DLUX: a LUT-based Near-Bank Accelerator for Data Center Deep Learning Training Workloads

Peng Gu, Xinfeng Xie, *Member, IEEE,* Shuangchen Li, Dimin Niu, Hongzhong Zheng, *Member, IEEE,* Krishna T. Malladi, *Member, IEEE,* and Yuan Xie, *Fellow, IEEE,*

***Abstract*—The frequent data movement between the processor and the memory has become a severe performance bottleneck for deep neural network (DNN) training workloads in data centers. To solve this off-chip memory access challenge, the 3D stacking processing-in-memory (3D-PIM) architecture provides a viable solution. However, existing 3D-PIM designs for DNN training suffer from the limited memory bandwidth in the base logic die. To overcome this obstacle, integrating the DNN related logic near each memory bank becomes a promising yet challenging solution, since naively implementing the floating-point (FP) unit and the cache in the memory die incurs large area overhead. To address these problems, we propose DLUX, a high performance and energy-efficient 3D-PIM accelerator for DNN training using the near-bank architecture. From the hardware perspective, to support the FP multiplier with low area overhead, an in-DRAM lookup table (LUT) mechanism is invented. Then, we propose to use a small scratchpad buffer together with a lightweight transformation engine to exploit the locality and enable flexible data layout without the expensive cache. From the software aspect, we split the mapping/scheduling tasks during DNN training into intra-layer and inter-layer phases. During the intra-layer phase, to maximize data reuse in the LUT buffer and the scratchpad buffer, achieve high concurrency, and reduce data movement among banks, a 3D-PIM customized loop tiling technique is adopted. During the inter-layer phase, efficient techniques are invented to ensure the input-output data layout consistency and realize the forward-backward layout transposition. Experiment results show that DLUX can reduce FP32 multiplier area overhead by 60% against the direct implementation. Compared with a Tesla V100 GPU, end-to-end evaluations show that DLUX can provide on average $6.3\times$ speedup and $42\times$ energy efficiency improvement.**

***Index Terms*—Near Data Processing, Deep Neural Network, 3D Stacking Circuit, DRAM**

I. Introduction

DEEP Neural Network (DNN) is playing an increasingly important role in modern data center workloads [1], [2], and these DNN tasks require frequent and time-consuming training. Developing fast and energy-efficient accelerators for these training tasks is necessary yet challenging, since they need both high memory capacity and bandwidth. First, the large capacity requirement comes from not only the training weights but also the intermediate data and gradients, since they need to be stored and accessed during every iteration of the training process. For example, with a relatively small batchsize of 4, DeepSpeech2 requires $6GB$ memory, over 75% of which belongs to intermediate data and gradients [3]. Second, many of these DNN training workloads are bandwidth-intensive, since they consist of operations with medium to low arithmetic density. Therefore, with the trend of larger model size and growing dataset, limited off-chip bandwidth poses a severe performance challenge on compute-centric accelerators. To show the bandwidth-bound behavior of these workloads, we conduct a case study on a Tesla V100 GPU [4] using 6 representative DNN training data center workloads [5]–[7], including tasks of machine translation, speech recognition, recommendation, text summarization, encoder, and compression. We observe that the profiled workloads spend a significant amount of time ($40\% \sim 100\%$) executing bandwidth-bound operations on GPU, according to the roofline model [8]. Further details about the profiling results are shown in Sec.II.

To meet the bandwidth demands for DNN training, the 3D stacking processing-in-memory (3D-PIM) architecture [9]–[13] has shown more promising potential than the current compute-centric architecture [4], [14], [15]. However, existing 3D-PIM solutions for DNN training cannot deliver competitive performance, due to the limited memory bandwidth available on the base logic die. For example, one heterogeneous 3D-PIM accelerator [9] only provides $\sim 10\%$ speedup compared with GTX 1080Ti GPU, since its peak memory bandwidth (480GB/s adopting HMC 2.0 [16] configuration) only improves 50% compared with GDDR5X (320GB/s). The performance gap roots in the bandwidth restriction that memory accesses from the base die still need to use the limited number of Through-Silicon-Vias (TSVs) (maximum 1024 TSVs per cube). Adding the TSV number to increase memory bandwidth will suffer from both large area overhead and reduced energy-efficiency, since TSVs already occupy $\sim 20\%$ area in the current 3D stacking memory [17]. Even worse, a previous work [18] shows that the in-cube data movement consumes more than 60% of the total access energy.

To abandon these restrictions, a promising approach is to closely integrate the DNN training logic with each memory bank. This near-bank architecture [19] enables computation resources and memory bandwidth to scale-up synergistically with increasing 3D stacking layers, significantly reduces the in-cube data movement, and allows the DNN training logic to fully utilize bank-level bandwidth without changing the

Manuscript received XX XX, XXXX; revised XX XX, XXXX; accepted XX, XX, XXXX. Date of publication XX, XX, XXXX. This work was supported in part by the NSF 1719160, 1725447, and 1730309, and in part by the Samsung Semiconductor, Inc.

P. Gu, X. Xie, and Y. Xie are with the Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA, 93106 USA e-mail: (yuanxie@ucsb.edu).

S. Li, D. Niu, and H. Zheng are with Alibaba DAMO Academy, Sunnyvale, CA, 94085, USA e-mail: (hongzhong.zheng@alibaba-inc.com). D. Niu and H. Zheng were previously with Samsung Semiconductor, San Jose, 95134 USA.

K. Malladi is with Samsung Semiconductor, San Jose, 95134 USA e-mail:(k.tej@samsung.com).

DRAM timing. Nevertheless, the low-area overhead hardware design in memory process [20] and the associated software mapping and scheduling techniques remain key challenges to be addressed. From the hardware perspective, a lightweight floating point (FP) unit design is required, and the expensive cache needs to be replaced without performance penalty. From the software perspective, efficient algorithms need to be invented to increase the utilization of the proposed FP unit, and novel mapping and scheduling schemes are required to hide the latency and allow flexible data layouts.

The goal of this work is to propose a low hardware overhead near-bank 3D-PIM architecture, DLUX, for DNN training acceleration providing high FP performance. We propose both the hardware design and the software mapping/scheduling techniques to solve the challenges mentioned above. From the hardware perspective, to support efficient FP arithmetic with low area overhead, we propose to use an in-DRAM lookup table (LUT), which trades memory capacity for computing performance. In order to reduce the lookup overhead, we adapt a hierarchical LUT structure, where the full LUT table is stored in a DRAM bank but the LUT entries are cached in a small buffer in each bank's peripheral. To reduce the cache overhead, we use a simple scratchpad memory buffer for data reuse, and a transformation unit to assist flexible data layouts. From the software aspect, we solve the mapping and scheduling problems from both the intra-layer phase and the inter-layer phase. During the intra-layer phase, to reduce LUT fetching overhead, LUT entries in the LUT buffer are reused a number of times before reloading. Then, to achieve high concurrency and low data movement among banks, the input data parallelism and the intermediate result stationary scheme are used. During the inter-layer phase, transparent and low overhead techniques are invented to ensure input-output layout consistency and forward-backward layout transpose.

Our specific contributions are listed as follows.

- We propose a near-bank architecture, DLUX, for DNN training acceleration with both high performance and low hardware overhead.
- We demonstrate the DLUX design, with the highlight of the in-DRAM hierarchical LUT for high-performance FP computing, efficient communication using shared data bus and lightweight support for data transformation.
- We present the DLUX software design. The intra-layer mapping/scheduling improves utilization, concurrency while minimizing data movement, and the inter-layer data transformation ensures layout consistency in dataflow processing.
- We evaluate DLUX and compare it with the Tesla V100 GPU. The results shows DLUX provides on average $6.3\times$ end-to-end speedup and $42\times$ energy-efficiency improvement on representative data center training workloads.

II. Background

a) The DNN training data flow: The DNN training process is very memory demanding, and can be abstracted as a data flow graph shown in Fig.1, represented as (a) inter-layer and (b) intra-layer data flow. For the inter-layer data flow, each iteration will feed the input data with a certain batchsize into the network, propagate the intermediate results of each

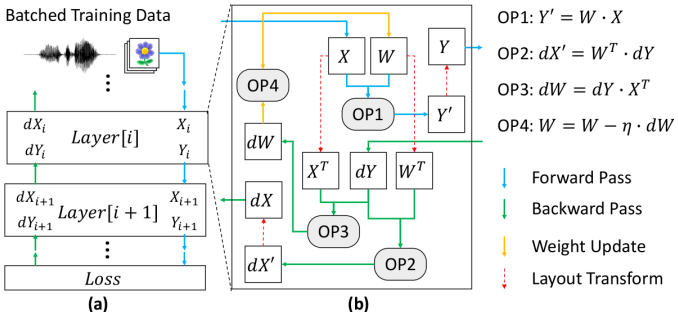


Fig. 1. **Typical data flow and operations in DNN training. (a) Inter-layer data flow. (b) Intra-layer data flow for a fully-connected layer.**

Application	Type	BatchSize	DataSet	Notation
Recommendation	MLP	1000	MovieLens	Recommendation [21]
Speech Recognition	RNN	32	TIMIT	DeepSpeech [22]
Translation	Attention	4096	WMT	Transformer [23]
Text Summarization	RNN	4	Gigaword	TextSum [24]
Sentence Encoder	RNN	128	BookCorpus	SkipThoughts [25]
Compression	RNN	4	Kodak	EntropyCoder [26]

TABLE I
Benchmark setting.

layer (Y_i, Y_{i+1}), calculate the loss at the final layer, and then back-propagate the gradient (dX_i, dX_{i+1}) through all layers. For the intra-layer data flow, the backward pass $(OP2, 3, 4)$ requires more computation than the forward pass $(OP1)$. Even worse, it needs to store the intermediate results (X), which significantly increases the memory overhead. Also, the layout transformation is required, since the layouts of the input and output tensor need to be consistent (Y_i and X_{i+1}), and the backward pass requires transposed format (X^T, W^T).

b) A case study for DNN training workloads: We conduct a detailed profiling of representative workloads in Table.I. We use tensorflow [27] to record the computation instructions, the memory access count, and the execution time of every operations in the benchmarks. First, to analyze the memory-bound behavior, we accumulate the execution time of operations with arithmetic density lower than GPU performance/bandwidth ratio $(17.5FLOP/Byte)$ as shown on the left of Fig.2. Second, to understand the performance bottleneck for each class of operations, we derive the trend of accumulated percentage of time as arithmetic density increases for each class on the right of Fig.2. To plot a data point with arithmetic density $X_{density}$ in the trend line, we first sum the execution time of all operations of that class whose arithmetic density is lower than $X_{density}$. Then, we divide the added time by the total execution time of that application to calculate the accumulated percentage of time for that data point. For the results, we first find that these workloads spend a large amount of time $(40\% \sim 100\%)$ executing bandwidth-bound operations on GPU. More detailed analysis shows that except for General Matrix Multiplication $(GEMM)$, other kernels exhibit memory bound behaviors (plateau occurs left of the GPU Perf/BW ratio according to the arithmetic density distribution). For $GEMM$ kernels, significant portion of time also shows memory bound behaviours. We also discover that most of the training execution time $(> 95\%)$ is dominated by 4 categories of kernels: $GEMM$, $Elementwise$ (e.g., elementwise addition), $Reduction$ (e.g., tensor reduction along one axis), and $DataManipulation$ (e.g., tensor concatenation).

