Accelerating Deep Neural Network Computation on a Low Power Reconfigurable Architecture

Y. Xiong*, J. Zhou*, S. Pal†, D. Blaauw†, H.-S. Kim†, T. Mudge†, R. Dreslinski†, and C. Chakrabarti*
*School of ECEE, Arizona State University, Tempe †Dept. of EECS, University of Michigan, Ann Arbor

***Abstract*—Recent work on neural network architectures has focused on bridging the gap between performance/efficiency and programmability. We consider implementations of three popular neural networks, ResNet, AlexNet and ASGD weight-dropped Recurrent Neural Network (AWD RNN) on a low power programmable architecture, Transformer. The architecture consists of light-weight cores interconnected by caches and crossbars that support run-time reconfiguration between shared and private cache mode operations. We present efficient implementations of key neural network kernels and evaluate the performance of each kernel when operating in different cache modes. The best-performing cache modes are then used in the implementation of the end-to-end network. Simulation results show superior performance with ResNet, AlexNet and AWD RNN achieving 188.19 GOPS/W, 150.53 GOPS/W and 120.68 GOPS/W, respectively, in the 14 nm technology node.**

I. Introduction

Deep neural networks (DNN) have shown superior performance in multiple domains, such as image processing, computer vision and natural language processing. The inherent parallelism in these networks has been well exploited by graphics processing units (GPU) [1], [2], [3] which also offer high programmability. Unfortunately most GPUs have high power consumption, making them unsuitable for edge devices. While there are GPUs integrated in Mobile SoCs designed for low power computation [4], they are not well suited for irregular workloads such as sparse workloads [5].

Several customized application-specific integrated circuits (ASIC) have been proposed to achieve high power and area efficiency [6], [7], [8], [9]. However many of them lack programmability that is required to support different network architectures To achieve both high computational efficiency and programmability, reconfigurable hardware has been proposed. An example is field programmable gate arrays (FPGA) which have been shown to provide high efficiency on machine learning [10], [11] and DNN [12] algorithms but at the expense of large area and power consumption. To balance programmability and power consumption, coarse-grained reconfigurable architectures (CGRA) have been proposed for k-means, SVM and CNN in [13] and for a customized six-layer fully connected network in [14]. Other recent architectures include DNA [15] and NeuroCGRA [16] which provide reconfigurability in the data path and optimize memory access patterns to achieve high resource utilization.

In this work, we map contemporary neural networks, namely, ResNet [17], AlexNet [18], AWD RNN [19] on a low power reconfigurable architecture similar to OuterSPACE [5], [20], [21]. The architecture, called Transformer, consists of a number of general-purpose processing elements (GPE) connected through a flexible crossbar to a two-level on-chip memory hierarchy as shown in Fig. 1. The on-chip L1, L2 caches can be operated in shared and private modes and can be reconfigured at runtime with minimal overhead.

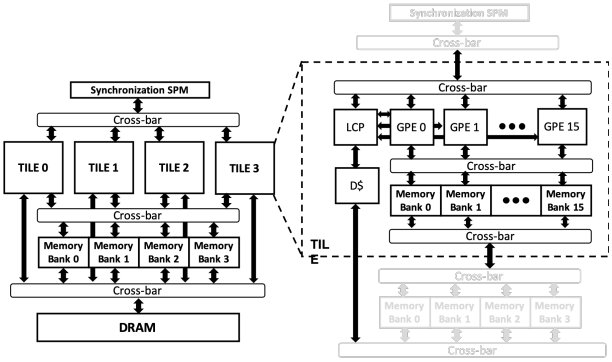

Fig. 1. Block diagram architecture of Transformer.

We consider implementations on a Transformer configuration with 4 tiles and 16 GPEs per tile. First, we analyze the performance of the common kernels in all these networks for different cache modes (shared vs private). Next, we evaluate the performance of the three networks and find that ResNet and AWD RNN have the best performance when operating in L1 private-L2 private (L1P-L2P) cache mode while AlexNet benefits from using L1P-L2P in some layers and L1P-L2S in others. The contribution of this work includes:

- Efficient implementations of key kernels with very high core utilization.
- For each kernel, determination of the cache mode that results in the best performance.
- Gem5 simulation results of Transformer when operating with the best cache mode show superior performance with ResNet, AlexNet and AWD RNN achieving 188.19, 150.53 and 120.68 GOPS/W, respectively.

