Deep Partitioned Training From Near-Storage Computing to DNN Accelerators

Yongjoo Jang, *Student Member, IEEE*,
Sejin Kim, *Student Member, IEEE*,
Daehoon Kim, *Member, IEEE*,
Sungjin Lee, *Member, IEEE*, and
Jaeha Kung, *Member, IEEE*

Abstract—In this letter, we present deep partitioned training to accelerate computations involved in training DNN models. This is the first work that partitions a DNN model across storage devices, an NPU and a host CPU forming a unified compute node for training workloads. To validate the benefit of using the proposed system during DNN training, a trace-based simulator or an FPGA prototype is used to estimate the overall performance and obtain the layer index to be partitioned that provides the minimum latency. As a case study, we select two benchmarks, i.e., vision-related tasks and a recommendation system. As a result, the training time reduces by 12.2∼31.0 percent with four near-storage computing devices in vision-related tasks with a mini-batch size of 512 and 40.6∼44.7 percent with one near-storage computing device in the selected recommendation system with a mini-batch size of 64.

Index Terms—DNN accelerators, near-storage computing, training deep neural networks, workload partitioning

1 Introduction

TO enhance the performance of computing Deep Neural Networks (DNNs), there is a plethora of research work in the domain of accelerator design [1], [2], [3]. There are two different research directions in developing DNN accelerators or so-called NPUs: i) low-power inference engines and ii) high-performance training engines. For low-power inference, a variety of dataflow models or hardware techniques have been proposed [1]. To reduce the training time, specialized tensor cores are designed or high-density DRAMs are utilized [2], [3]. This paper focuses on the training of a DNN model which is a challenging task as a bulk of training data has to travel across expensive interconnect interfaces.

Recently, both academia and industry have focused on moving compute capability to storage devices in dealing with data-intensive workloads [4], [5]. As data resides in the storage server, it is natural to process some of computations then reduce data size to alleviate the burden for SSD-CPU communication. For DNN training with a small dataset, it is possible to move the entire training data to DRAM attached to an NPU then sample mini-batches directly from the DRAM with higher bandwidth. If the training data exceeds the DRAM capacity, however, mini-batches have to be sampled from the storage and then fetched to the NPU system. Yet, none of the prior work discusses the possible advantage of utilizing near-storage computing for high-performance DNN training.

More importantly, there is a lack of research effort in bridging these two approaches, DNN accelerators and near-storage computing, to maximize the overall throughput in training DNN models. In this work, we put these decoupled hardware fabrics together so that each takes charge of a fraction of DNN training workloads using a hybrid of data parallelism and layer partitioning. This is the *first work that partitions the DNN model across storage devices, an NPU and a CPU for the training process*.

• *The authors are with the Department of Information and Communication Engineering, Daegu Gyeongbuk Institute of Science and Technology, Daegu 42988, South Korea. E-mail: {dracol, sejink06, dkim, sungjin.lee, jhkung}@dgist.ac.kr.*

Manuscript received 16 Apr. 2021; accepted 14 May 2021. Date of publication 19 May 2021; date of current version 9 June 2021.
(Corresponding author: Jaeha Kung.)
Digital Object Identifier no. 10.1109/LCA.2021.3081752

2 Background

2.1 Process of DNN Training

- Forward Propagation (FP): The forward propagation is equivalent to the inference in deep learning. Depending on the layer type, a DNN accelerator performs the corresponding computation, e.g., a convolution with 3x3 kernel (conv-3 layer) or a matrix multiplication (fully-connected layer). The computations at each DNN layer for a single training sample during the forward propagation can be expressed as

$$O[k][x][y] = \sum_{c=0}^{C-1}\sum_{s_x=0}^{S-1}\sum_{s_y=0}^{S-1} W[k][c][s_x][s_y]\cdot I[c][x+s_x][y+s_y], \tag{1}$$

where W is weight kernel, 'SxS' is the kernel size, 'C' and 'K' are the number of input (I) and output (O) channels, and 'XxY' is the size of a feature map (fmap). For dense processors (CPU/GPU) or accelerators (TPU), convolutions are converted to a matrix multiplication by applying an '`Im2Col`' operation. Thus, it is possible to assume a generic DNN operation as a GEMM operation [1].