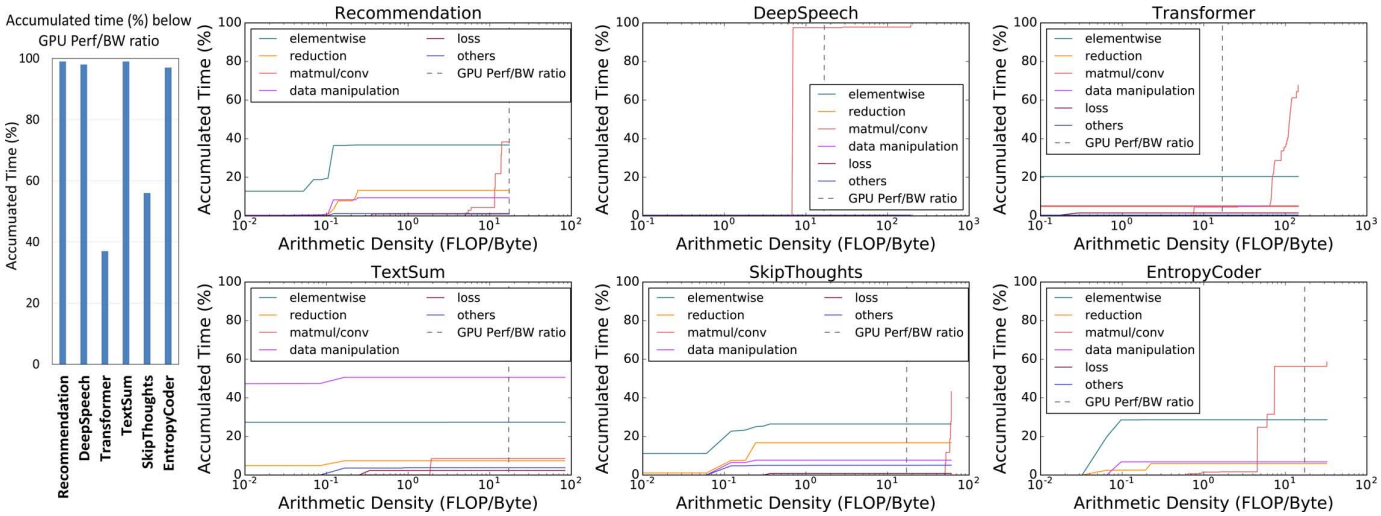


Fig. 2. **Left: the percentage of time for operations that have lower arithmetic density than GPU performance/bandwidth ratio. Right: accumulated percentage of time distribution according to arithmetic density for different classes of operations.**

c) 3D Stacking Memory: Though with different interfaces, the architectures in various 3D memory standards are similar. One memory *cube* contains a base logic die and multiple stacking DRAM dies. The logic die carries the control circuit and the physical layer (PHY). In one DRAM die, the *subarray* is the minimal DRAM cell array with the dedicated local decoder and the sense amplifier. A group of subarrays that share the global data lines form a *bank*. A bank has a *row buffer* to support data burst and provides spatial locality. A group of banks in the same DRAM die forms a *bank group* and are linked by shared data buses. Several bank groups in different DRAM dies are connected with Through Silicon Vias (TSVs) to controllers in the base logic die, forming a vertical *vault*. Note that DLUX is based on general 3D memory and is adoptable for both High Bandwidth Memory (HBM) [28] and Hybrid Memory Cube (HMC) [29] architecture.

III. Challenges for DLUX design

a) Area Overhead in the DRAM die: The 3D-PIM's performance challenge is aggravated by the requirement for area-expensive FP units (1.6× larger than an integer unit for 32-bit Multiply–Accumulate (MAC) in 45nm [30], [31]). As mentioned in Sec.I, to alleviate the bandwidth bottleneck of the base die logic integration, a near-bank design places logic inside the DRAM die. However, building complex logic inside the DRAM die results in significant area overhead due to specialized DRAM technology (as high as 80% [20] overhead). In Sec.IV, DLUX overcomes this area-constrained performance challenge by using a configurable DRAM-based LUT as the extra computing resource.

b) Slow DRAM-based LUT: Using the DRAM as a LUT for computing is non-trivial for achieving high performance. Different from sub-ns fast SRAM-based LUT, DRAM row access latency is tRC (e.g., 48ns [28]). With every DRAM bank serving as a LUT, the extra performance gain from LUTs of a typical 8-die memory cube is marginal 0.01TFLOPS. In Sec.IV-B, DLUX overcomes this slow LUT challenge by introducing a buffer for the LUT and the software scheduling method to improve its data reuse.

c) Data Movement: A compute-centric accelerator employs customized on-chip interconnect network for efficient inter-node data communication. However, PIM is based on the 3D memory, which only has shared buses for interconnecting. Although the shared data buses are wide thanks to 3D integration, operations requiring inter-bank communication will incur long latency due to shared bus data congestion. In Sec.IV-C, DLUX leverages this shared data bus structure for bankgroup-local and vault-local data movement.

d) Requirement for Layout Transformation: As shown in Fig.1 (b), since the input-output layout needs to be consistent between DNN layers, and the forward-backward layout needs to be properly transposed, potential data movement introduced by the layout transformation is time and energy consuming. Two unique features of PIM make this problem more challenging. First, unlike unified memory abstraction for compute-centric architecture, PIM adopts a distributed memory model, where data are partitioned so that the majority of the data feeding for compute units come from local banks. Improper data partitioning will introduce unnecessary inter-bank data movement, such as all-to-all broadcasting during data transpose. Second, PIM lacks a complex cache design which can be effective in hiding memory latency and allowing flexible memory access patterns. If data placement in the local bank results in poor spatial locality in row buffer, frequent activations of different rows will result in inefficient intra-bank data movement. DLUX first adds light weight logic in Sec.IV-D, and then uses layout transformation, locality-aware mapping, and PIM-friendly partial transpose format in Sec.V-B to address this challenge.

IV. DLUX Architecture

A. Overall Architecture

The DLUX design highlights the near-bank architecture with efficient supports for the LUT-based FP operations, the hierarchical shared data buses for efficient data communication, and the lightweight hardware for layout transformation. To fulfill these features, DLUX adopts a scalable architecture

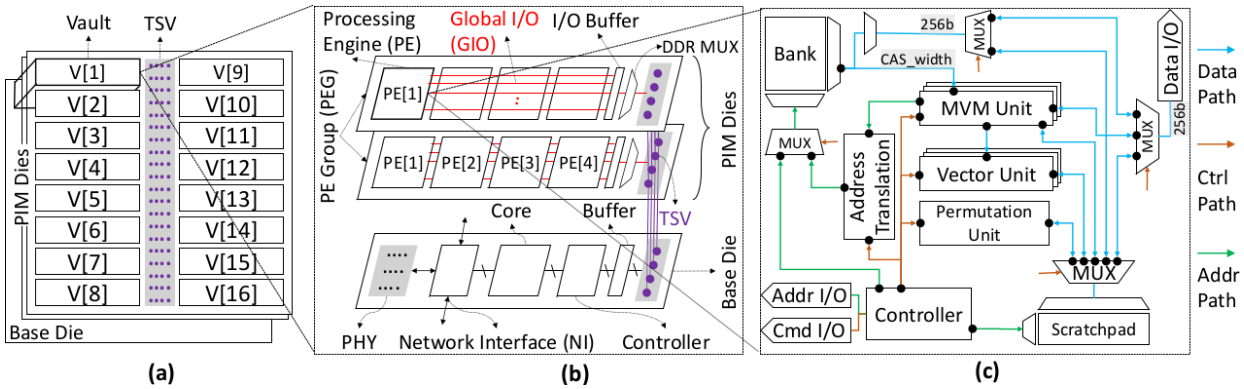


Fig. 3. **DLUX architecture overview: (a) a cube, (b) a vault, and (c) a processing engine.**

composed of cubes, vaults, processing engine groups (PEGs), and processing engines (PEs), as shown in Fig.3. A DLUX cube consists of 8 processing-in-memory (PIM) memory dies on the top of a base logic die, connected by TSVs. Each cube is further divided into 16 vertical vaults, each of which owns $64b$ TSVs spreading across 8 layers, as shown in Fig.3(a). Fig.3(b) zooms into a vault. The base logic die of a vault contains a network interface (NI) for inter-vaults and inter-cube data communication, a simple programmable ARM core to provide friendly user interface and issue DLUX instructions, a hardwired controller, and a buffer. For the other PIM dies in the vault, each die contains one PEG, which executes kernels and communicates with other PEGs in the same vault through the shared TSVs. A PEG contains 4 PEs. They are all connected to a $256b$ global I/O (GIO) bus.

Fig.3(c) further zooms into a PE. The PE employs the near-bank architecture, in which computing logics are in the bank peripheral region, without any modification to the memory array. Such architecture fully exploits the high bank-level bandwidth, while remaining manufacture friendly [32]–[34].

In the bank peripheral region, we design computing units, control units, data paths, and a scratchpad memory. The computing units include matrix-vector-multiply (MVM) units, vector units, and a permutation unit. The control units include a controller and an address translator. The data paths include the links connecting (1) the computing units and the scratchpad memory; (2) the scratchpad memory and the bank/GIO; (3) the bank and the MVM units/GIO. We double the data bus CAS_width and the bank-level row buffer to reduce communication overhead between the bank and other units. The scratchpad memory supports both the computation units and the communication and layout transformation operations.

