PIMFlow: Compiler and Runtime Support for CNN Models on Processing-in-Memory DRAM

Yongwon Shin*
ywshin@postech.ac.kr
Graduate School of AI
POSTECH
South Korea

Juseong Park*
wntjd9805@postech.ac.kr
Dept. of Computer Science
and Engineering
POSTECH
South Korea

Sungjun Cho
allencho1222@postech.ac.kr
Dept. of Computer Science
and Engineering
POSTECH
South Korea

Hyojin Sung†
hsung@postech.ac.kr
Graduate School of AI
POSTECH
South Korea

Abstract

Processing-in-Memory (PIM) has evolved over decades into a feasible solution to addressing the exacerbating performance bottleneck with main memory by placing computational logic in or near memory. Recent proposals from DRAM manufacturers highlighted the HW constraint-aware design of PIM-enabled DRAM with specialized MAC logic, providing an order of magnitude speedup for memory-intensive operations in DL models. Although the main target for PIM acceleration did not initially include convolutional neural networks due to their high compute intensity, recent CNN models are increasingly adopting computationally lightweight implementation. Motivated by the potential for the software stack to enable CNN models on DRAM-PIM hardware without invasive changes, we propose PIMFlow, an end-to-end compiler and runtime support, to accelerate CNN models on a PIM-enabled GPU memory. PIMFlow transforms model graphs to create inter-node parallelism across GPU and PIM, explores possible task- and data-parallel execution scenarios for optimal execution time, and provides a code-generating back-end and execution engine for DRAM-PIM. PIMFlow achieves up to 82% end-to-end speedup and reduces energy consumption by 26% on average for CNN model inferences.

CCS Concepts: • **Software and its engineering → Compilers**; • **Hardware → Emerging architectures**.

Keywords: Processing-in-memory, CNN models

*Both authors contributed equally to this research.
†Also with POSTECH, Dept. of Computer Science and Engineering.

1 Introduction

The main memory has become an increasingly critical performance and energy bottleneck in modern computing systems, as the performance gap between compute units and memories widens [9, 10] and the demand for efficient large-scale data processing grows. Among many architectural efforts to address this "memory wall" [5, 6, 24, 36], processing-in-memory (PIM) places computational logic in or near memory devices to perform in-memory operations, thus effectively eliminating the data movement overhead for PIM-offloaded computations [3, 15, 23, 26, 27, 37, 38, 45].

While the idea of PIM is not new [17, 18, 42], recent advances in memory technologies motivated major DRAM manufacturers to explore its potential as a commercial DRAM solution [26, 37, 38]. These efforts discovered that the area and power constraints on the number and complexity of PIM compute units are much more stringent than those assumed by previous PIM approaches, and focused on integrating multiply-accumulate (MAC) units in the logic layer of 3D-stacked memory [37], or after bit-line sense amplifier (BLSA) [26] while meeting fab-level energy and area constraints. The resulting "DRAM-PIM" achieved an order of magnitude acceleration for memory-intensive fully-connected (FC) layers in various DNN models.

On the other hand, convolutional layers, one of the two major building blocks of DNN models along with FC layers, were not considered main targets for PIM due to their high computational intensity and data reuse, with little prospect for PIM logic with limited computational power to beat massively parallel GPUs. Nevertheless, we see a strong potential for PIM acceleration with a type of separable convolutional layers, e.g., pointwise or 1x1 convolutional layers. As shown in Fig. 1, these layers have relatively low arithmetic intensity as they are equivalent to FC layers after convolution lowering, and are increasingly used in modern CNN models to reduce dimensionality (*ResNet50* [25]) or combine with depthwise (DW) separable layers to replace regular convolutional layers to reduce computation load while retaining accuracy (*EfficientNetB0* [57] and *MobileNetV2* [51]).
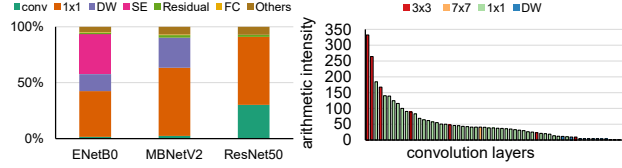
Figure 1. Runtime breakdown of CNN models on NVIDIA RTX 2080 Ti GPU (left) and arithmetic intensity (# of MAC divided by # of LD/ST) of convolutional layers in the models (right).