- Backward Propagation (BP): After the forward propagation is completed over a mini-batch with size B, the loss $\mathcal{L}$ for each training sample is computed. Then, the loss is backpropagated to compute gradient maps at each layer (G_I). The gradient propagation is performed by

$$G_I[c][x][y] = \sum_{k=0}^{K-1}\sum_{s_x=0}^{S-1}\sum_{s_y=0}^{S-1} W_{flip}[c][k][s_x][s_y] \cdot G_O[k][x+s_x][y+s_y], \tag{2}$$

where W_{flip} is the flipped kernel and G_I and G_O are the input and output gradient maps. Note that W_{flip} also rearranges the channel dimension as the input to the layer is an output gradient map (G_O) to compute G_I.

- Weight Update (WU): While backpropagating the loss, Hadamard product is performed between the gradient map and its corresponding input feature map. This is identical to a convolution operation on a relevant input region with the output gradient map (G_O) as a kernel. It is performed on every input and output channel pair (c, k). Then, the $\Delta W[k][c]$ is computed and we can update the corresponding weight kernel.

2.2 Challenges in DNN Training

In this work, we leverage the benefit of layer partitioning across near-storage computing devices and an NPU hosted by CPU on two representative DNN models, i.e., Convolutional Neural Network (CNN) and Deep Learning Recommendation Model (DLRM). Note that this work focuses on scaling up a single compute node by using near-storage computing devices with the NPU. The training of a DNN model is challenging when the training dataset becomes large and do not fit in off-chip DRAM. For instance, ImageNet dataset requires 150 GB (compressed size) for training which far exceeds the capacity of a high-density HBM stack (8∼16 GB [6]). Thus, sampling and processing mini-batches at multiple near-storage devices (*data parallelism*) will greatly reduce the computation load on the NPU (*layer partitioning*). In addition, as projected in Table 1, the size of embedding tables for a production-scale recommender system exceeds the NPU DRAM capacity (12 GB/16 GB for expensive HBM2/2E) [7], [8]. Thus, it is reasonable to do `reduce` operations, i.e., sum, average or concat,

TABLE 1
The Size of Embedding Tables Projected by Referring to [7], [8]

	Bandana [7]	RMC [8]
# of Emb. Tables	8	4 ∼40
Size / Emb. Table	1.2 ∼2.4GB	10MB ∼3-4GB

TABLE 2
FPGA Specifications for a SmartSSD and a Host-Side NPU

FPGA Specifications	LUTs	FFs	DSP Slices
Kintex KU15P (smartSSD) [5]	523K	1,045K	1,968
Alveo U250 (NPU) [10]	1,728K	3,456K	12,288

on embedding vectors near storage and fetch the result to the NPU for computing the following MLPs.

3 Deep Partitioned Training

3.1 Overall System Architecture

We present a system architecture for efficient DNN training by partitioning DNN layers across near-storage computing units, i.e., smartSSDs [5] and an NPU/CPU (Fig. 1). For NPU design, we select a 2D systolic array similar to Google TPU [2]. The size of a systolic array in a smartSSD or an NPU is selected by implementing RTL on each FPGA (Section 3.2). The FPGA specifications of the smartSSD and NPU are provided in Table 2. The bandwidths of various interfaces used in our simulation are provided in Fig. 1. The bandwidth within the storage is limited by the I/O parallelism of underlying NAND chips and PCIe interconnection speed. Also, DRAM bandwidth in the storage is assumed to be 4x lower than the one attached to CPU/NPU. As the PCIe bandwidth assigned to storage devices is limited to 16GB/s, we may allow up to four near-storage computing units.

3.2 Selection of NPU Size

As half-precision (FP16) is enough for training DNNs, we synthesized a systolic array with FP16 on each FPGA and obtained the maximum allowable array size. As a result, a smartSSD allows a 32x16 systolic array and an NPU allows a 48x48 array using FP16 (test case ①). The size of a systolic array is limited by the number of available DSP slices. As LUTs and FFs are not fully utilized, we also implemented a systolic array based on the block floating-point presented in [9]. By using block FP16 (Block16), we can increase the size of a systolic array by 4× for both smartSSD and NPU (test case ②). To consider the host-side NPU as an ASIC implementation, e.g., Google TPU (bfloat16), we also tested a 128x128 systolic array as an NPU with a 64x32 array at each smartSSD (test case ③). We actually trained and tested a CNN model with data types of the test case ①∼③ reaching the same accuracy and convergence speed.