B. Computation Support

a) LUT-based Multiplication: The key idea is to increase the memory-side FP performance with a limited area budget by leveraging part of the DRAM memory for computing, i.e., using DRAM as a LUT for the FP arithmetic implementation. However, there are two challenges: (1) the exponential memory capacity demand, and (2) the DRAM's long latency.

To overcome the large LUT capacity challenge, instead of looking up the whole FP32-MAC operation directly (2^{98}Byte memory), we only lookup the most area consuming part of the arithmetic. We find that a FP32-MUL's area is $1.8\times$ larger than that of a FP32-ADD [35]. Inside the FP32-MUL, the significand MUL contributes $\sim 87\%$ of the total area. Therefore, we only use LUT for the significand MUL, while implementing other parts with digital logic circuits. Furthermore, even in the 23b significand MUL, we only conduct LUT for the partial product of a $12b\times 4b$ MUL, while adding partial product adders in digital circuits. To overcome DRAM's long latency challenge, we propose a hierarchical and buffered LUT architecture. We first lookup the first operand from the DRAM bank, and store the partial LUT results in the faster SRAM based buffers. Then, we lookup the second operand from the faster buffers. DLUX scheduling will optimize data reuse from the SRAM buffer (i.e., the first operand stays unchanged). Note that the SRAM buffer only stores all possible results given a known first operand, so it is much more efficient than the all-logic FP multiplier.

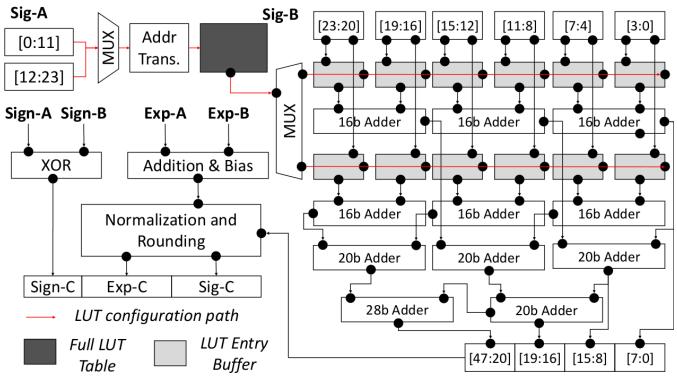


Fig. 4. **Hierarchical lookup table based FP32 multiplier design.**

Fig.4 shows the detailed design. As illustrated, we only use the LUT for partial product results in the significand MUL, i.e., Sig-A$\times$Sig-B. Others including the addition of partial product in the significand MUL, the exponent addition, and the normalization etc. are implemented with digital logic circuits. All the computations shown in Fig.4 happen in the DRAM layer in each PE, so there will not be inter-layer data movement. The hierarchical LUT architecture has the full LUT table stored in the dense but slow DRAM bank, while having the lightweight but fast SRAM buffers allocated outside of each bank. The first operand (Sig-A) is used as the row address of the DRAM bank (after a simple pre-loaded address translation) to fetch one entry to the SRAM LUT buffer, and the second operand (Sig-B) is used as the column address of the LUT buffer to get a partial product result. These partial results are then summed up by the digital adders.

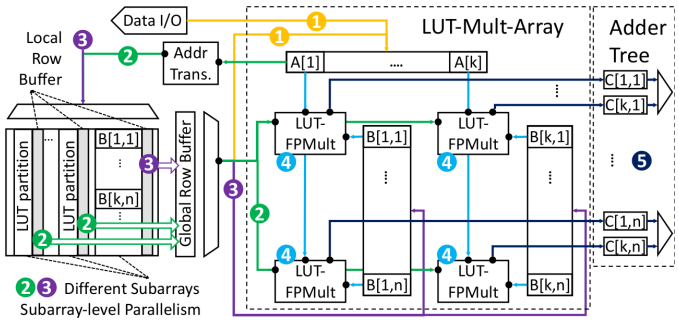


Fig. 5. **Matrix-vector-multiply (MVM) unit design.**

b) Matrix-Vector-Multiply (MVM): We build the MVM unit using the lookup table based floating point multiplier (LUT-FPMult), while exploiting the faster LUT buffer, i.e., maximizing the reuse of the first operand. The design and the working flow is shown in Fig.5. We denote the input vector as $A[1:k]$ and the input matrix as $B[1:k, 1:n]$. We assign A as the first operand to lookup the full LUT table inside DRAM (like Sig-A in Fig.4) and B as the second operand to lookup the LUT buffer. The MVM working flow is divided into five steps: ❶ $A[k:1]$ is fetched from the local bank or the broadcast data from the GIO, to the LUT index buffer. ❷ Values from the LUT index buffer are translated to fetch the corresponding entries of the full lookup table in the DRAM bank. The results are stored in the LUT buffers in each LUT-FPMult. ❸ B matrix is fetched from local bank to the input vector buffers. ❹ Each $B[i,j]$ decodes the LUT buffer to complete the FP32-MUL computing. ❺ The results will be summed ($\sum_{i=1}^{k} A[i] \cdot B[i,j]$) by the adder tree. Such working flow maximizes the LUT buffer locality. Each LUT buffer result is reused n times, since each $A[i]$ will be multiplied with all values in vector $B[i,:]$. We further optimize the performance by hiding latency during data fetching. We place the full LUT table and B matrix data into different subarrays of the same bank, and employ subarray-level parallelism [36] to overlap the time of ❷ and ❸.

c) Vectorized Computing and SFU: The vector unit contains an array of FP32-ADD, FP32-MUL, and other simple logical units implemented by digital logic circuits. It has two functions. First, it accumulates and updates the partial results stored in the scratchpad memory with new output values from the MVM units. Second, it performs simple elementwise operations, including addition and logical operations (e.g., min) for DNN layers such as ReLu. To support more complex non-linear functions, we add one SFU (Special Function Unit) per PE. This unit enables frequently used non-linear activation functions in training, such as $sigmoid$ and $tanh$.

d) Permutation Unit: The permutation unit consists of multiple cyclic-shift FIFOs. The input is fed column-wise to each FIFO, and the output is fetched row-wise where one FIFO contributes one element in the row. The data transposition is scheduled after the completion of a layer and is overlapped with the computation of the next layer.

C. Communication Support

An efficient communication scheme is critical for DLUX's performance and energy efficiency. From the hardware perspective, DLUX adopts a distributed memory model owing to its PIM nature, where accesses to non-local addresses will introduce expensive inter-PE data movement. From the application perspective, many important DNN kernels incur significant communication traffic due to the requirement for the scattering (e.g., data-sharing) and the gathering (e.g., reduction) computation patterns.

To meet these requirements, we propose to fully exploit and reuse the already existing hierarchically shared bus architecture in 3D memory for data movement. First, 4 PEs in the same PEG share $256b$ GIO, which is used for the inter-PE communication. Second, 8 PEGs in the same vault share $64b$ TSVs, which are used for the inter-PEG communication. Third, 16 vaults communicate through the on-chip network, which is used for the inter-vault communication. Last, each cube has a full-duplex serial link with peak bandwidth $80GB/s$ [16], which is used for the inter-cube communication.

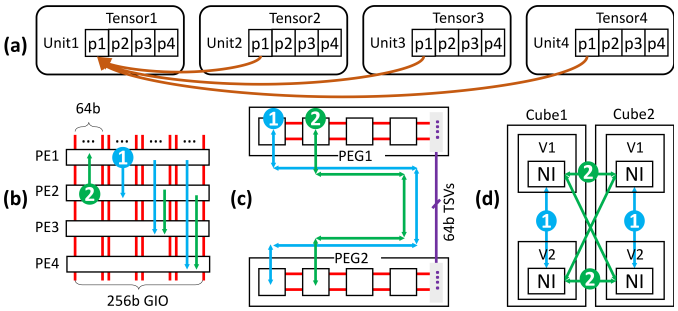


Fig. 6. **Hierarchical interconnects to perform reduction operations: (a) reduction pattern, (b) inter-PE reduction, (c) inter-PEG reduction, (d) inter-vault and inter-cube reduction.**

Fig.6 illustrates an example of using the hierarchical interconnects to perform a tensor reduction operation. Fig.6 (a) shows the reduction pattern among four tensors, where each hardware unit contains one tensor. For load balancing, each tensor is partitioned into four parts with equal size marked as $p1$ to $p4$, and each unit is responsible for reduction of a single part among the four tensors. For example, $Unit1$ will collect all $p1$ parts from the other three units and perform local accumulation. Fig.6 (b)-(d) shows the cases when the reduction is at PE-level, PEG-level, and vault/cube-level. All cases need four passes for the 4-tensor reduction. The blue and the green arrows show the examples of the first two passes. In Fig.6 (b), the PE-level reduction, each PE sends corresponding data to other PEs through the shared $256b$ GIO, one after another. In Fig.6 (c), the inter-PEG reduction, the shared TSV bus is used for data movement. In Fig.6 (d), since data paths involved in high-level nodes (e.g., inter-cube) always have lower bandwidth and higher data movement costs than those in the low-level nodes (e.g., inter-vault), low-level data communication is always granted with a higher priority.

D. Layout Transformation Support

Customized memory layout is important when attempting to increase PE's performance and utilization. Lacking complex cache hierarchy, DLUX's performance heavily relies on the memory coalescing and spatial locality in the bank row buffer. To guarantee the layout for each data structure during every

phase of the data flow, data transformations are needed, which is efficiently supported by permutation unit (Sec.IV-B) in each PE. First, the input data is read to scratchpad memory, and streamed into the permutation unit using a row major order. Second, the column reorder operation is performed by selecting a certain cyclic first-in-first-out (FIFO) queue to write each input row. For matrix transpose, the cyclic FIFO is selected from left to right. Third, the row reorder operation is performed by cyclically moving the FIFO, so that elements in the same row can be shuffled. Lastly, all FIFOs will output its elements in sequence, and each time a permuted row is acquired and written to scratchpad memory.

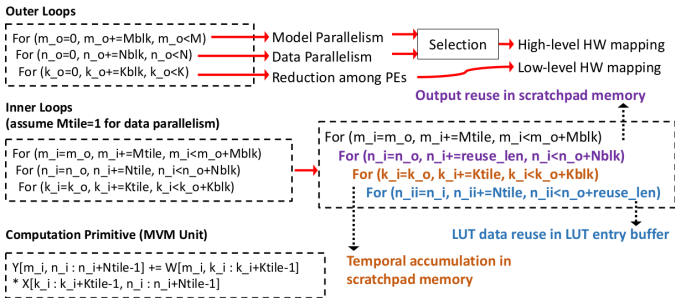


Fig. 7. **GEMM mapping scheme loop formulation.**

V. DLUX Mapping and Scheduling

A. Intra-layer Partitioning and Scheduling

We first focus on the most important general matrix multiplication (GEMM) kernel, which is followed by the description of supporting other kernels, with a detailed example of the batch normalization (BN) kernel.

