. This enables us to apply the techniques developed in optimizing signal processing systems to

TABLE I
DATAFLOW IN THE SYSTOLIC ARRAY ALONG THE VERTICAL AND HORIZONTAL DIRECTIONS

| Time / Node | I/Os | $T_0$ | $T_1$ | $T_2$ | $T_3$ | $T_4$ |
|---|---|---|---|---|---|---|
| $PE_{x,y}$ | $PE_{x-1,y}$ | $\delta_{y,n}$ | $\delta_{y,n}$ | $\delta_{y,n+1}$ | $\delta_{y,n+1}$ | $\delta_{y,n+2}$ |
| | $PE_{x,y-1}$ | $a_{x,n}$ | $res_{x,y-1}$ | $a_{x,n+1}$ | $res_{x,y-1}$ | $a_{x,n+2}$ |
| | $res_{x,y}$ | acc($\delta_{y,n}a_{x,n}$) | $res_{x,y-1}$+$\delta_{y,n}w_{x,y}$ | acc($\delta_{y,n+1}$ $a_{x,n+1}$) | $res_{x,y-1}$+$\delta_{y,n+1}w_{x,y}$ | acc($\delta_{y,n+2}a_{x,n+2}$) |
| $PE_{x,y+1}$ | $PE_{x-1,y+1}$ | 0 | $\delta_{y+1,n}$ | $\delta_{y+1,n}$ | $\delta_{y+1,n+1}$ | $\delta_{y+1,n+1}$ |
| | $PE_{x,y}$ | 0 | $a_{x,n}$ | $res_{x,y}$ | $a_{x,n+1}$ | $res_{x,y}$ |
| | $res_{x,y+1}$ | 0 | acc($\delta_{y+1,n}a_{x,n}$) | $res_{x,y}$+$\delta_{y+1,n}w_{x,y}$ | acc($\delta_{y+1,n+1}$ $a_{x,n+1}$) | $res_{x,y}$+$\delta_{y+1,n+1}w_{x,y}$ |
| $PE_{x+1,y}$ | $PE_{x,y}$ | 0 | $\delta_{y,n}$ | $\delta_{y,n}$ | $\delta_{y,n+1}$ | $\delta_{y,n+1}$ |
| | $PE_{x+1,y-1}$ | 0 | $a_{x+1,n}$ | $res_{x+1,y-1}$ | $a_{x+1,n+1}$ | $res_{x+1,y-1}$ |
| | $res_{x,y+1}$ | 0 | acc($\delta_{y,n}a_{x+1,n}$) | $res_{x,y-1}$+$\delta_{y,n}w_{x+1,y}$ | acc($\delta_{y,n+1}a_{x+1,n+1}$) | $res_{x,y-1}$+$\delta_{y,n+1}w_{x+1,y}$ |




(a) (b)

Fig. 4. Normalized single loop performance of the interleaved scheduler versus traditional dataflow approaches while varying network size (Normalized to WS). a) Normalized number of memory access. b) Normalized number of cycles. InterSch denotes proposed Interleaved Scheduler.

improve the backpropagation algorithm. Based on the requirements from Sections II-A and II-B, this paper proposes to interleave [23] the gradient computations for (4) and (6). To enable this new paradigm of computing we propose to develop *configurable systolic arrays* that can switch between different modes of operation on a per cycle basis. This is only possible due to the reformulation of (5) to (6) as in both cases $\delta^{(l-1)}$ is input at the Western edge of the array. Thus interleaving (4) and (6) allows for the reuse of $\delta^{(l-1)}$ across both equations.