3.3 Selection of Partition Index

The problem of layer partitioning across smartSSDs and an NPU can be formulated by

$$\begin{aligned} \mathrm{partition\_idx} &= \underset{i}{\mathrm{argmin}}(B \cdot \max(\mathrm{FP}_{\mathrm{SSD}}^{0:i-1}, \mathrm{FP}_{\mathrm{NPU}}^{i:N-1}) \\ &+ \mathrm{T}_{\mathrm{PCIe}} + B \cdot (\mathrm{BP}_{\mathrm{NPU}}^{i:N-1} + \mathrm{BP}_{\mathrm{SSD}}^{0:i-1}) + \mathrm{WU}_{\mathrm{total}}), \\ \mathrm{WU}_{\mathrm{total}} &= \mathrm{WU}_{\mathrm{NPU}}^{i:N-1} + \mathrm{WU}_{\mathrm{SSD}}^{0:i-1} + \mathrm{T}_{\mathrm{interSSD}}^{0:i-1}, \end{aligned} \tag{3}$$

where 'i' is the partition index, 'B' is the mini-batch size, and FP, BP, WU represent the latency required for the forward pass, backward pass, and weight update. The subscript 'NPU' or 'SSD' indicates where the computation happens. During the forward propagation, the pipelined execution is allowed for higher performance. '$\mathrm{T}_{\mathrm{PCIe}}$' is the latency for moving fmaps from smartSSDs to the NPU. '$\mathrm{T}_{\mathrm{interSSD}}^{0:i-1}$' is the latency of inter-SSD communication required for the gather, reduce and broadcast operations on locally computed gradients (ΔW) at multiple smartSSDs. To solve Eq. (3), we sweep 'i' from layer '0' to 'N-1' in a DNN model with 'N' layers. After evaluating every possible 'i' with our simulator (Section 3.4), we select the layer index providing the minimum latency.

3.4 Workflow of Our Simulator

For the evaluation of Eq. (3), we implement a trace-based simulator, i.e., DPT-sim, in the granularity of accessing each component in the system shown in Fig. 1. The trace file stores the list of operations by (tensor type, tensor index, storage or memory RD/WR time, NPU start/end time). The unit size of data read from SSDs is assumed to be 4KB, i.e., logical block size, for the simple latency projection. For accessing DRAM and SRAM, we consider the unit of data bus width for the estimation of access latency. For NPU design, we implement a cycle-level simulator of a 2D systolic array that deals with 2-Byte operands. As various types of DNN layers can be mapped to a set of GEMM operations, the systolic array can be used for DNN computation in general.

The proposed system exploits the data parallelism at multiple smartSSDs and the layer partitioning between smartSSDs and the NPU (Fig. 2). A mini-batch with a fixed size, e.g., 32, 64 or 128, is

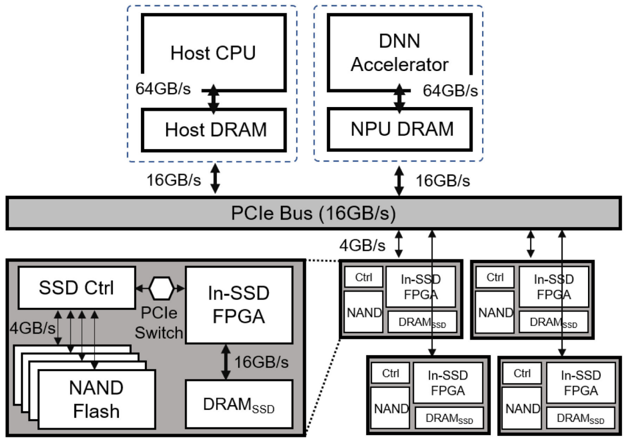

Fig. 1. The overall system architecture for deep partitioned training of DNN models using a hybrid of data parallelism and layer partitioning.

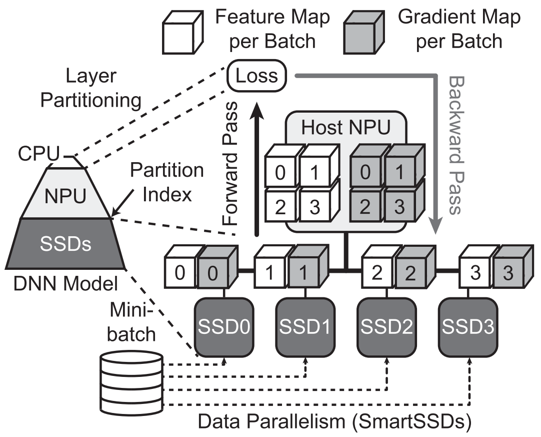

Fig. 2. A hybrid of data parallelism and layer partitioning of the proposed deep partitioned training for CNN models.