1) General Matrix Multiplication (GEMM): The GEMM kernel is the most important kernel, because it is used to compute many major DNN layers, such as the fully-connected layer, the recurrent layer, and the convolutional layer [37], and hence is the time-dominating kernel for some DNN tasks ($40\% - 90\%$ as shown in Fig.2). Comprehensive optimizations on mapping and scheduling are introduced as follows.

a) Problem Formulation with Nested Loop: GEMM can be formulated as a three-level nested loop. We denote the input matrix as X of size $K \cdot N$, the weight matrix as W of size $M \cdot K$, and the output matrix as Y of size $M \cdot N$. We apply two-level loop tiling to the original loop nest, as shown on the left side of Fig.7 (a). For the first tiling, we denote it as the outer loops, which partition the M, N, and K dimensions into $Mblk$, $Nblk$, and $Kblk$, respectively. We also denote tiles from these outer loops as *blocks*, to distinguish from these in the second tiling. For the second tiling, we denote it as the inner loops, and it further partitions the $Mblk$, $Nblk$, and $Kblk$ dimensions into $Mtile$, $Ntile$, and $Ktile$, respectively. The loop body is the computation primitive calculating the matrix multiplication of $W_{Mtile \cdot Ktile} \times X_{Ktile \cdot Ntile}$.

b) Outer Loop Optimization: Spatial Partition to Increase Concurrency: We use the outer loop tiling for spatial partition, i.e., we partition the input (X) or weight (W) matrix into blocks and distribute them to different PEs.

First, we explain how to partition, i.e., to determine the value of $Mblk$, $Nblk$, and $Kblk$. It is a 3-step decision. Step-1: we decide to partition either M or N, i.e., setting either $Mblk = M$ or $Nblk = N$. We do not partition both because we want to minimize the data movement. Allowing only one partition means that only one input matrix (either X or W) is required to be broadcast across all spatial partitions. The other input matrix with the spatial parallelism can be stationary in their local banks during the entire process of training, so that the data movement is minimized. We choose the larger one from M and N to partition in order to maximize spatial concurrency. Step-2: we determine the value of $Nblk$ (or $Mblk$ if selecting M to partition in Step-1). This partition is not straightforward. On the one hand, we want $Nblk$ (or $Mblk$) as small as possible, in order to maximize spatial parallelism. On the other hand, we want the $Nblk$ to be large enough to fully utilized the hardware, e.g., scratchpad memory, in a PE , so that each block runs faster. The overall performance is then an overall consideration of both the parallelism and the single block performance. We solve the problem by building an analytical model with the objective of maximizing the overall performance. Step-3: we determine the K partition, i.e., the value of $Kblk$. We find the maximal $Kblk$ that ensures every PE has at least one block to work on.

Second, we explain the mapping. The partition of M or N is mapped to the cube and then to the vault, which is referred to as high-level hardware mapping. The partition of K is mapped to PEG and PE, as the low-level hardware mapping. Each block from the tiling of the outer loop is mapped to one PE.

c) Inner Loop Optimization: Temporal Scheduling to Increase Reuse: The inner loop tiling further partitions the input matrix block into tiles to ensure all the tiles run on the same PE but in different time frame.

First, we explain the partition, i.e., the selection of $Mtile$, $Ntile$, and $Ktile$. All these parameters are fixed according to the configuration. We assign the input tile size to be the same as the input tensor size of an MVM unit. Therefore, $Mtile$ is set to 1 since MVM takes 1-D vector as the input. $Ntile$ and $Ktile$ are set as the weight and the height of the MVM unit.

Second, we describe the scheduling. In order to improve temporal data reuse, we need to fully exploit the scratchpad memory. For this reason, we apply another level of tiling inside the inner loop for the N dimension, as shown in the right side of Fig.7. We assign this tile size, $reuse\_len$, according to the capacity of the scratchpad memory. The loop order represents the scheduling, i.e., which tile runs first. We apply loop permutation so that we can first compute the tiles within the same $reuse\_len$, which means all their results can fit in the scratchpad. The second loop we schedule is the K dimension with the purpose of applying output stationary, so that all the results can be accumulated by only accessing the scratchpad memory, eliminating the use of the slow DRAM. The third loop in the schedule is the N-loop and the last one is the M-loop. We set the M-loop last because we want the vector input of the MVM unit remain unchanged (see Sec.IV-B for detail reasons) so that LUT loading overhead is reduced and the LUT latency is hidden by buffer LUTs.

Finally, we describe an example shown in Fig.8 (a). The

brown arrows indicate the running order of the tiles, i.e., the scheduling. Fig.8 (b) further explains the working flow. In Step-❶, we read a vector of $Tile\_W$ in from the GIO. The vector initializes the LUT buffer and will remain there until every related computation is done. In Step-❷, we read the matrix $Tile\_X$ from the local DRAM bank. Note that we optimize the data layout in order to maximize DRAM row buffer hit when access X by $Tile\_x$. In Step-❸, we get the result vector $Tile\_Y$ from the MVM unit and apply accumulations (if necessary) with previous partial results stored in the scratchpad memory using the vector unit. The accumulated result vector is then updated and stored to the scratchpad memory. In Step-❹, when K-loop ends and we get the final result, we write it back to the DRAM bank.

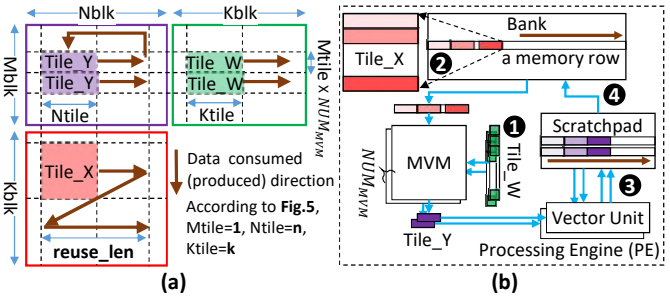


Fig. 8. **Inner loop mapping. (a) Data partitioning within a PE. (b) Computation flow and temporal data reuse scheme based on (a).**

2) Other Kernels: DLUX can also efficiently supports other kernels, including activation, elementwise, and batch normalization. Activation kernels (e.g., $ReLu$) are fused with the GEMM kernel, supported by vector units. Elementwise kernels are also supported by vector units, but require reading data from the local bank (with potential support of permutation unit) into the scratchpad memory before computing. For batch normalization, the mean, variance, and normalization are calculated using the MVM unit and Vector unit, with the help of hierarchical interconnects to perform reduction and broadcasting, and SFU to perform square root and inversion. For Resnet50's FusedBatchnorm, DLUX shows $13.13\times$ speedup an 97.14% energy savings over one V100 GPU.

B. Inter-layer Layout Transformation

Although intra-layer mapping can achieve maximal performance and minimal data movement, inter-layer operation efficiency can be challenging for DLUX due to specialized layout requirements of each DNN layer, and the distributed memory model intrinsic to PIM architecture. We solve that issue by maintaining the input-output layout consistency and the forward-backward layout consistency.

1) Input-Output Layout Consistency: The coalesced data layout is the key to improve the spatial and temporal reuse. To maintain the coalesced layout during the layer-by-layer computation, we need to keep each layer's input and output data layout consistent, considering both the matrix layout transposing and the data partition.

The matrix layout transposing is heavily demanded in the training process, as shown in Fig.1. Before writing back the result of the previous layer, we apply the matrix transpose if necessary, so that the next layer can use the previous result as it is. The transposing is done on-the-fly and locally by the permutation unit in each PE. Here, the data transformation can be local since we only require the conversion between row-major data fetching to column-major data fetching without involving inter-PE traffic. Therefore, there is insignificant data movement energy overhead and the latency can be hidden by layer computations.

We also keep the input/output data layout consistent in terms of data partition across PEs. The tensor reduction in Fig.6 (a) is a good example. The four input tensors are partitioned across four units (PE), so we want to partition and store the output tensor also in four units, accordingly. Otherwise, if we apply a different data partition for the result tensor, e.g., storing the whole tensor into one unit without partition, the next layer which takes the result tensor as the input can only use one unit for computing, leaving the other three underutilized. To keep the data partition consistent, we control the data communication destination addresses like the cases in Fig.6 (b)-(d) when storing the data.

2) Forward-Backward Layout Transpose: Layout transpose of the same tensor is required for the backward pass. A naive transpose scheme will introduce all-to-all broadcasting traffic.

We propose a partial transpose layout which involves no inter-PE data movement. First, instead of moving data blocks from forward mapping PE to backward mapping PE, we only change the index of the block stored in local PE. For example, given a PE associated with block index (i, j) in the forward pass, the block index will be transposed to (j, i). Then, the PE-level data transpose is performed by the permutation unit introduced in Sec.IV-D, where continuous row coalescing becomes continuous column coalescing. Here, coalescing means data is stored in adjacent memory locations in DRAM so access locality is maximal. This transpose operation can be overlapped with other operations. Also, as the same input data tile is read multiple times from the local bank during one inner loop calculation, it is only transposed once for the last round, which incurs very low time overhead.

VI. System Integration

We position DLUX as a standalone accelerator, not as the system memory with computing capability. Though using the PIM architecture, the DRAM in DLUX is its local device/scratchpad memory, instead of part of the system memory. Therefore, DLUX's integration scheme is exactly the same as other accelerators (e.g., GPU). We integrate DLUX to the system with standard bus such as PCIe. The runtime and driver on the host machine copies data from the system memory to the device memory on DLUX, launches the kernels, and finally copies the data back to the system memory if necessary.

DLUX is designed to be transparent to the user. To this end, we provide library-based interface. The library is similar to cuDNN [37] on GPU, which provides well tuned high efficient binary for most DNN operations. The library is then integrated to the backend of frameworks such as Tensorflow [27]. In addition to the library-based interface, we also provide the instruction set for future extensions.