Table I summarizes the data flow and operations in the array along the vertical ($y$) and horizontal ($x$) directions. Each node in the array has inputs from the $PE_{x-1,y-1}$, and an output $res$ that represents result. $PE_{x,y}$ and $PE_{x,y+1}$ show the relationship between PEs that are adjacent to one another along the vertical direction. Similarly, $PE_{x,y}$ and $PE_{x+1,y}$ show the relationship between PEs that are adjacent to one another along the horizontal direction. $T_0$, $T_1$ *etc*. represent consecutive time steps. From Table I the inherent advantages of the systolic arrays is that once $W$, $\delta^{(l-1)}$ and $a^{(l)}$ are loaded to the array, these are reused multiple times by different PEs. The proposed interleaving also enables $\delta^{(l)}$ to be used in both calculations and it moves with a single cycle delay horizontally. Along the vertical direction the interconnects transfer data alternating between the result for (4) and (6). This reuse of $\delta$ effectively reduces the number of accesses to the on-chip memory by $B \times \lfloor \frac{N}{Q} \rfloor \times \lfloor \frac{M}{P} \rfloor$. Here, $N \times M$ is the dimension of the weight matrix.

Finally, once the gradients are obtained, the weights can be updated as per the following equation:

$$W^{(l)}(k+1) = F\left(W^{(l)}(k), G^{(l)}(k)\right) \tag{7}$$

where $k$ represents the iteration step of the algorithm and $F$ represents the gradient update optimizer such as stochastic gradient descent (SGD), Adam, *etc*. From a careful observation of the computations of (4) and the *reformulated* (6) it can be seen that each element of the gradient matrix that is generated from the above architecture is *located* in the same PE as the corresponding element of the weight matrix. This is the other advantage of reformulating (5) as it fortuitously aligns the intended variables. The gradient $G^{(l)}$ 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 (7). 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 $G^{(l)}$. Thus, a further $3 \times B \times \lfloor \frac{N}{P} \rfloor \times \lfloor \frac{M}{Q} \rfloor$ accesses are saved.

## III. PERFORMANCE EVALUATION OF INTERLEAVED GRADIENTS

### A. Methodology

To evaluate the effectiveness of the proposed methodology versus traditional dataflow models, we use the structure shown in Fig. 1 as a reference. Each layer in Fig. 1 can be modeled for the number of neurons or network size to accommodate various sizes of the fully-connected layer. To enable fast and rapid design space exploration we developed a simulation tool over the open-source python-based NN simulation framework SCALE-sim[1] [21].

For evaluation, the underlying PE array size was chosen based on the configuration of Google TPU [6], a 128x128 systolic array with a matrix-vector multiply unit, and the batch size of the input is configurable. The scope of the proposed work is to limit the number of access to the on-chip memory so all numbers will be reported for the on-chip SRAM without regard to the DRAM access and latency.

### B. Single-layer scheduling

To measure the single loop efficiency, the innermost loop in Fig. 1 (loop 4) is used as a basis for evaluation. For the evaluation of the forward pass Eq. (1), as the architecture is flexible, the proposed design is just modeled to match the number of cycles and accesses of the baseline dataflow. For comparison, we evaluate the 3 traditional dataflow models, i.e., weight-stationary (WS), output-stationary (OS), and input-stationary (IS). For the computations in the backward pass, the traditional dataflow computes each equation, Eqs. (4), (6) and (7), as a separate matrix-matrix operation, whereas in the

[1]https://github.com/ARM-software/SCALE-Sim

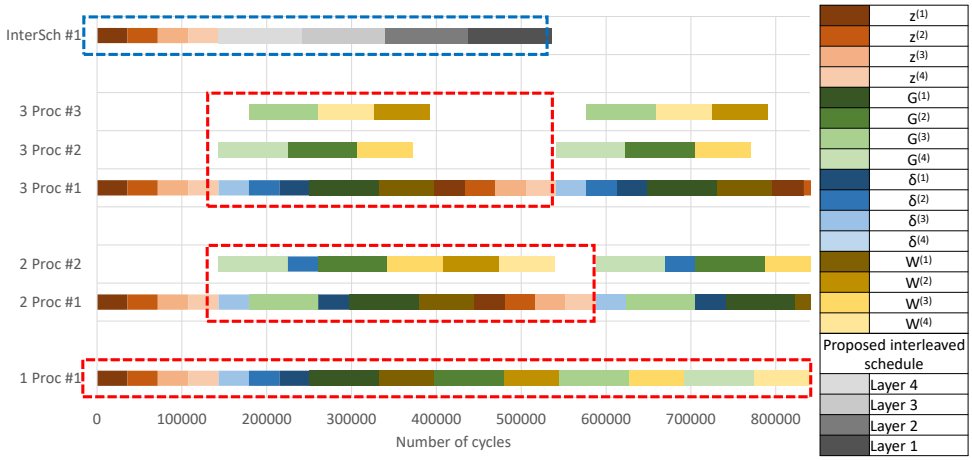



Fig. 5. Number of cycles to evaluate the proposed interleaved Scheduler versus baseline output-stationary dataflow for upto 3 Processors.