TABLE 3
The Required Number of Cycles for Training CNN Models at Various Mini-Batch Sizes Only With a Host-Side NPU (Baseline)

CNN Model	Latency with 96×96 Systolic Array in NPU (# of Cycles)				
	B=32	B=64	B=128	B=256	B=512
ResNet-50	58,929,786	116,910,970	262,873,338	464,798,074	928,647,546
Faster R-CNN	49,436,937	98,009,481	195,154,569	389,444,745	778,025,097

split into four smaller mini-batches and they are processed by each smartSSD in parallel. After reaching the DNN layer with a given partition index, intermediate fmaps from four smartSSDs are moved to the NPU together. Then, the remaining layers are processed by the NPU and finally the loss per batch is computed sequentially on the CPU. The backward pass is activated to perform the gradient propagation and the weight update. Note that the gradient maps are distributed back to the corresponding smartSSDs. The process of forward and backward passes for a given mini-batch is simulated using DPT-sim for the performance estimation.

4 Analysis of Deep Partitioned Training

4.1 Case Study I: Vision-Related DNN Models

First, we evaluate the efficiency of deep partitioned training on the vision-related tasks, image classification [11] and object detection [12]. The training consists of three parts: i) forward prop, ii) backward prop, and iii) weight update. Feature maps are reduced in advance for computing ΔW. The baseline is running the entire computations on the NPU with a relatively larger systolic array. Note that the classifier, i.e., softmax layer and computing loss, is processed at the host CPU. To set up the best possible baseline model, we allow prefetching the next mini-batch to NPU DRAM for higher performance. In Table 3, the latency to complete the training on a mini-batch at various sizes is provided for ResNet-50 (54 layers) and Faster R-CNN (46 layers).

For the deep partitioned training, we utilize up to four smartSSDs and a host-side NPU. Our simulation results are summarized in Table 4. As we increase the number of smartSSDs and/or the size of a mini-batch, a significant performance improvement is observed thanks to batch-wise data parallelism and increased PCIe bandwidth. The number inside the parenthesis denotes the optimal partition index by the sweep test showing the minimum latency. For instance, '31.0 percent (32)' means that the performance improves by 31.0 percent over the baseline when we partition the model at layer 32 in Faster R-CNN. The benefit of utilizing smartSSDs for test case ③ is smaller compared to other cases as the NPU has 2× larger systolic array than four smartSSDs. We can see that the optimal partition index is governed by the ratio of computing power of NPU to that of smartSSDs.

In Fig. 3, the latency breakdowns of the proposed deep partitioned training on ResNet-50 at three different mini-batch sizes are provided (B=32, 64, and 128). As we pipeline the forward propagation for each training instance, most of the computation time in the storage is overlapped with the compute time of the NPU (refer to 'Hide' for the actual training time). Most of the computation time

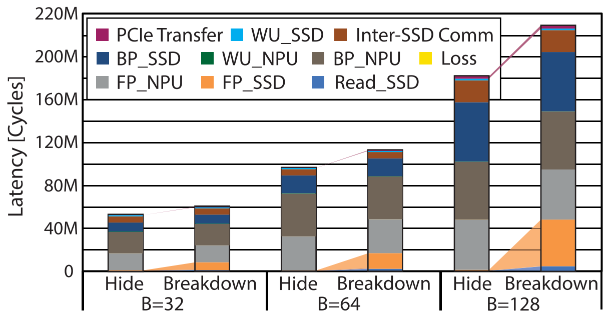

Fig. 3. The latency breakdowns of deep partitioned training on ResNet-50 with four smartSSDs (test case 2) at various mini-batch sizes.