VII. Experiments

A. Experimental Setup

a) Workload: For end-to-end workloads analysis, we use 6 representative benchmarks as shown in Table.I. These benchmarks are selected from MLperf [5], Fathom [6] and NNBench-X [7], and are trained using their default configurations.

b) Hardware Configuration: DLUX adopts 3D-stacking memory configuration similar to previous HMC-based accelerator [9], but additionally, DLUX has the near-bank computing design. The detailed configuration, hardware latency, and energy settings are shown in Table.II. In addition, the CAS_width per bank for internal data reading is set as 2048 bits, the total number of TSVs per cube is 1024, and the inter-cube communication uses 4 full-duplex Serializer/Deserializer (SERDES) links with maximum $30Gb/s$ per link bandwidth.

c) Evaluation Methods: We develop an in-house simulator adapted from ramulator [38] based on the key architecture and timing parameters in Table.II. DLUX is designed to run at a clock frequency of $1GHz$ under $22nm$ technology node. We use cacti-3DD [39] to evaluate the inter-PE interconnects, TSVs, and the 3D DRAM bank access latency and energy. The energy, performance, and area of the scratchpad memory and the SRAM LUT is also simulated by cacti-3DD. The base logic die and the SERDES energy is set based on previous near data processing work [40]. For an FP unit evaluation in the MVM unit and the vector unit, we use an open-source tool [35] to generate VHDL code for various $FP32$ arithmetic and logic units. We also implement RTL codes for the DLUX controller, address translation unit, and permutation unit. These hardware components are synthesized by design compiler considering DRAM process overhead [32] to derive performance, power, and area results. For the GPU evaluation, the baseline DNN training performance is derived from Tensorflow profiling tool and $nvprof$, and the power information is sampled using *nvidia-smi*. When measuring a kernel, we disable all other tasks on the target GPU to isolate the power consumption number. Also, we acquire the steady state of the power consumption for the kernel by running it multiple times and sample the plateau area to ensure accuracy. Because ScaleDeep [41] simulator is not open-source, we optimistically estimate its performance using an analytical model based on the roofline model [8], by using its peak computing throughput ($comp$), peak memory bandwidth (bw), and average hardware utilization ($util$) provided in the paper. For each operator in a given benchmark, we use its arithmetic density to determine whether it is compute-bound or memory-bound in the roofline model. The operator information contained in a computation graph is generated by Tensorflow profiling. For a compute-bound operator, we divide its total operations by the sustained computing throughput ($comp \times util$) to get its execution time. For a memory-bound operator, we divide its total memory footprint by the sustained memory bandwidth ($bw \times util$) to get its execution time. The end-to-end time of a benchmark on ScaleDeep is the accumulation of all its operator time.

B. Performance, Bandwidth, and Area

a) Performance Analysis: For fair comparison assuming a single compute-node, in the end-to-end analysis, we choose one DLUX cube with 8 PIM layers. **DLUX-1** represents naively implementing FP32 logic without LUT optimization, so the number of FP units is halved under the same area constraint. **DLUX-2** represents removing cache without applying the layout transformation scheme, so extra latency is added due to increased row activation number for sub-optimal access locality. **DLUX-3** incorporates both LUT optimization and layout transformation scheme. One Nvidia V100 [4] GPU is used as the performance baseline, and one ScaleDeep [41] FcLayer chip is used as the state-of-the-art DNN training accelerator baseline. During the simulation process, we acquire the entire computation graph of the workload, including kernel types, input/output shapes, and total execution time on a GPU. Then, we evaluate the currently supported kernels on ScaleDeep and DLUX (more than 98% of total execution time for most benchmarks) and assume the non-supported kernels have the same execution time as GPU.

Parameter Names	Configuration
Cubes / Vaults_cube / PEGs_vault	1 / 16 / 8
PEs_PEG / MVMs / Ktile / Ntile	4 / 2 / 4 / 4
Bank / RowBuffer / Scratchpad (Kb)	131072 / 16 / 32
tCK/tRCD/tCCD/tRTP/tRP/tRAS (ns)	1 / 14 / 4 / 4 / 15 / 33
RD,WR/PRE/ACT/Spad (nJ/access)	0.224 / 0.507 / 0.521 / 0.005
MVM / VU / PU (pJ/access)	139.9 / 3.6 / 64.3
interPE/TSV/SERDES (pJ/bit)	0.017 / 4.64 / 4.50

TABLE II
DLUX HARDWARE CONFIGURATION PARAMETERS.

Fig.9 shows the end-to-end execution speedup for 6 representative data center DNN training workloads. We observe that on average, one optimized DLUX-3 cube achieves $6.3\times$ speedup compared to a single V100 GPU, and $2.4\times$ speedup compared to a single ScaleDeep chip. We further investigate the time breakdown, and find DLUX accelerates $GEMM$, $Elementwise$, $Reduction$, and $DataManipulation$ kernels w.r.t GPU at $5.8\times$, $26.5\times$, $3.3\times$, and $14.8\times$ respectively. DLUX can provide significant performance gain for typical memory-bound kernels ($Elementwise$ and $DataMani-pulation$) due to the abundant memory bandwidth provided by near-bank architecture. DLUX can also provide decent speedup for compute-intensive ($GEMM$) and communication intensive kernels ($Reduction$). For $GEMM$, the speedup is attributed to both the high peak FP performance and the high utilization enabled by the proposed software mapping and scheduling techniques. For $Reduction$, the performance roots from the utilization of hierarchical data buses in the 3D memory for efficient data communication.

However, naively implementing near-bank architecture may result in sub-optimal results. From Fig.9 we can observe that DLUX-3 can achieve $1.43\times$ and $1.50\times$ speedup compared with DLUX-1 and DLUX-2, respectively.

We also observe the variation of speedup numbers for different workloads when compared to ScaleDeep, since each workload is a mix of different types of kernels, and each type of kernel takes up varying portions of total execution time and shows various arithmetic density values. For DeepSpeech and Transformer, DLUX obtains significant speedup on $GEMM$ (98% and 65% of total time respectively) with good data reuse and parallelism. For EntropyCoder, DLUX suffers from $GEMM$ (60% of total execution time) with inferior LUT data reuse. For other workloads, since there is not a single time-

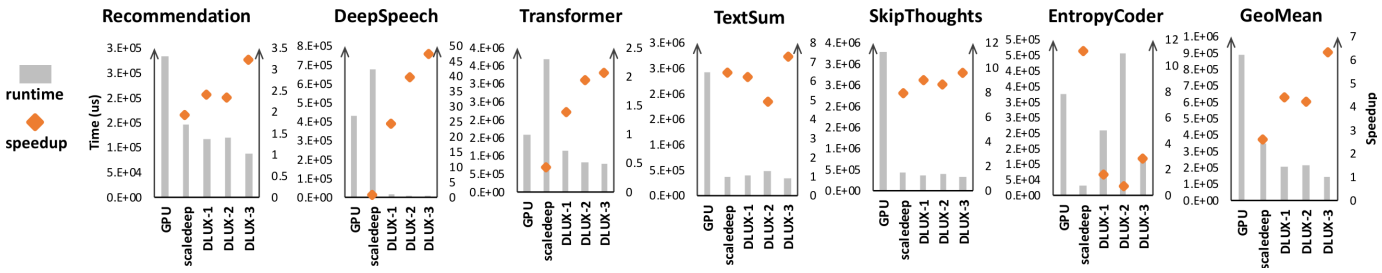


Fig. 9. **End-to-end execution time and speedup comparison for benchmarks in Table.I.**

dominating operation (less than 50% of total execution time), DLUX provides slightly better speedup w.r.t. ScaleDeep due to a mix of operations (1.3× on $GEMM$, 7× on $Elementwise$, 1.1× on $Reduction$, 0.7× on $DataManipulation$).

b) Bandwidth Analysis: We profile the total memory bandwidth and TSV traffic of DLUX and show the results in Fig.10. Fig.10 (a) shows that on average, one DLUX cube can achieve $6TB/s$ sustained memory bandwidth, which is $\sim 6.6\times$ of one V100 GPU bandwidth. The significantly high bank-level bandwidth justifies the adoption of near-bank architecture, which benefits performance for memory bound kernels and reduces data movement for memory intensive kernels. This also proves the necessity of integrating computation logic in the memory die, since by putting the computation units on base logic die, only $256GB/s$ peak memory bandwidth can be achieved, assuming 1 cube. Fig.10 (b) shows that on average, one DLUX cube can achieve $53GB/s$ sustained TSV bandwidth, which equals to 21% utilization assuming $256GB/s$ peak TSV bandwidth. The high TSV bandwidth for Recommendation, DeepSpeech, and Transformer proves the importance of 3D stacking architecture, since different PEs inside the same vault need to use TSVs.

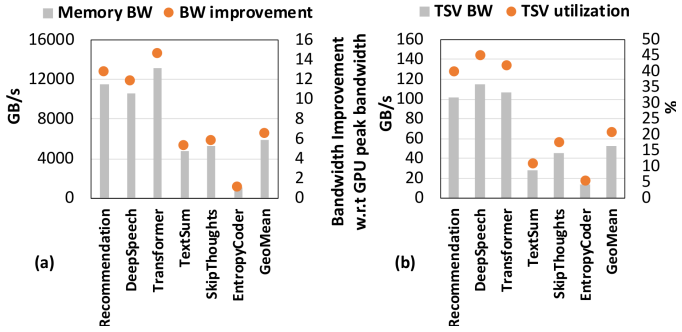


Fig. 10. **(a) Memory bandwidth and improvement w.r.t. GPU. (b) TSV bandwidth and utilization.**

c) Area Analysis: To achieve high performance for compute intensive kernels ($GEMM$), we need a high performance LUT-FPMult. Currently, we equip 32 FP multipliers (FPMult) per PE, and the proposed LUT-FPMult significantly reduces this area overhead by 60.6% as shown in Fig.11. (a) shows that, by replacing area-consuming significand MUL by the proposed LUT-based design, the total overhead is greatly reduced. (b) shows the area breakdown of the LUT-FPMult design, where LUTs takes 35.2% of total area. The FPMult uses an autogenerated VHDL-code multiplier from Flopoco [35]. The LUT-FPMult uses the same VHDL-code, and replaces expensive mantisa-multiplier ($\sim$ 87% of FPMult)

Component	Area (mm^2)	Overhead (%)
MVM Unit (x2)	0.2440	20.02
Vector Unit (x2)	0.1172	9.62
Permutation Unit (x1)	0.0040	0.33
SFU (x1)	0.0140	1.15
Scratchpad (x1)	0.0260	2.12
Full LUT (x1)	0.0095	0.78
Total	0.4146	34.02

TABLE III
THE AREA OF COMPONENTS IN A PE.

with lightweight SRAM-based LUT array ($\sim$ 26.5% of FPMult). We use CACTI-3DD [39] to estimate SRAM LUT's area, performance, and power.