II. Key kernels in deep neural networks

While the structure of DNNs vary according to the application, most use the same set of kernels. For example, all CNNs use 2D convolution layers with max pooling to extract features and many use fully connected layers for classification and batch normalization for regularization. Next, we briefly describe these key kernels and their implementation on Transformer.

978-1-7281-3320-1/20/$31.00 ©2020 IEEE

A. Fully Connected Layer

Fully connected layer is the most commonly used classifier structure. It is essentially a dense matrix-vector multiplication, where the matrix of weights of size $N \times M$, is multiplied by the input vector of size $M \times 1$ to generate an output vector of size $N \times 1$. In this work, both input vector and weight matrix are stored in main memory and after the matrix-vector computation, the output vector is stored back in main memory. To maximize GPE utilization, we distribute the matrix-vector multiplication computation across GPEs. Each GPE reads the input vector and one row of weight matrix from main memory and computes one value of output vector. The output value is passed through a non-linear function such as Sigmoid or ReLU and written back to the main memory.

B. 2D Convolution

The most computationally intensive kernel in convolution neural networks (CNN) is 2D convolution. Convolution kernel sizes range from 3×3 in ResNet [17] to 5×5 and 11×11 in AlexNet [18]. In CNNs, the inputs, kernel and outputs have multiple channels. Assume input has a channels and output has b channels, then there are b sets of convolution kernels each having a channels. In our 2D convolution implementation, input matrices and convolution kernels are stored in main memory. Each GPE reads one set of convolution kernels and all channels of input matrices from main memory and carries out multiplication-accumulation operation between the kernel values and input values over all input channels to compute the final output. The process is repeated for all b sets. This implementation effectively reuses the kernel values and allow inputs to be reused when the L2 is configured as shared cache.

C. Pooling

Pooling is another significant kernel which enables CNNs to concentrate on the most representative part of the input array. Examples include max pooling and average pooling. The memory access pattern of max pooling is similar to 2D convolution. Each GPE works on one channel at a time. It reads the values corresponding to a receptive area, computes the max value, writes it back to the main memory and then moves to the next receptive area.

D. Batch Normalization

Batch normalization [22] is a popular method for regularization. It requires computation of the mean $E[x]$, and variance $Var[x]$. Both these terms can be calculated with reduced overhead if the running sum of the output ($\sum(.)$) and the running sum of square of outputs ($\sum(.)^2$) is updated after every output is computed. For example, in 2D convolution, when one GPE finishes the multiplication-accumulation computation of one output value, it writes back the value and updates the sum and sum of square values. The mean and variance of a batch can be calculated easily using these two terms. The workload in a batch is equally distributed to all tiles. Each GPE works on one input channel at a time. It reads the input along with the mean and variance values, computes batch normalization and writes back the batch normalized output.

E. Element Wise Operations

Element wise operations include element-wise addition and multiplication between vectors as in the LSTM RNN [19] and element wise 2D convolution using 1×1 kernel as in ResNet [17]. In the computation of element-wise 2D convolution, all the values in input matrix is multiplied with a constant weight. Here each GPE reads one row of input, computes the product and writes back to the main memory. This implementation also results in good workload distribution across GPEs.

III. Performance Results

A. Platform Configuration

In this work, we use a Transformer configuration with 4 tiles and 16 GPEs in a tile. GPEs are based on Arm cores working at 1.0 GHz. The L1 cache consists of 16 cache banks each of size 4 kB. The L2 cache consists of 4 cache banks each of size 16 kB. All simulations are done on the gem5 cycle-accurate simulator [23], [24]. Simulation results such as execution time, cache misses, cache misses are obtained from gem5 statistics. Throughput is reported in terms of raw giga-operations per second (GOPS) and algorithmic giga-floating-point-operations per second (GFLOPS). In addition, the metric of GPE utilization (i.e. number of cycles that a GPE spends in computation, relative to total number of cycles in the program) is reported as a measure of the efficiency of the workload implementation. The power estimations are done for the 14 nm node. Dynamic power is calculated using active cycles and access numbers obtained from gem5 statistics. The static power and transaction energy per access of the reconfigurable cache banks are modeled using CACTI 7.0 [25].