Taking a step forward, we envision a holistic deep learning (DL) software stack for DRAM-PIM with optimizing compiler and runtime support. Prior work on software support for PIM focused mainly on providing user interfaces to explicitly specify and schedule PIM-offloadable code regions [27, 33], while recent proposals include more sophisticated placement and scheduling mechanisms with cost models [22, 58] and GPU-PIM parallel execution support [48]. Our key insight is that compilers can exploit PIM-offloading opportunities inherent in input DL models, but also can systematically *create* them by transforming model graphs so that they have more nodes to accelerate on PIM and overlap execution with GPU. This will significantly extend the scope of PIM target software and improve PIM utilization. Automating the offloading process can be challenging due to the complexity of code generation, but we observed that configuring the DRAM-PIM as both GPU memory and PIM device can simplify the task.

Thus, we propose PIMFlow, a compiler and runtime solution that accelerates CNN models on a PIM-enabled GPU memory based on [26], with PIM-aware graph transformations and PIM command generation support. PIMFlow takes model inference graphs from DL frameworks as input and searches the space of possible placement and scheduling options for convolution layers. PIMFlow supports multiple mixed-parallel execution models, where a convolution layer can be distributed across GPU and DRAM-PIM (multi-device data parallel) or multiple convolution layers partially overlap execution (pipelined), in addition to traditional device offloading (heterogeneous parallel). The search result is used to transform the graphs accordingly, which are then compiled and executed by our TVM back-end for DRAM-PIM.

In the DRAM-PIM back-end, PIMFlow performs a memory layout optimization to minimize data movement overheads introduced by parallelization, and optimizes DRAM-PIM code generation for more efficient command sequences. PIMFlow can also be viewed as an SW/HW co-design effort; our target DRAM-PIM architecture is extended with new DRAM-PIM commands to allow more fine-grained data reuse specifically needed to efficiently handle convolutional layer matrices, and the code generator supports them. Our experimental results showed up to 82% and 33% (34% and 23%, on average) speedup for all evaluated CNN models against the GPU and DRAM-PIM baselines without PIMFlow, respectively. The reduced execution time led to lower energy

consumption by 26% on average against the GPU baseline.

The contributions of the paper are as follows:

- We systematically analyze performance trends of convolution layers on the DRAM-PIM hardware, and define the PIMFlow execution model accordingly. To our knowledge, this paper concerns the first effort to expand the scope of PIM-offloadable computations and increase PIM utilization by transforming computations to exploit GPU-PIM mixed-parallel execution.
- We propose an end-to-end compiler and runtime solution that enables all types of convolution layers on DRAM-PIM. The resulting PIMFlow provides up to 82% speedup over GPU for evaluated CNN models, showing the strong potential for PIMFlow as an optimizing compiler toolchain for industrial DRAM-PIM.
- We extend the DRAM-PIM memory architecture to reduce PIM command latencies, support convolution more efficiently, and facilitate mixed-parallel execution across GPU and DRAM-PIM.

In the rest of the paper, we first provide background information for DRAM-PIM architectures and convolution operation. Section 3 describes our preliminary analysis, which motivates the design and implementation of PIMFlow in Section 4. Section 5 and 6 show the experimental result for PIMFlow with several CNN models. Section 7 discusses the overheads incurred by PIMFlow implementation. Section 8 presents related work, and Section 9 concludes the paper.