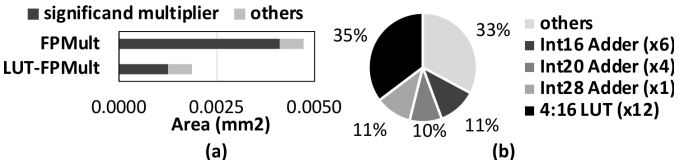


Fig. 11. **LUT-FPMult area analysis**

Using the hardware configuration from Table.II, the area breakdown of a PE is demonstrated in Table.III. The total area overhead of a PE (34.02%) is normalized to a DRAM bank ($1.22mm^2$) under $20nm$ technology node [17]. Here, we assume DRAM process with 3 metal layers will double the area overhead of bank peripheral logic compared to a normal CMOS process with 8 metal layers [32]. LUT can help reduce 14.27% total overhead of peripheral, since LUT can help reduce the normal FPMult overhead of MVM Units (from 35.06% to 20.02%), but only introduce a small extra full LUT overhead (0.78%). Using this area overhead number and assuming one 3D cube has the area footprint of $96mm^2$ [17], we estimate the total silicon footprint of one DLUX cube (8 layers) to be $\sim 998mm^2$. In contrast, one V100 GPU has one processor die with $815mm^2$ and four HBM stacks (4 layers), and the total silicon footprint is $2351mm^2$. Comparison with a V100 GPU, one DLUX cube has 65% less silicon footprint.

C. Energy Analysis

From Fig.13 (a), we observe that one DLUX cube on average improves energy-efficiency (GFLOPS/W) by 42.2× compared with one V100 GPU. To further understand the improvement, we profile the detailed energy consumption numbers and plotted them in Fig.13 (b). The memory access energy includes bank activation/precharge and access energy. The data movement energy includes inter-PE, inter-PEG, inter-vault, and inter-cube energy. The computation energy includes the energy consumption of MVM units, vector units, SFU, and permutation unit. The scratchpad energy covers all access

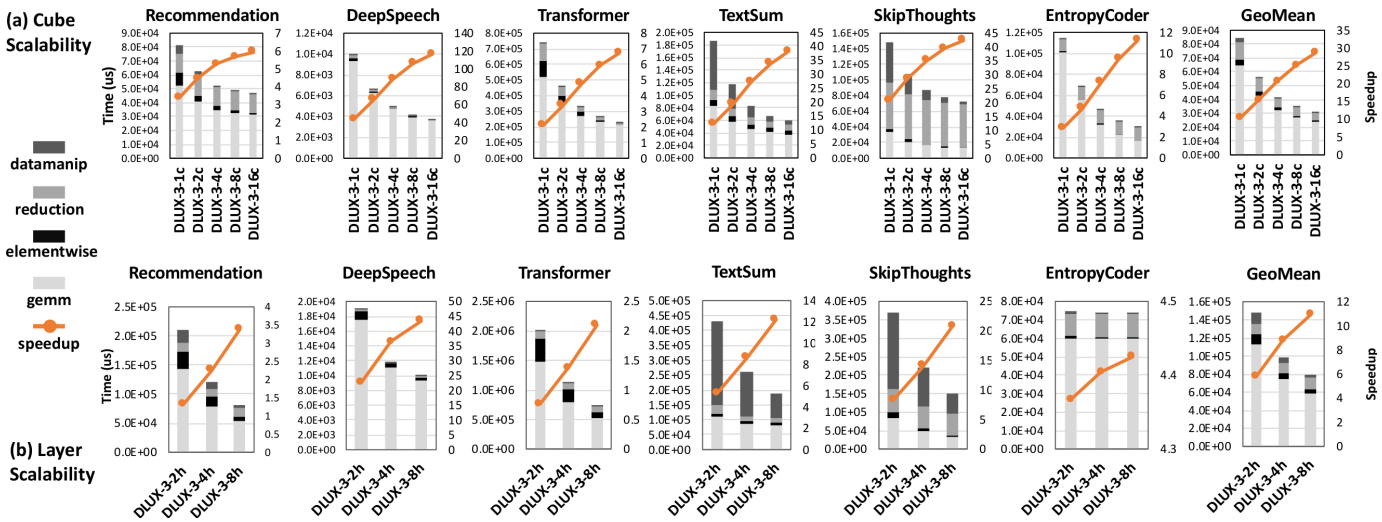


Fig. 12. **(a) Scalability analysis w.r.t the number of cubes. (b) Scalability analysis w.r.t the number of layers.**

energy to scratchpad memory. The energy of the controller and the address translation unit is very small ($< 1\%$), so it is ignored in the analysis. On average, memory access, data movement, computation, and scratchpad access consumes 66.1%, 6.3%, 25.2%, and 2.4% of total energy, respectively. Since the benchmarks we evaluate are memory-intensive, and DLUX has optimization for data movement reduction, it is expected that memory access and computation dominate the energy consumption (over 91.3%).

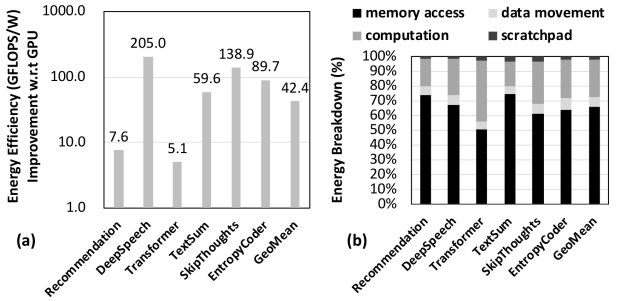


Fig. 13. **(a) Energy-efficiency improvement. (b) Energy breakdown.**

DLUX can provide significant energy savings for all benchmarks, since DLUX can greatly reduce data movement, which is the major energy overhead for compute-centric architectures [42]. This is shown by the small percentage of energy spent on data movement (6.3%) for DLUX. This small energy consumption number is first attributed to the reduction of data movement of near-bank architecture, since the majority of the data is streamed from the local bank. Second, DLUX spatial partitioning and mapping method guarantees minimum data sharing between different PEs, and DLUX scheduling techniques ensure that local data reuse is very high for a PE, so that a remote memory transfer from other PEs is infrequent. The consistent input-output layout assumptions and PIM friendly forward-backward transpose format further reduces inter-layer data movement.

D. Scalability Study

a) Scaling Cube and Layer Number: We study the scalability of the proposed software partitioning and mapping techniques by conducting a weak scaling analysis, where each workload remains unchanged but the number of hardware components are scaled. As shown in Fig.12, we first keep the total number of layers per cube unchanged, while scaling the total cube number from 1 (DLUX-3-1c) to 16 (DLUX-3-16c) in (a), and then we keep the total number of cubes unchanged, while scaling the number of layers per cube from 2 (DLUX-3-2h) to 8 (DLUX-3-8h) in (b). For each evaluated configuration, we also plot the breakdown of the execution time into four categories of computation kernels as well as the speedup ratio w.r.t. one V100 GPU.

For the cube scaling results, on average, by doubling the total cube number, a $1.30\times$ scaling ratio can be achieved. Further analysis shows that $GEMM$, $Elementwise$, $Reduction$, and $Datamanipulation$ kernels can achieve $1.28\times$, $1.50\times$, $1.23\times$, and $1.84\times$ scaling ratio, respectively. For all the benchmarks, we observe that, when we scale the cube number, the performance bottleneck will gradually become either $GEMM$ or $Reduction$, which all have low scaling ratio. The reason is that $Elementwise$ and $Datamanipulation$ kernels have abundant parallelism and simple data access and communication patterns, so it is relatively simple to scale them by adding more cubes. For $Reduction$ kernels, depending on whether the reduction dimension is distributed among different cubes, different scaling ratios can be achieved due to the data dependency in reduction process. For $GEMM$ kernels, the non-ideal (ideal scaling ratio should be 2) scaling ratio is attributed to the tradeoff between the spatial utilization and the temporal utilization. The current partitioning scheme increases spatial utilization by distributing input data evenly across all vaults, so that an increasing vault number will decrease the local data reuse per vaults, harming temporal utilization. The N dimension is partitioned among all vaults, where the bank in each vault gets an $Nblk$ partition. Since the LUT data reuse depends on the tiling of the $Nblk$ dimension, for $GEMM$ kernels with small N size, the larger the total cube number is, the smaller the effective data reuse will be. This explains that for $GEMM$ kernels with larger N, near optimal scaling ratio is achieved, and other kernels with smaller N achieves sub-optimal scaling ratio.

For the layer scaling results, by doubling the number

of layers per cube, on average, a $1.23\times$ scaling ratio can be achieved for all the evaluated benchmarks. Further analysis shows that $GEMM$, $Elementwise$, $Reduction$, and $Datamanipulation$ kernels can achieve $1.24\times$, $1.36\times$, $0.98\times$, and $1.54\times$ scaling ratio, respectively. For $GEMM$ and $Reduction$ kernels, the scaling is non-ideal since data reduction overhead exists. Taking $GEMM$ kernels for example, the K dimension is partitioned among all PEs in the same vaults, where the bank in each vault gets a $Kblk$ partition. By adding more layers, reduction operations will increase. However, in the shared hierarchical bus, TSVs among layers are shared resources. Adding more layers will reduce the workloads per PE, as well as add total reduction latency in the final step. Although spatial parallelism is achieved by adding more layers, the inter-PEG reduction time will take over the PE local computation time if K is small and layer number increases. This explains why adding more layers will not help reduce the $GEMM$ execution time for EntropyCoder.