B. Kernel-Level Results

1) Fully Connected Layer: We choose to first showcase computation in a fully connected layer of ResNet. Here, the size of the input and output vectors are 128×1 and 10×1, respectively, and the weight matrix is of size 10×128. The performance numbers are shown in Table I. In the table, 'Config' indicates configuration of L1 and L2 cache where 'S' means shared cache and 'P' means private cache. 'Exe Time' refers to execution time. The results shows that L1-P/L2-P mode achieves the shortest execution time and highest GOPS/W and GFLOPS/W.

TABLE I
Fully Connected Layer for Small Size matrices

Config	Exe Time (ms)	Power (W)	GOPS/W	GFLOPS/W
L1-S/L2-S	0.050	0.180	24.81	7.30
L1-S/L2-P	0.028	0.199	49.09	11.79
L1-P/L2-S	0.017	0.196	63.25	18.65
L1-P/L2-P	**0.016**	**0.195**	**66.69**	**19.67**

Table II shows that in the computation of fully connected layer with small weight matrices (10×128 and 1150×1150), private/private mode has the shortest execution time. However when the matrix size increases, L1-P/L2-P may not be the most efficient configuration. For instance, when the weight matrix size is large as in 2048×2048 in AlexNet, L1-P/L2-S

achieves the shortest execution time. This is because in such cases private cache is not large enough to hold both the inputs and weights resulting in cache misses.

TABLE II
EXECUTION TIME FOR FULLY CONNECTED LAYER FOR DIFFERENT WEIGHT MATRIX SIZES

Config	Weight Matrix Size $N \times M$		
	10×128	1150×1150	2049×2048
L1-S/L2-S	0.050	3.362	12.494
L1-S/L2-P	0.028	3.945	16.782
L1-P/L2-S	0.017	2.928	**7.040**
L1-P/L2-P	**0.016**	**2.814**	8.051

2) 2D Convolution: We first present results for input size of $32 \times 32 \times 3$ (size of CIFAR-10 images) and convolution kernel of size 3×3. We set the output channel number to be 4 and use a small batch size of 16 (4 per tile). The performance numbers in Table III show that the shortest execution time and highest GOPS/W and GFLOPS/W is achieved when L1 and L2 are both in private cache mode. The private L1 cache has high cache hit rates ($> 98\%$), which indicates that 4 kB private cache is adequate for storing input and kernel values.

TABLE III
3X3 2D CONVOLUTION ON $32 \times 32 \times 3$ INPUT

Config	Exe time(ms)	Power(W)	GOPS/W	GFLOPS/W
L1-S/L2-S	0.37	0.289	177.66	69.77
L1-S/L2-P	0.36	0.294	187.36	73.84
L1-P/L2-S	0.31	0.292	235.24	87.01
L1-P/L2-P	**0.29**	**0.298**	**242.66**	**89.74**

Next, we investigate the performance of 2D convolution for larger input sizes and larger kernel sizes. Table IV presents results for input size of 13×13, 32×32 and 55×55, where 13×13 and 55×55 are intermediate input sizes in AlexNet. We see that for almost all input sizes, private-private mode is the best choice with the shortest execution time.

TABLE IV
EXE TIME FOR 3X3 2D CONVOLUTION WITH DIFFERENT INPUT SIZE

Input Size	13x13	32x32	55x55
L1-S/L2-S	0.255	0.374	4.676
L1-S/L2-P	0.228	0.362	4.439
L1-P/L2-S	**0.055**	0.307	0.823
L1-P/L2-P	0.056	**0.292**	**0.815**

Table V presents results for different kernel sizes when the input size is $32 \times 32 \times 3$. We find that for large kernel sizes, such as 15×15, shared cache mode is better. This is because a shared L1 cache provides for a larger effective cache size and is thus beneficial when inputs and kernel values are shared by multiple GPEs.