2 Background

2.1 Digital DRAM-PIM Architecture

We assume Newton and its sister architecture [26, 38] as our baseline PIM-enabled DRAM. They present a DRAM manufacturer's constraint-aware design for commercially viable DRAM-PIM. Newton defines its target operation as memory-bound matrix-vector multiplication of one large operand with low data reuse and one small operand possibly with high data reuse. Fig. 2 shows a structural overview of the architecture on the right. With the large operand in a memory cell array (1) and the small operand in a global buffer per memory channel (2), Newton performs bank-level parallel matrix-vector multiplication of these two operands using MAC units placed after BLSA. MAC logic consists of the reduction tree after column I/O (3), where fetched matrices are multiplied with input data in the global buffer and then summed up, and the result latches (4) to accumulate MAC results. Other operations (e.g., element-wise operations) are not supported. Although Newton features a small number of compute units (16 multipliers and a reduction tree per bank) compared to previous digital DRAM-PIM approaches [15], it produces approximately 20x speedup for memory-bound DL models such as BERT [16] and recommendation models (DLRM) [44], showing the feasibility of commercial DRAM-PIM.
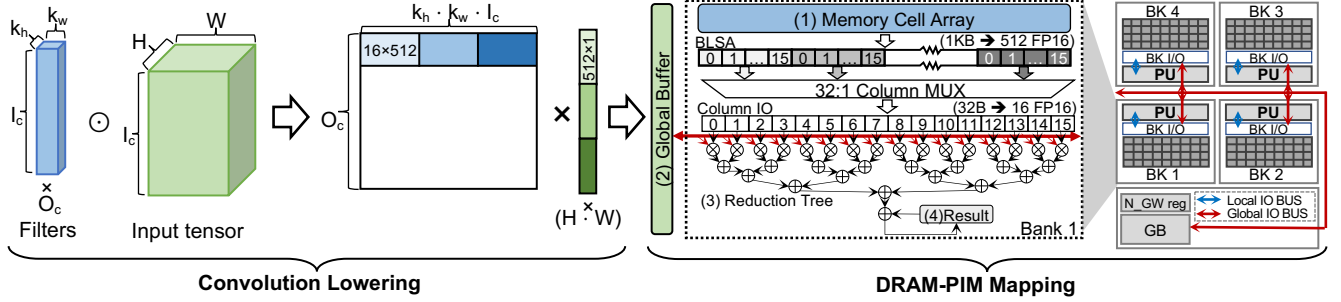
Figure 2. Convolution lowering and mapping to the baseline DRAM-PIM architecture [26].

2.2 Convolution Operation and Implementation

Convolutional neural networks (CNN) are widely used in various vision and graphics tasks such as image classification, object detection, and image/video segmentation [25, 28, 49, 50]. CNN stacks multiple convolutional (CONV) layers to extract image features. Shallow stacks detect simple geometric shapes like edges, while deeper stacks extract more complex and higher-level features. Convolution operation works by sliding "filters" across spatial dimensions (width and height) of input images and computing output values as the dot product between the filter and input element. There exist many convolution algorithms optimized for specific data shapes and hardware [7, 19, 35], and "convolution lowering," which implements convolution as matrix-matrix multiplication of rearranged image and filter matrices, is widely used for data-parallel accelerators [13].

To map CONV layers to the DRAM-PIM hardware, we apply convolution lowering first and iteratively perform matrix-vector multiplications. For example, as shown in Fig. 2, after convolution lowering, convolution becomes the multiplication of input tensor (green) and filter weight (blue) matrices. To map the input tensor to the DRAM-PIM global buffer, the input tensor matrix is broken down into multiple vectors. Similar to the tiling approach in [26], we place the filters in the memory cell array in advance, and then fetch the input tensor from GPU memory to the global buffer and activate the PIM compute unit. We assume the NHWC, i.e., channels-last, format for input data layout as it guarantees contiguous memory access in the channel dimension.