E. Sensitivity Study

This section conducts the sensitivity study for choosing the best $Ktile$ and $Ntile$ for MVM, and the number of MVM units (NUM_{MVM}) per PE. Note that by adding multiple number of MVM units per PE, they together function as a larger matrix-matrix multiplication unit. The result will justify the PE hardware configuration detailed in Table.II.

$$Perf_{sustain} \approx Perf_{peak} \cdot \frac{N_{round} \cdot t_{comp}}{N_{round} \cdot t_{comp} + t_{LUT}} \quad (1)$$

We will use $GEMM$ as a case study throughout this section. A simple performance model is shown in Eq.1, where t_{comp} is the MVM computation time per round, and N_{round} is the total rounds of computation before reloading LUT. LUT reload latency is represented as t_{LUT}. In order to explore the design tradeoff, we assume the total number of LUT-FPMult units (NUM_{unit}) is fixed per bank ($Ktile * Ntile * NUM_{MVM} = NUM_{unit}$), and NUM_{MVM} is fixed for each exploration. To increase $Perf_{sustain}$, a simple approach is to decrease t_{LUT} by reducing $Ktile$, but it will increase $Ntile$. Given a fixed $Nblk$, increasing $Ntile$ will reduce N_{round}, which in turn harm $Perf_{sustain}$. We explore this design space in Fig.14(a), where each line assumes a fixed NUM_{unit}. The y axis is $Perf_{sustain}$, and the x axis is one configuration of [$Ktile$,$Ntile$] pair, where NUM_{MVM} is implied. Each data point is the geomean of the $Perf_{sustain}$ under different data reuse conditions ($Nblk$ ranges from 16 to 512). For each setting of $Ktile$ and $Ntile$, We also estimate the number of adders and the scratchpad size to sustain this throughput. We choose an optimal design point ($Ktile = Ntile = 4$, $NUM_{MVM} = 2$) where area overhead is minimized while $Perf_{sustain}$ is high enough.

VIII. Discussion

A. Scaling Model Size

In order to study the impact of larger model size with the increasing number of hardware components, we conduct a case study on Transformer [23] with different combinations of DLUX cubes and training batchsize (bs). The speedup and

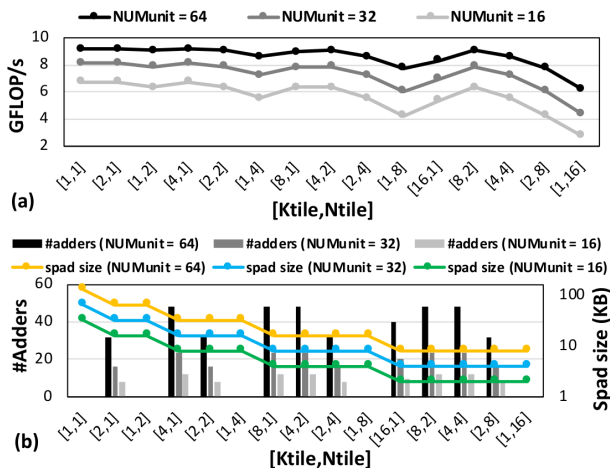


Fig. 14. **PE design space exploration. (a) $Perf_{sustain}$ (b) Hardware overhead (#adders and Spad size)**

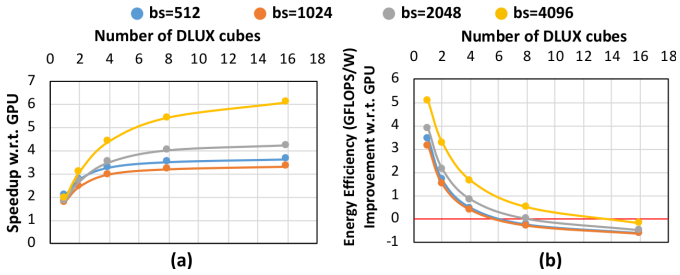


Fig. 15. **Scaling the number of cubes under different batchsize (bs) of Transformer [23]: (a) Speedup, (b) Energy-efficiency improvement.**

energy-efficiency improvement results of DLUX is shown in Fig.15 (a) and (b), respectively. On the one hand, we observe that DLUX performance scales better with larger batchsize as shown in Fig.15 (a). That is because larger batchsize is beneficial for DLUX to expoit data parallelism, where added DLUX cubes can perform computation in parallel. When batchsize is not large enough, providing more DLUX cubes does not boost performance significantly as shown in $bs = 512$ and $bs = 1024$. On the other hand, we observe that DLUX energy efficiency improvement is higher for larger batchsize under the same cube number as shown in Fig.15 (b). That is because larger batchsize enables each DLUX cube to have more workloads for local processing before global data communication, which increases local data reuse and improves energy-efficiency. We also note that the energy efficiency drops as the number of cubes increases. This is caused by the increased amount of data traffic among cubes, which incurs more energy consumption than local data processing.

B. Supporting CNN Models

DLUX also supports the training of CNN models. We use VGG16 [43] as a case study to show that DLUX can achieve comparable performance with GPU. According to our profiling results using Tensorflow, VGG16 consists most of convolution operations which takes majority of the execution time on GPU ($> 99\%$). We support the convolution operations using the im2col method [37] which executes them as normal GEMM operations. Using the hardware configuration in Table.II, we measure the performance of VGG16 and find that DLUX achieves $1.44\times$ speedup and $4.37\times$ energy-efficiency improvement in comparison with GPU. To

further explain the speedup, we discover that most of the convolution operations in VGG16 have medium arithmetic density ($15FLOP/Byte$) that is very close to GPU's performance/bandwidth ratio ($17.5FLOP/Byte$). This enables GPU to have more data reuse and generate less DRAM traffic as compared to the RNN/MLP models evaluated before, which explains the less significant speedup. To further investigate the energy-efficiency, we profile the energy breakdown and find that data movement energy only constitutes 14.35% on DLUX, while computation takes 63.81% of total energy. The small portion of energy dedicated on data movements explains the energy-efficiency improvement over GPU.

C. Comparison with Software Methods

Some software-based techniques, including neural network pruning [44] and gradient compression [45], have been proposed to tackle the memory-bound challenges in DNN training. Compared with DLUX, they either decrease training accuracy [44], or focus on a different training problem [45]. First, the pruning method to reduce memory footprint introduce approximation error during the computation, which reduces the final training accuracy. For example, PruneTrain [44] reduces accuracy on Imagenet by 1.9%. DLUX provides DNN training speedup without sacrificing accuracy. Second, the pruning methods mostly focus on CNN but have not achieve success on RNN models which are more memory-bound. In comparison, DLUX can support all models and is more effective for memory-bound models like RNN and MLP. Third, previous work [45] focuses on distributed training among multiple accelerators by compressing the gradient, which is orthogonal to the research of this work. DLUX focuses on improving the DNN training performance and energy-efficiency in a single accelerator, and previous method [45] can be applied on multiple DLUX accelerators.

IX. Related Work

a) Process-in-DRAM: Previous Process-in-DRAM work either suffers from large area overhead of integrating general-purpose logic [46], or confined to supporting fixed-point arithmetic [33], [47] or approximate computing [32]. Without changing DRAM timing, our work incorporates lightweight control logic to support general DNN training, and enables significant floating point performance improvements by adding a large number of low-overhead LUT-FP units.

b) DNN inference accelerators: Most of the DNN accelerator researches [2], [48]–[51] focus on inference, which embraces the simplicity of the fix-point multiplier without accuracy loss [52], [53]. They are not adoptable for training, which must use floating point to avoid the accuracy loss.

c) Compute-centric DNN training accelerators: Since mix precision training cannot maintain accuracy for all DNN applications, we only compare DLUX to accelerators that support FP32. ScaleDeep [41] claims $6\times$-$28\times$ faster than Titan-X GPU (Maxwell) in iso-power mode. For fair comparison, we use one DLUX cube to compare with 1 FcLayer chip for memory intensive data center training workloads, and demonstrates $2.4\times$ speedup compared with ScaleDeep.

d) Memory-centric training accelerators: Plenty of ideas have been proposed for both the process-in-memory (PIM) architecture [47], [54]–[56] and the near-data processing (NDP) architecture [9], [33], [57], [58]. Drisa [47], Prime [54], Neurocube [57],and Tetris [58] have proposed to apply PIM/NDP for DNN applications, but are limited to fix-point precision inference. For DNN training with PIM architecture, Pipelayer [55] and FloatPIM [56] leverages RRAM-based analogy/digital computing and therefore remains a long-term research. The endurance problem of RRAM devices (~ 1000 training tasks before failure [56]) and long latency to support an FP32 multiplication (43190 cycles [56]) remain unsolved. McDRAM [33] designed MAC close to the bank buffer. However, limited by the large overhead of building complex circuit with the DRAM process, McDRAM cannot deliver competitive performance. For DNN training with NDP architecture, processing units are designed on the logic die of either HBM or HMC. DLUX achieves $6.59\times$ improvement over the peak bandwidth of processing-on-logic-die, because putting logic on the base die of the 3D-stack memory does not embrace extra bandwidth than GPU's HBM. Although software transformation could improve the bandwidth utilization, the performance is bounded by the peak memory bandwidth.

X. Conclusion

We present both the hardware architecture and the software mapping and scheduling techniques for DLUX, a 3D-stacking LUT-based near-bank accelerator for DNN training. We address the area-constrained performance challenge by leveraging DRAM LUT for computation. We design a hierarchical lookup-table for high performance and low overhead FP computing. Then, to enable efficient communications, the hierarchical data buses are utilized to perform high bandwidth data broadcasting operations. Beside the hardware, we also propose mapping and scheduling techniques to further improve the spatial and temporal utilization of DLUX. In addition, transparent and low overhead techniques are invented to ensure the input-output layout consistency and forward-backward layout transposition. We finally evaluate DLUX on representative deep learning training tasks and compare the results with Tesla V100 GPU. Area analysis shows that DLUX can reduce overhead by 60% against direct implementation of FP32 unit. Compared with Tesla V100 GPU, end-to-end evaluation shows that DLUX provides on average $6.3\times$ speedup and $42\times$ energy efficiency improvement.

References

[1] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro *et al.*, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 620–629.

[2] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick,

N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*. IEEE, 2017, pp. 1–12.

[3] H. Zhu, M. Akrout, B. Zheng, A. Pelegris, A. Phanishayee, B. Schroeder, and G. Pekhimenko, "Tbd: Benchmarking and analyzing deep neural network training," *arXiv preprint arXiv:1803.06905*, 2018.

[4] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the nvidia volta gpu architecture via microbenchmarking," *arXiv preprint arXiv:1804.06826*, 2018.

[5] "MLPerf," https://github.com/mlperf/training.

[6] R. Adolf, S. Rama, B. Reagen, G.-Y. Wei, and D. Brooks, "Fathom: Reference workloads for modern deep learning methods," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2016, pp. 1–10.

[7] X. Xie, X. Hu, P. Gu, S. Li, Y. Ji, and Y. Xie, "NNBench-x: Benchmarking and understanding neural network workloads for accelerator designs," *IEEE Computer Architecture Letters*, vol. 18, no. 1, pp. 38–42, 2019.