TABLE V
EXE TIME FOR 2D CONVOLUTION WITH DIFFERENT KERNEL SIZE

Kernel Size	3x3	5x5	9x9	11x11	15x15
L1-S/L2-S	0.369	1.968	2.562	3.683	6.167
L1-S/L2-P	0.360	1.253	2.565	3.663	**6.132**
L1-P/L2-S	0.307	0.622	2.286	3.960	7.336
L1-P/L2-P	**0.292**	**0.604**	**2.071**	**3.362**	6.258

3) Element-Wise 2D Convolution: Table VI shows the performance results for 1×1 convolution on an input size of $32 \times 32 \times 3$ and 4 output channels. Results shows that private-shared mode has the lowest execution time. Since each GPE works independently and there is no reuse of inputs, private L1 cache has the best performance.

TABLE VI
1X1 2D CONVOLUTION

Config	Exe time(ms)	Power(W)	GOPS/W	GFLOPS/W
L1-S/L2-S	2.22	0.169	6.68	2.95
L1-S/L2-P	0.955	0.187	13.95	6.15
L1-P/L2-S	**0.110**	**0.227**	**98.48**	**43.57**
L1-P/L2-P	0.113	0.230	97.04	42.87

4) Max Pooling: The input size is chosen to be $32 \times 32 \times 3$ and pooling stride is 2×2 as in ResNet. From Table VII, we see that private-shared has the best performance. Each GPE works independently on non-overlapping partitions of the inputs and thus private L1 is the most beneficial.

TABLE VII
MAX POOLING

Config	Exe time(ms)	Power(W)	GOPS/W	GFLOPS/W
L1-S/L2-S	0.398	0.173	13.45	8.07
L1-S/L2-P	0.205	0.184	25.61	14.78
L1-P/L2-S	**0.028**	**0.228**	**143.07**	**85.92**
L1-P/L2-P	0.032	0.227	128.24	76.20

For element-wise operations as well as max pooling, L2 shared cache has a slight advantage over private L2 cache. This is because each GPE reads all input matrices and so shared cache provides for an effective larger cache size.

C. Network-Level Results

1) ResNet: We evaluate the performance of ResNet with 3 residual blocks. The input size is $32 \times 32 \times 3$ and output is a 10×1 vector. Each residual block consists of a convolution layer followed by batch normalization and max pooling. There is also an element-wise convolution layer on the skip path. The convolution layers use kernel size of 3×3 with a stride of 1. The zero padding is set to be 1 to keep the input and output sizes the same. The max pooling kernel size is 2×2. The output of the last residual block is max-pooled and fed into a fully connected layer for classification.

Execution time distribution of each kernel is described in Fig. 2. Since 2D convolution (3×3, 1×1) is dominant, the use of L1-P cache decreases the overall execution time significantly. The total execution times with L1-P/L2-P mode and L1-P/L2-S are significantly shorter than the other two modes.

The GPE utilization for most kernels is very high with 3×3 convolution at 97.63%, pooling at 91.91%, batch norm at 99%. The utilization of 1×1 convolution is the lowest at 88.46%. This is because the number of computations here is low compared to the number of memory accesses. The overall GPE utilization of the ResNet implementation is 95.54%.

The overall performance for different cache modes is presented in Table VIII. The optimal configuration refers to the case where the configuration with the shortest execution

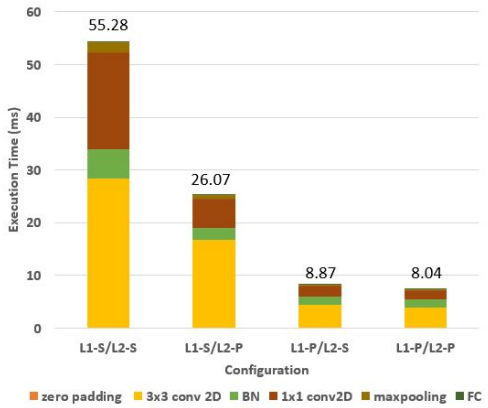

Fig. 2. Execution Time of ResNet Kernels

time is used for each kernel. Thus, 3×3 convolution, batch normalization and fully connected layer use the L1-P/L2-P mode, and zero padding, 1×1 convolution and max pooling use the L1-P/L2-S mode. Overall the optimal configuration achieves 53.73 GFLOPS/W and 188.19 GOPS/W.