3 Preliminary Analysis

Prior work on software support for DL models on DRAM-PIM assumed the default heterogeneous parallelism, where a host CPU schedules computational graph nodes on the host *or* device(s) (including GPU and PIM) and serially launches each node in a graph traversal order that is then executed, exploiting data parallelism within the node. While PIM can provide significant runtime acceleration for memory-intensive operations in this mode of parallelism [3, 27, 48], support for both intra- and inter-node parallelism across multiple devices could bring out additional speedup for a wider range of computations, and recent research explored its potential [21, 47]. Thus, we conducted a preliminary analysis of

how convolution layers perform on GPU-PIM systems with varying configurations to examine the prospect of the performance gain from parallelizing them across devices and guide our compiler and architecture design in Section 4.

1. The majority of DNN inference models including CNN do not have enough inherent inter-node parallelism to fully utilize PIM units in parallel with GPU. Many DNN models are known to not offer much inter-node parallelism since data simply flow through a straight-lined sequence of layers without branches. We found that zero or less than 17% of the graph nodes have nodes without data-flow dependency in 75% of the Torchvision [40] CNN models. Enabling independent nodes to execute in parallel on GPU and PIM introduces software complexity to solve the placement and scheduling problems with some form of GPU and PIM performance models. It would be challenging to achieve a meaningful parallelization speedup and justify the complexity if target graphs had few independent nodes.

2. The PIM performance of many convolutional layers does not dominate the GPU performance, and vice versa; parallelization across GPU and PIM can further reduce the critical execution path. Even in graphs with little inter-node parallelism, each node has abundant intra-node data parallelism over input or output tensors. GPU and PIM are specifically designed to exploit such data parallelism to accelerate compute-intensive and memory-intensive operations, respectively. For example, memory-intensive FC layers are an order of magnitude faster on PIM than on GPU [26, 37, 38], while CONV layers often fully utilize GPU cores with cached activations and filters. However, for layers with moderate data reuse and arithmetic intensity, e.g., pointwise CONV layers with deep input/output channels, neither hardware outperforms the other by a crushing margin, i.e., PIM and GPU performance are within a close range. Thus, supporting mixed-parallel execution, i.e., task-parallelism for data-parallel kernels, can achieve a further speedup by overlapping the execution of GPU and PIM kernels.

3. GPU memory can be configured to act as both a regular DRAM and a PIM device to minimize data movement overheads for inter-node parallel execution while maintaining GPU kernel performance. Inter-node parallel execution introduces synchronization and data movement overheads between devices. If devices connected through
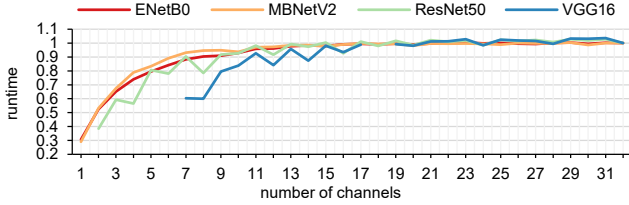
Figure 3. Model inference time on GPU with different number of memory channels, normalized to runtime on GPU with 24 memory channels.

Figure 4. PIM-enabled GPU memory architecture with data movement.

PCIe operate independently, communication overheads can easily offset performance gain from computation acceleration. However, if a part of GPU memory is enabled with PIM compute units, a GPU kernel can run in parallel with a PIM kernel sharing the same physical memory and moving data only between channels. This configuration also simplifies PIM command generation as it does not require a separate device driver for address mapping. Dedicating a subset of channels to PIM could slow down GPU kernels compared to when all channels are accessible to GPU, but our preliminary experiments showed that compute-intensive models are not noticeably impacted, even when the number of memory channels is halved (Fig. 3), due to its high computation-communication ratio. By offloading memory-intensive layers to PIM-enabled channels, this GPU-PIM dual configuration can achieve PIM acceleration without sacrificing GPU performance and increasing DRAM size.

4 Design and Implementation

Guided by the motivating observations in the previous section, we propose PIMFlow, compiler and runtime mechanisms that enable mixed-parallel GPU-PIM acceleration for CNN models. We also introduce a PIM-enabled GPU memory architecture based on [26].