TABLE II
NUMBER OF CYCLES AND MEMORY ACCESSES FOR A 4 LAYER MLP SYSTEM

| No of Proc | Cycles ($\times 10^3$) | Utilization | Memory accesses ($\times 10^6$) |
|---|---|---|---|
| (OS) 1 | 840.08 | 100.00% | 129.14 |
| (OS) 2 | 444.96 | 94.12% | |
| (OS) 3 | 398.10 | 69.58% | |
| (IS) 1 | 536.00 | 100.00% | 76.15 |

proposed interleaved scheduler methodology the single equivalent time is stated for processing all of the equations in an interleaved manner. Performing Eqs. (2) and (3) are not shown but are assumed to be processed element-wise separately.

Fig. 4 analyzes the effect of the network size on the performance of the proposed method. It shows the normalized number of cycles and memory accesses to the local on-chip SRAM. This is obtained by sweeping and evaluating across different network sizes and batch sizes. In Fig. 4, for fixed network sizes, the values obtained are averaged across batch sizes and normalized to the value for weight-stationary. The proposed methodology reduces the overall number of cycles to compute this loop by $30\%$. This corresponds closely to the formulas provide in Section II-C. Also, the proposed method reduces the number of single-loop memory accesses by $42\%$.

### C. Multi-layer scheduling

To measure the system efficiency, all computations for the system in Fig. 1 are evaluated. At the system level, it is important to consider the dependence graph of all the computations to decide a schedule. Also, in a fully-connected layer, Eqs. (4), (6) and (7) can be computed in parallel. Thus to test the design for utilization and parallelizability we investigate 3 scenarios: a single processor, 2 processors and 3 processors. In the example shown in Fig. 1, the dependence graph is calculated and a schedule is developed for the above scenarios.

Table II summarizes the number of memory accesses to the local on-chip SRAM and the number of cycles required to completely process the system with different number of processors for conventional scheduling and using a single processor for the proposed method. As seen in Fig. 5 and Table II the proposed method reduces the overall number of processor cycles ((Number of processors) $\times$ (cycles/processor)) to compute this loop by $36\%$, $40\%$ and $55\%$ for 1, 2 and 3 processors, respectively. The proposed method reduces

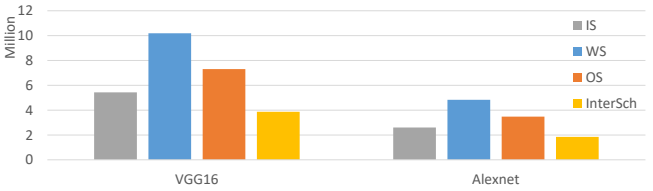



Fig. 6. Number of cycles for common CNN architectures.

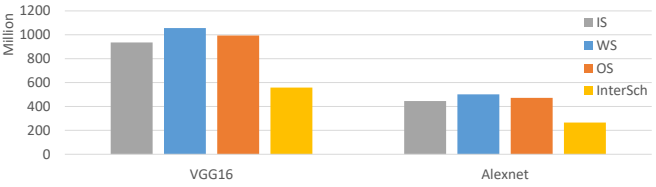



Fig. 7. Number of memory accesses for common CNN architectures.

the total memory accesses by $41\%$. This corresponds to significant savings in both processor cycles and memory accesses.