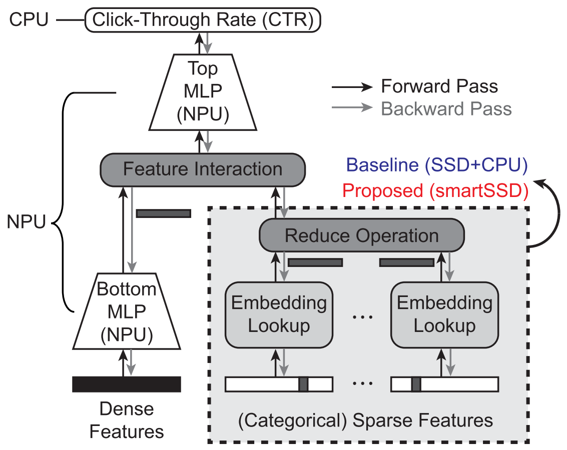

Fig. 4. The baseline and the proposed training system for the recommendation system (DLRM).

is consumed by systolic arrays in the NPU and smartSSDs. The other dominant part of computation is the time for inter-SSD communications to gather, reduce and broadcast weight gradients, i.e., $T^{0:i-1}_{\text{interSSD}}$ in Eq. (3).

4.2 Case Study II: Recommendation System

As another benchmark, we evaluate the deep partitioned training on a DLRM model for recommendation system [7]. Note that this is a preliminary result which will be extensively studied in our future work. We compare two systems, adding embedding vectors (**emb**) i) on a host CPU (baseline) and ii) within a smartSSD (proposed), and process the 'bottom & top MLPs' on an NPU (Fig. 4). The prediction layer computing the click through rate (CTR) is processed at the CPU. For the experiment, we use our FPGA prototype with NAND flash chips that stores embedding tables. For both the baseline and the proposed system, embedding vectors are read from the storage. The parameters used in our DLRM experiments are summarized in Table 5. Considering the baseline, we need to move 26 vectors per request to complete the `reduce` operation prior to computing the top MLP. Instead, we can perform the

TABLE 4
The Performance Improvement by Utilizing Deep Partitioned Training on Two Representative CNN Models (ResNet-50 and Faster R-CNN)

CNN Model	Array Size (SSD, NPU)	Parameters: (# of smartSSDs, Mini-batch Size)									
		(2, 32)	(4, 32)	(2, 64)	(4, 64)	(2, 128)	(4, 128)	(2, 256)	(4, 256)	(2, 512)	(4, 512)
ResNet-50	① (32x16, 48x48)	5.4% (11)	15.9% (25)	6.5% (11)	20.5% (31)	7.1% (11)	23.9% (31)	7.4% (11)	25.6% (31)	7.5% (11)	26.5% (31)
	② (64x32, 96x96)	3.9% (12)	10.8% (25)	5.1% (12)	17.4% (25)	7.0% (24)	21.9% (35)	8.0% (24)	26.9% (35)	8.5% (24)	29.4% (35)
	③ (64x32, 128x128)	2.4% (2)	6.8% (10)	3.5% (6)	9.5% (12)	4.2% (9)	11.0% (11)	4.6% (9)	11.8% (11)	4.9% (9)	12.2% (11)
Faster R-CNN	① (32x16, 48x48)	6.0% (11)	17.7% (25)	7.2% (11)	21.9% (27)	7.9% (11)	24.9% (29)	8.2% (11)	26.4% (29)	8.3% (11)	27.2% (29)
	② (64x32, 96x96)	4.6% (12)	12.9% (25)	6.0% (12)	20.8% (25)	7.8% (22)	25.2% (29)	8.8% (22)	28.8% (29)	9.3% (22)	31.0% (32)
	③ (64x32, 128x128)	2.9% (2)	8.4% (10)	4.3% (6)	11.7% (12)	5.2% (9)	13.5% (11)	5.7% (9)	14.5% (11)	6.0% (9)	15.0% (11)

TABLE 5
The Parameters Used in the Experiment for Recommendation System

Mini-batch Size	PCIe BW	# of Vectors/Request	Word Size
64	4GB/s	26	32bit

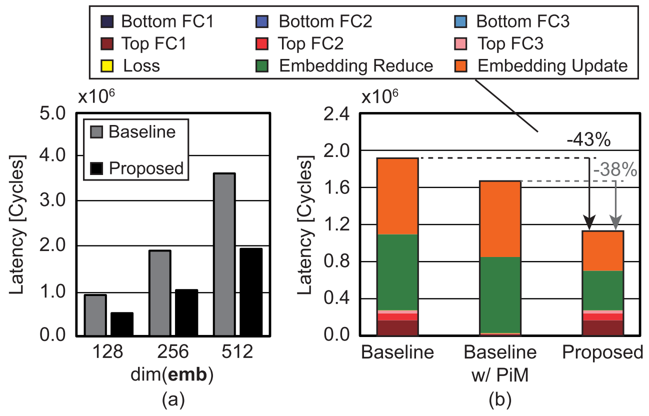

Fig. 5. The experimental results on a DLRM model. (a) Total runtime comparison between the baseline and the proposed system at various embedding vector sizes. (b) Latency breakdown when dim(**emb**)=256.