4.1 PIM-Enabled GPU Memory

We extended the DRAM-PIM architecture in prior work [26, 38] to support GPU-PIM parallel execution with minimal overheads and optimize PIM operation latencies.

GPU/PIM memory channel grouping. In order to efficiently execute GPU and PIM workloads on a single memory, independently and in parallel while minimizing software changes, we configure a single DRAM to serve as both GPU memory and PIM device by dividing the memory channels into two contiguous sets: regular channels for GPU data and PIM-enabled channels. There is a trade-off between performance and area/power with regard to the number of PIM-enabled channels in the memory. Having all channels PIM-capable will maximize the PIM computing power, but the added complexity with area and power overheads is hard to justify as many GPU kernels with unsupported operations on PIM cannot use PIM features at all. Thus, we augment only a subset of memory channels with PIM hardware so that we can achieve close-to-ideal PIM acceleration while
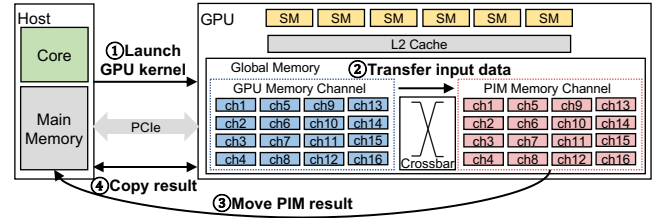
minimizing underutilized PIM resources.

For GPU kernels that do not issue DRAM-PIM commands, the memory behaves the same way as traditional GPU memories. When GPU kernels activate PIM units, offloaded data are mapped to the PIM-enabled memory channels and then used as operands for PIM compute units. The memory controller of the PIM-enabled DRAM is updated to place data in the proper channels and move them between the channels, if needed. Fig. 4 shows how data is transferred between the host memory and distinct sets of channels in the PIM-enabled GPU memory. The data is initially transferred from the host memory to the GPU memory channels. When GPU and PIM kernels are launched to execute in parallel (1), data is moved to the PIM channels before PIM kernels execute (2). Once a PIM kernel finishes its execution, data is transferred back to the host memory for CPU kernels (3) (e.g., activation functions following FC or CONV layers) or back to GPU memory if the data is requested by another kernel (4). We assume all GPU and PIM memory channels are connected to each other forming memory networks [33], since direct memory interconnect incurs much less contention than the GPU L2 cache crossbar, as shown in [63].

PIM command and global buffer extension. The DRAM-PIM hardware supports a set of PIM commands to move data for PIM compute units and activate them; GWRITE and READRES commands push input data to the global buffer and read out computed results respectively, while COMP triggers PIM computation and G_ACT activates multiple banks. These commands are usually issued in the following order: GWRITE-G_ACT-COMP-READRES. We extended the GWRITE command as follows to accelerate CONV operations on PIM more efficiently, while reusing the command semantics as described in [26] for the rest:

- *Multiple global buffers:* We observed that CONV filters are often lowered into small matrices that cannot fully utilize PIM units. As G_ACT fetches the same filter elements from memory to be multiplied by data brought in by different GWRITE commands, we use four global buffers instead of one [26] or two [38] to reuse G_ACTs for higher PIM utilization and data reuse (the command reuse optimization is implemented in the DRAM-PIM back-end). We also added GWRITE_2 and GWRITE_4 commands to access two or four global buffers with a single PIM command.
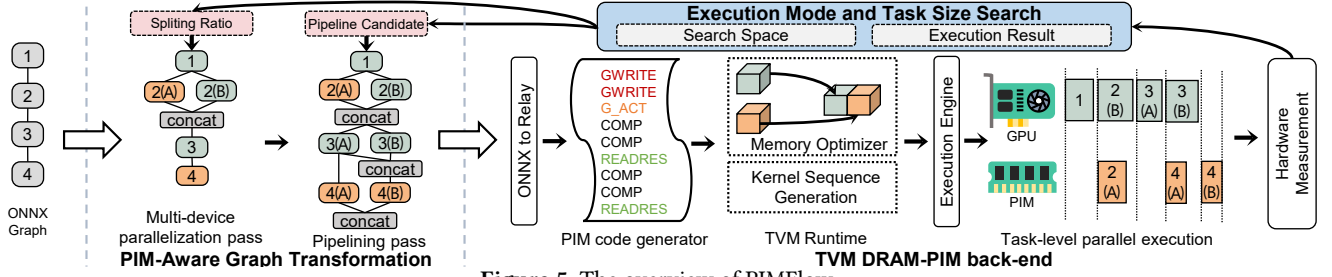
Figure 5. The overview of PIMFlow.