### D. Applications to FC Layers in CNNs

In order to benchmark the performance of the system we evaluate the proposed method on the fully connected layers of well known convolutional neural networks (CNNs) (VGG16 [2], Alexnet [1]). It is shown in Fig. 6 that the proposed interleaved scheduler requires $29\%$ less cycles compared to even the best data-flow chosen by a flexible architecture. In terms of memory accesses, the proposed method reduces that requirement by a minimum of $40\%$.

## IV. CONCLUSION

This paper proposes a novel scheduling scheme based on interleaving the various computations to reduce the latency and memory access of the design. It has been shown that the proposed method outperforms even the best traditional dataflow schemes by a factor of $1.4\times \sim 2.2\times$ in terms of cycles and by a factor of upto $1.9\times$ in terms of memory accesses in fully-connected layers found in common CNNs. Future work will consider the effect of sparsity and specifically structured sparsity on the training process using the proposed approach. Currently, this paper focuses exclusively on feed-forward multi-layer perceptron and it would be of interest to adapt these techniques to the convolutional and recurrent layers as well.

## REFERENCES


[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the Advances in neural information processing systems*, 2012, pp. 1097–1105.

[2] 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.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[4] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan *et al.*, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[5] M. Johnson, M. Schuster, Q. V. Le, M. Krikun, Y. Wu, Z. Chen, N. Thorat *et al.*, "Google's multilingual neural machine translation system: Enabling zero-shot translation," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 339–351, 2017.

[6] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, June 2017, pp. 1–12.

[7] C. Deng, S. Liao, Y. Xie, K. K. Parhi, X. Qian, and B. Yuan, "PermDNN: Efficient compressed DNN architecture with permuted diagonal matrices," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2018, pp. 189–202.

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

[9] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient inference engine on compressed deep neural network," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 243–254.

[10] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 269–284.

[11] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li *et al.*, "DaDianNao: A machine-learning supercomputer," in *Proceeding of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2014, pp. 609–622.

[12] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec 2017.

[13] J. Zhang and J. Li, "Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 25–34.

[14] X. Chen, S. Peng, L. Jin, Y. Zhuang, J. Song, W. Du, S. Liu, and T. Zhi, "Partition and scheduling algorithms for neural network accelerators," in *Proceedings of the International Symposium on Advanced Parallel Processing Technologies*. Springer, 2019, pp. 55–67.

[19] H. Kwon, A. Samajdar, and T. Krishna, "MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects," in *Proceedings of the International Conference on Architectural*

[15] Y. Zhuang, S. Peng, X. Chen, S. Zhou, T. Zhi, W. Li, and S. Liu, "Deep Fusion: A software scheduling method for memory access optimization," in *Proceedings of the IFIP International Conference on Network and Parallel Computing*. Springer, 2019, pp. 277–288.

[16] J. Li, G. Yan, W. Lu, S. Jiang, S. Gong, J. Wu, J. Yan, and X. Li, "TNPU: An efficient accelerator architecture for training convolutional neural networks," in *Proceedings of the Asia and South Pacific Design Automation Conference*, ser. ASPDAC '19. New York, NY, USA: ACM, 2019, pp. 450–455.

[17] J. Zhu, J. Jiang, X. Chen, and C. Tsui, "SparseNN: An energy-efficient neural network accelerator exploiting input and output sparsity," in *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018, pp. 241–244.

[18] J. Zhang, X. Chen, M. Song, and T. Li, "Eager pruning: Algorithm and architecture support for fast training of deep neural networks," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019, pp. 292–303.
*Support for Programming Languages and Operating Systems*, 2018, pp. 461–475.

[20] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017, pp. 553–564.

[21] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "SCALE-Sim: Systolic CNN Accelerator Simulator," *arXiv e-prints*, p. arXiv:1811.02883, Oct 2018.

[22] 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.

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

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

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

[26] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang, "HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 242–251.

[27] T. Moreau, T. Chen, L. Vega, J. Roesch, E. Yan, L. Zheng, J. Fromm *et al.*, "A hardware–software blueprint for flexible deep learning specialization," *IEEE Micro*, vol. 39, no. 5, pp. 8–16, Sep. 2019.

[28] 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.

[29] 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.