[8] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[9] J. Liu, H. Zhao, M. A. Ogleari, D. Li, and J. Zhao, "Processing-in-memory for energy-efficient neural network training: A heterogeneous approach," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 655–668.

[10] D. Kim, T. Na, S. Yalamanchili, and S. Mukhopadhyay, "Deeptrain: A programmable embedded platform for training deep neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2360–2370, Nov 2018.

[11] E. Azarkhish, D. Rossi, I. Loi, and L. Benini, "Neurostream: Scalable and energy efficient deep learning with smart memory cubes," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 2, pp. 420–434, 2017.

[12] F. Schuiki, M. Schaffner, F. K. Gürkaynak, and L. Benini, "A scalable near-memory architecture for training deep neural networks on large in-memory datasets," *arXiv preprint arXiv:1803.04783*, 2018.

[13] L. Xu, D. P. Zhang, and N. Jayasena, "Scaling deep learning on multiple in-memory processors," in *Proceedings of the 3rd Workshop on Near-Data Processing*, 2015.

[14] N. P. Jouppi, C. Young, N. Patil, and D. Patterson, "A domain-specific architecture for deep neural networks," *Communications of the ACM*, vol. 61, no. 9, pp. 50–59, 2018.

[15] C. Nicol, "A Dataflow Processing Chip for Training Deep Neural Networks," 2017.

[16] H. Consortium *et al.*, "Hybrid memory cube specification 2.1," *Retrieved from hybridmemorycube. org*, 2013.

[17] K. Sohn, W. Yun, R. Oh, C. Oh, S. Seo, M. Park, D. Shin, W. Jung, S. Shin, J. Ryu, H. Yu, J. Jung, K. Nam, S. Choi, J. Lee, U. Kang, Y. Sohn, J. Choi, C. Kim, S. Jang, and G. Jin, "A 1.2 v 20 nm 307 gb/s hbm dram with at-speed wafer-level io test scheme and adaptive refresh considering temperature distribution," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 250–260, 2017.

[18] M. O'Connor, N. Chatterjee, D. Lee, J. Wilson, A. Agrawal, S. W. Keckler, and W. J. Dally, "Fine-grained dram: energy-efficient dram for extreme bandwidth systems," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017, pp. 41–54.

[19] P. Gu, X. Xie, Y. Ding, G. Chen, W. Zhang, D. Niu, and Y. Xie, "ipim: Programmable in-memory image processing accelerator using near-bank architecture," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 804–817.

[20] Y.-B. Kim and T. W. Chen, "Assessing merged dram/logic technology," *Integration*, vol. 27, no. 2, pp. 179–194, 1999.

[21] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T.-S. Chua, "Neural collaborative filtering," in *Proceedings of the 26th international conference on world wide web*, 2017, pp. 173–182.

[22] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates *et al.*, "Deep speech: Scaling up end-to-end speech recognition," *arXiv preprint arXiv:1412.5567*, 2014.

[23] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[24] A. M. Rush, S. Harvard, S. Chopra, and J. Weston, "A neural attention model for sentence summarization," in *ACLWeb. Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, 2017.

[25] R. Kiros, Y. Zhu, R. R. Salakhutdinov, R. Zemel, R. Urtasun, A. Torralba, and S. Fidler, "Skip-thought vectors," in *Advances in neural information processing systems*, 2015, pp. 3294–3302.

[26] G. Toderici, D. Vincent, N. Johnston, S. Jin Hwang, D. Minnen, J. Shor, and M. Covell, "Full resolution image compression with recurrent neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 5306–5314.

[27] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[28] J. Standard, "High bandwidth memory (hbm) dram," *JESD235*, 2013.

[29] J. T. Pawlowski, "Hybrid memory cube (HMC)," in *2011 IEEE Hot chips 23 symposium (HCS)*. IEEE, 2011, pp. 1–24.

[30] "Berkeley hardware floating-point units," https://github.com/ucb-bar/berkeley-hardfloat.

[31] P. Kurup and T. Abbasi, *Logic synthesis using Synopsys*. Springer Science & Business Media, 2012.

[32] A. Yazdanbakhsh, C. Song, J. Sacks, P. Lotfi-Kamran, H. Esmaeilzadeh, and N. S. Kim, "In-dram near-data approximate acceleration for gpus," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, 2018, pp. 1–14.

[33] H. Shin, D. Kim, E. Park, S. Park, Y. Park, and S. Yoo, "Mcdram: Low latency and energy-efficient matrix computations in dram," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2613–2622, 2018.

[34] S. Aga, N. Jayasena, and M. Ignatowski, "Co-ml: a case for collaborative ml acceleration using near-data processing," in *Proceedings of the International Symposium on Memory Systems*. ACM, 2019, pp. 506–517.

[35] F. De Dinechin and B. Pasca, "Designing custom arithmetic data paths with flopoco," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, 2011.

[36] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A case for exploiting subarray-level parallelism (SALP) in DRAM," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 368–379, 2012.

[37] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[38] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, Jan 2016.

[39] K. Chen, S. Li, N. Muralimanohar, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-3dd: Architecture-level modeling for 3d die-stacked dram main memory," in *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2012, pp. 33–38.

[40] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "Ndc: Analyzing the impact of 3d-stacked memory+ logic devices on mapreduce workloads," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2014, pp. 190–200.

[41] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, and A. Raghunathan, "Scaledeep: A scalable compute architecture for learning and evaluating deep networks," in *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, vol. 45, no. 2. ACM, 2017, pp. 13–26.

[42] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, Feb 2014, pp. 10–14.

[43] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[44] S. Lym, E. Choukse, S. Zangeneh, W. Wen, S. Sanghavi, and M. Erez, "Prunetrain: fast neural network training by dynamic sparse model reconfiguration," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–13.

[45] S. Han and W. J. Dally, "Bandwidth-efficient deep learning," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.

[46] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent ram," *IEEE micro*, vol. 17, no. 2, pp. 34–44, 1997.

[47] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "Drisa: A dram-based reconfigurable in-situ accelerator," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 288–301. [Online]. Available: http://doi.acm.org/10.1145/3123939.3123977

[48] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 1–13, 2016.

[49] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," in *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*. IEEE, 2017, pp. 535–547.

[50] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," in *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, vol. 45, no. 2. ACM, 2017, pp. 548–560.

[51] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 2016, p. 17.

[52] 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 *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 609–622.

[53] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43th Annual International Symposium on*, vol. 44, no. 3. IEEE Press, 2016, pp. 367–379.

[54] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 27–39, 2016.

[55] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 541–552.

[56] M. Imani, S. Gupta, Y. Kim, and T. Rosing, "Floatpim: In-memory acceleration of deep neural network training with high precision," in *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, 2019, pp. 802–815.

[57] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 380–392, 2016.

[58] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 751–764.

Peng Gu received the B.S. degree from Tsinghua University, Beijing, China, in 2015, and the M.S. degree from University of California, Santa Barbara, USA, in 2017. He is currently pursuing the Ph.D. degree with the Department of Electrical Computer Engineering from University of California, Santa Barbara, USA. His research interests include architecture studies for memory system and process-in-memory accelerators.

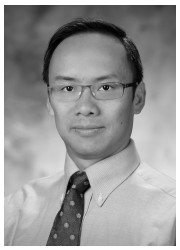
Xinfeng Xie received the B.S. degree from Peking University, Beijing, China, in 2017. He is currently a Ph.D. student at the Department of Electrical and Computer Engineering, University of California, Santa Barbara. His current research interests include computer architecture (especially emerging architectures such as near-data processing), programming language, and computer system.

Shuangchen Li received the B.S. and M.S. degree from Tsinghua University, Beijing, China, in 2011 and 2014, respectvely, and the Ph.D. degree in Department of Electrical Computer Engineering from University of California, Santa Barbara, USA, in 2018. He is currently a research scientist in the Computing Technology Lab of Alibaba DAMO Academy. He works on memory related computer architecture, with emphasis on processing-in-memory architectures, emerging non-volatile technologies, and deep learning accelerators.

Dimin Niu received the B.S. and M.S. degree from Tsinghua University, Beijing, China, in 2005 and 2008, respectvely, and the Ph.D. degree in Computer Science from the Pennsylvania State University, Pennsylvania, USA, in 2012. He is currently a research scientist in the Computing Technology Lab of Alibaba DAMO Academy. Previously, he was a staff memory architecture in Memory Solutions Lab at Samsung Semiconductor Inc. His research interests include memory and storage system, process-in-memory, and domain specific architecture.

Krishna T. Malladi received the B.S. degree from Indian Institute of Technology, Kanpur, India, in 2004 and 2009, respectvely, and the Ph.D. degree in Electrical Engineering from Stanford University, Palo Alto, USA, in 2013. Krishna T. Malladi is currently a staff architect in the Memory Solutions Lab in the US R&D center at Samsung Semiconductor. His research interests include next-generation memory and storage systems for datacenter platforms.

Hongzhong Zheng received the B.S. and M.S. degree from Huazhong University of Science and Technology, Wuhan, China, in 1998 and 2001, respectvely, and the Ph.D. degree in Computer Engineering from the University of Illinois at Chicago, Chicago, USA, in 2009. He is currently a research scientist in the Computing Technology Lab of Alibaba DAMO Academy. Previously, he was a staff memory architecture in Memory Solutions Lab at Samsung Semiconductor Inc. His research interests include computer architecture, energy-efficient computing system designs, novel memory architecture, emerging memory technology and performance modeling.



Yuan Xie (F'15) received the Ph.D. degree from the Electrical Engineering Department, Princeton University, Princeton, NJ, USA, in 2002.,He was with IBM, Armonk, NY, USA, from 2002 to 2003, and AMD Research China Lab, Beijing, China, from 2012 to 2013. He was a Professor with Pennsylvania State University, State College, PA, USA, from 2003 to 2014. He is currently a Professor with the Department of Electrical and Computer Engineering, University of California at Santa Barbara, Santa Barbara, CA, USA. His current research interests include computer architecture, electronic design automation, and very large scale integration design.,Dr. Xie is an expert in computer architecture who has been inducted to ISCA/MICRO/HPCA Hall of Fame. He served as the TPC Chair for HPCA 2018 and he is the Editor-in-Chief for the ACM Journal on Emerging Technologies in Computing Systems, a Senior Associate Editor for the ACM Transactions on Design Automation for Electronics Systems, and an Associate Editor for the IEEE Transactions on Computers.