- *Strided GWRITE:* We extend GWRITE to accept three additional arguments so that input tensor elements in non-contiguous memory addresses can be pushed into the global buffer with a single GWRITE command. This command helps simplify PIM command sequences to compute non-pointwise CONV layers.
- *GWRITE latency hiding:* We hide GWRITE latency by asynchronously issuing a following G_ACT command, which significantly reduces PIM cycles. These commands cannot be activated simultaneously when all memory channels are involved in data fetch as in [26], but in our DRAM-PIM architecture with separate GPU and PIM memory channels, data can be fetched from GPU channels while PIM channels activate memory rows.

4.2 PIMFlow Compiler and Runtime Support

PIMFlow is implemented as a compiler and runtime extension in DL frameworks, and consists of three main components as shown in Fig. 5. **PIM-aware graph transformations** identify candidate ONNX graph nodes for PIM acceleration, determine the execution mode, e.g., full/no offloading, multi-device data-parallel, or pipeline-parallel, and transform them accordingly. We use the **execution mode and task size search** engine to search for the optimal execution mode for each node. The resulting ONNX graphs are compiled and executed by the **TVM back-end for DRAM-PIM**. This includes memory and command optimization passes to reduce data communication overheads and improve PIM utilization, as well as a PIM command generation pass.

4.2.1 PIM-aware Graph Transformations.
We extend the current heterogeneous-parallel execution model in DL frameworks to execute independent graph nodes in parallel on GPU and DRAM-PIM. We support the following mixed-parallel execution modes.

- **Multi-Device Data-Parallel (MD-DP) execution:** GPU and PIM kernels execute the same task but with a disjoint portion of input data. Each kernel exploits internal data-parallelism.
- **Pipelined execution:** GPU and PIM kernels with data dependency overlap the execution of their pipeline stages across GPU and PIM. Each pipeline stage kernel exploits internal data-parallelism.

PIMFlow implements two ONNX graph transformation passes that create inter-node parallelism to activate the mixed-parallel execution modes (Fig. 5). For both passes, we assume the PIM candidate nodes to be FC and CONV layers (except for DW CONV), while all the other layers are GPU-executable only. These passes work with the code generator and the execution engine so that proper GPU and DRAM-PIM kernel codes are generated and scheduled to execute in parallel while respecting data dependency.

Multi-device parallelization pass. This pass splits a single PIM-candidate node into two nodes to activate the MD-DP execution mode. For example, node 2 in Fig. 5 is split into 2(A) and 2(B), connected to the producer node of 2. The input data flowing into 2 is sliced into two subsets, each of which becomes input data for 2(A) and 2(B). Lastly, the output data of 2(A) and 2(B) are concatenated to produce a single output tensor equivalent to the original output of 2.

Pipelining pass. This pass takes a subgraph of two or more consecutive nodes and splits each node into multiple pipeline stage nodes to generate inter-node parallelism between pipeline stage nodes processing different data. In Fig. 5, nodes 3 and 4 are pipelined with two pipeline stages. Node 3(A) and 4(B) are prologue and epilogue nodes, which can be executed on GPU or DRAM-PIM, while 3(B) and 4(A) are executed in parallel. The "concat" node before 4(B) is inserted to enforce data dependency for boundary elements from 3(A) when filters are bigger than 1x1. Finally, the outputs of 4(A) and 4(B) are concatenated to produce the subgraph output.