TABLE VIII
PERFORMANCE OF RESNET

Config	Exe time(ms)	Power(W)	GOPS/W	GFLOPS/W
L1-S/L2-S	55.28	0.188	44.21	10.62
L1-S/L2-P	26.07	0.217	80.90	19.50
L1-P/L2-S	8.87	0.247	181.90	50.20
L1-P/L2-P	8.04	0.258	175.55	53.15
Optimal	**7.81**	**0.263**	**188.19**	**53.73**

2) AlexNet: AlexNet is a popular CNN which has convolution layers with different input sizes and kernel sizes, and three fully connected layers. The input sizes vary from 224×224 to 13×13 and the kernel sizes vary from 11×11 to 3×3. The fully connected layers have large weight matrices of sizes 4096×4096 and 4096×1000. The overall GPE utilization of the AlexNet implementation is 98.12%.

TABLE IX
PERFORMANCE OF ALEXNET

Config	Execution time(ms)	Power(W)	GOPS/W	GFLOPS/W
L1-S/L2-S	450.36	0.233	106.09	48.47
L1-S/L2-P	491.51	0.232	94.86	43.50
L1-P/L2-S	317.24	0.268	121.85	59.84
L1-P/L2-P	324.91	0.255	131.23	60.17
Optimal	**273.17**	**0.2633**	**150.53**	**69.04**

The execution times for each layer is described in Fig. 3. Layers 1 and 2 have large input and kernel sizes and so shared L2 cache has better performance, while layers 3 through 5 have small input and kernel sizes and so L1-P/L2-P mode has significantly shorter execution time compared to L1-P/L2-S.

Table IX evaluates the performance of AlexNet under different cache modes. We see that for fixed configuration, L1-P/L2-S has the shortest execution time. However, when Layer1, Layer2 and fully connected layers use L1-P/L2-S mode, and Layers 3 through 5 use L1-P/L2-P mode, the execution time decreases by 13.9% compared to the L1-P/L2-S mode.

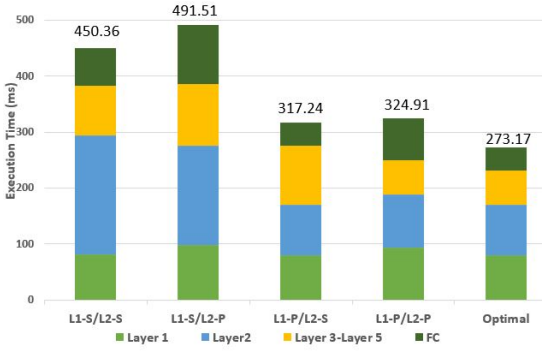

Fig. 3. Execution Time of AlexNet Layers

3) AWD LSTM RNN: RNN with LSTM cells are widely used to process speech and text. AWD LSTM RNN is a RNN that use a three-layer LSTM model with 1150 units in the hidden layer and input size of 400 [19]. A dropout of 0.5 is applied to the recurrent weight matrices for regularization.

In each LSTM cell, there are four gates where each gate computes two matrix-vector multiplications. This is followed by element-wise addition and multiplication. The performance results are shown in Table X.

TABLE X
PERFORMANCE OF AWD LSTM RNN

Config	Execution time(ms)	Power(W)	GOPS/W	GFLOPS/W
L1-S/L2-S	29.960	0.260	100.24	49.23
L1-S/L2-P	31.565	0.256	96.50	47.46
L1-P/L2-S	23.869	0.268	121.85	59.84
L1-P/L2-P	**23.279**	**0.277**	**120.68**	**59.26**

In our RNN implementation, more than 95% of the time is spent in gate computation, which is essentially fully connected layer computations with weight matrices of sizes 400×1150 and 1150×1150 and the overall GPE utilization is 98.26%. L1-P/L2-P mode has the best performance for these matrix sizes, resulting the shortest execution time.