`reduce` operation within the smartSSD to minimize the data that move across the PCIe interface. The computation time of the bottom MLP at the NPU is shorter than that of reading embedding vectors, thus hidden in the total latency.

As shown in Fig. 5a, we can reduce the total runtime per mini-batch over the baseline by 40.6, 43.3 and 44.7 percent with vector dimension of 128, 256 and 512. The `reduce` operation (sum) in the proposed system is actually performed on an ARM processor on the Xilinx Zynq board, while it is performed on an x86 processor (host CPU) for the baseline. As the bottom and the top MLP are processed on the NPU for both cases, the same latency is consumed on the MLP computations (extracted from DPT-sim). Thus, the reduction in total runtime comes from the `reduce` operation, updating embedding tables, and its PCIe transfer time. The performance of the baseline can be improved by using Processing-in-Memory (PiM) instead of the NPU as proposed in [13]. By deploying a PiM device, the latency of a forward/backward propagation in the top MLP is minimized (Fig. 5b). As the dominant part of training the DLRM model is in reducing and updating the embedding vectors, the proposed system still reduces the training time by 37.9 percent compared to the baseline with the PiM device.

5 Conclusion

In this work, we proposed a system architecture, named deep partitioned training, that splits a DNN model and training data across multiple storage devices, an NPU and a CPU. The proposed system utilizes a hybrid of data parallelism and layer partitioning schemes. For two representative DNN workloads, vision-related tasks and a recommendation system, we were able to efficiently reduce the training time by up to 31.0 and 44.7 percent, respectively. Considering the sparsity in input features during the training or more flexible assignment, allowing more than a single partition index, of DNN layers to either smartSSD or NPU at runtime remains as our future work.

Acknowledgments

This work was supported by the Samsung Research Funding Incubation Center of Samsung Electronics through Project SRFC-IT1902-03.

References

[1] V. Sze, Y. Chen, T. Yang, and J. S. Emer, “Efficient processing of deep neural networks: A tutorial and survey,” *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.
[2] Cloud TPU. Accessed: Nov. 26, 2020. [Online]. Available: https://cloud.google.com/
[3] NVIDIA A100 tensor core GPU. Accessed: Nov. 26, 2020. [Online]. Available: https://www.nvidia.com/en-us/data-center/a100/
[4] M. Kim, J. Kung, and S. Lee, “Towards scalable analytics with inference-enabled solid-state drives,” *IEEE Comput. Archit. Lett.*, vol. 19, no. 1, pp. 13–17, Jan.–Jun. 2020.
[5] J. H. Lee, H. Zhang, V. Lagrange, P. Krishnamoorthy, X. Zhao, and Y. S. Ki, “SmartSSD: FPGA accelerated near-storage data analytics on SSD,” *IEEE Comput. Archit. Lett.*, vol. 19, no. 2, pp. 110–113, July.–Dec. 2020.
[6] High bandwidth memory (HBM) DRAM, JESD235D, JEDEC, Arlington, VA, USA, 2020.
[7] A. Eisenman *et al.*, “Bandana: Using non-volatile memory for storing deep learning models,” 2018, *arXiv: 1811.05922*.
[8] U. Gupta *et al.*, “The architectural implications of Facebook’s DNN-based personalized recommendation,” in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2020, pp. 488–501.
[9] M. Drumond *et al.*, “Training DNNs with hybrid block floating point,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2018, pp. 451–461.
[10] Xilinx Alveo U200 and U250 data center accelerator cards data sheet. Accessed: Apr. 8, 2021. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds962-u200-u250.pdf
[11] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.
[12] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards real-time object detection with region proposal networks,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2015, pp. 91–99.
[13] M. He *et al.*, “Newton: A DRAM-maker’s accelerator-in-memory (AiM) architecture for machine learning,” in *Proc. IEEE/ACM Int. Symp. Microarchit.*, 2020, pp. 372–384.