IV. CONCLUSION

This work investigated the performance of commonly-used kernels in DNN computations for different cache mode configurations on a low power reconfigurable architecture with two levels of cache hierarchy. All kernels were mapped on this architecture to achieve greater than 90% core utilization for most cases. Kernel level evaluation show that private-private mode has the lowest execution time. End to end implementation of ResNet, AlexNet and AWD LSTM RNN showed very high performance of 188.19, 150.53 and 120.68 GOPS/W, respectively, in the 14nm node.

Acknowledgment: The material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7864. The views and conclusions contained herein are those of the authors and do not represent the official policies or endorsements, either expressed or implied, of ARFL and DARPA or the U.S. Government.

REFERENCES

[1] T. NVIDIA, "K40 gpu active accelerator," *Board specification*, 2013.

[2] P.-K. Tsung, S.-F. Tsai, A. Pai, S.-J. Lai, and C. Lu, "High performance deep neural network on low cost mobile gpu," in *2016 IEEE International Conference on Consumer Electronics (ICCE)*, 2016, pp. 69–70.

[3] S. Feng, S. Pal, Y. Yang, and R. G. Dreslinski, "Parallelism analysis of prominent desktop applications: An 18-year perspective," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2019, pp. 202–211.

[4] K.-T. Cheng and Y.-C. Wang, "Using mobile gpu for general-purpose computing–a case study of face recognition on smartphones," in *Proceedings of 2011 International Symposium on VLSI Design, Automation and Test*. IEEE, 2011, pp. 1–4.

[5] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "Outerspace: an outer product based sparse matrix multiplication accelerator," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 724–736.

[6] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ACM Sigplan Notices*, vol. 49, no. 4. ACM, 2014, pp. 269–284.

[7] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 609–622.

[8] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 92–104.

[9] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *arXiv preprint arXiv:1807.07928*, 2018.

[10] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "Cnp: An fpga-based processor for convolutional networks," in *2009 International Conference on Field Programmable Logic and Applications*, 2009, pp. 32–37.

[11] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 26–35.

[12] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 161–170.

[13] S. Cadambi, A. Majumdar, M. Becchi, S. Chakradhar, and H. P. Graf, "A programmable parallel accelerator for learning and classification," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 2010, pp. 273–284.

[14] Y. Li and A. Pedram, "Caterpillar: Coarse grain reconfigurable architecture for accelerating the training of deep neural networks," in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2017, pp. 1–10.

[15] F. Tu, S. Yin, P. Ouyang, S. Tang, L. Liu, and S. Wei, "Deep convolutional neural network architecture with reconfigurable computation patterns," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 8, pp. 2220–2233, 2017.

[16] S. M. Jafri, T. N. Gia, S. Dytckov, M. Daneshtalab, A. Hemani, J. Plosila, and H. Tenhunen, "Neurocgra: A cgra with support for neural networks," in *2014 International Conference on High Performance Computing & Simulation (HPCS)*, 2014, pp. 506–511.

[17] 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.

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

[19] S. Merity, N. S. Keskar, and R. Socher, "Regularizing and optimizing lstm language models," *arXiv preprint arXiv:1708.02182*, 2017.

[20] S. Pal, D.-h. Park, S. Feng, P. Gao, J. Tan, A. Rovinski, S. Xie, C. Zhao, A. Amarnath, T. Wesley *et al.*, "A 7.3 m output non-zeros/j sparse matrix-matrix multiplication accelerator using memory reconfiguration in 40 nm," in *2019 Symposium on VLSI Technology*. IEEE, 2019, pp. C150–C151.

[21] D.-H. Park, S. Pal, S. Feng, P. Gao, J. Tan, A. Rovinski, S. Xie, C. Zhao, A. Amarnath, T. Wesley *et al.*, "A 7.3 m output non-zeros/j, 11.7 m output non-zeros/gb reconfigurable sparse matrix-matrix multiplication accelerator," *IEEE Journal of Solid-State Circuits*, 2020.

[22] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[23] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The m5 simulator: Modeling networked systems," *Ieee micro*, no. 4, pp. 52–60, 2006.

[24] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[25] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, p. 14, 2017.