4.2.2 Execution Mode and Task Size Search.
PIMFlow performs a hardware-measurement-based search prior to model compilation to determine which execution mode and task splitting ratio to use for each node in the input graph. To model node performance in the MD-DP mode, PIMFlow generates and profiles samples with different GPU-PIM task splitting ratios (at an interval of 10%) for each PIM-candidate layer (FC and CONV). Thus, the search generates 11 samples for each target layer, including 0/100 and 100/0 ratios for full GPU and DRAM-PIM execution, respectively. PIMFlow uses the multi-device parallelization pass to generate samples with splitting ratios between 10/90 and 90/10, while using the original graph for full GPU or PIM execution.[1]

---

[1] More fine-grained samples with 2% ratio intervals provided a 1.13% speedup for *EfficientNetB0*, so we use 10% ratio intervals for simulation efficiency.

Algorithm 1 Execution Mode and Task Size Search

Require: Graph G
**Ensure:** Nodes in graph G are topologically sorted

1: **function** OPTIMAL_SPLIT(G, node, T)
2:     $best\_runtime \leftarrow \infty$
3:     **for** ratio for every 10% **do**
4:         $runtime \leftarrow do\_split(G, node, ratio)$
5:         **if** $best\_runtime > runtime$ **then**
6:             $best\_runtime \leftarrow runtime$
7:         $T[node.index][1] \leftarrow best\_runtime$
8: **function** PIPELINE(G, node, T)
9:     $next\_node \leftarrow skip\_activation(G, node)$
10:    $l \leftarrow 2$
11:    **while** $next_node.op_type = $ Conv **do**
12:        $runtime \leftarrow do\_pipeline(G, node, l)$
13:        $T[node.index][l] \leftarrow runtime$
14:        $next\_node \leftarrow skip\_activation(G, next\_node)$
15:        $l \leftarrow l + 1$
16: **function** EXECUTION_MODE_SEARCH(G)
17:    $N \leftarrow G.size()$ ▷ The number of nodes in the G
18:    $T \leftarrow map[N][N]$ ▷ Allocate table T
19:    **for** node in G **do**
20:        **if** $node.op_type = $ Conv **then**
21:            OPTIMAL_SPLIT(G, node, T)
22:            PIPELINE(G, node, T)
23:    **for** $l \leftarrow 1$ to N **do**   ▷ Solve by dynamic programming
24:        **for** $i \leftarrow 1$ to N **do**
25:            **for** $k \leftarrow 1$ to $l - 1$ **do**
26:                **if** $i + k > N$ **then**
27:                    continue
28:                $T[i][l] \leftarrow \min(T[i][l], T[i][k] + T[i+k][l-k])$
29:    **return** $T[1][N]$

PIMFlow also recursively traverses the model graph to identify all pipelining candidate subgraphs and executes the pipelining pass in Section 4.2.1 to transform them. We extract promising subgraphs with a PIM-candidate node followed or preceded by a non-PIM node (unsupported operator) from the graph. Subgraphs with non-PIM nodes that are computationally very lightweight, e.g., element-wise multiplication/addition and max pooling, compared to CONV and FC layers, or that have a data-flow dependency with multiple nodes that make pipelining impossible or very complicated, are excluded from candidates, since the cost of pipelining in these cases is likely to exceed the potential performance gain. We analyzed the model architectures of CNN models and identified a sequence of 1x1 and DW CONV layers as a frequent and promising subgraph pattern. These layers often follow each other and have similar arithmetic intensity, but DW CONV layers are only executed on GPU because it is not straightforward to offload DW CONV to current DRAM-PIM as it requires the global buffer to be flushed for each input channel.

Algorithm 1 illustrates the overall search process. The algorithm searches for optimal runtime at 10% ratio intervals (lines 3-6), and measures all pipelining candidates starting from each node and expanding subgraphs one by one (lines 11-15). The above search results are recorded in T (lines 21-22), and we compute an optimal solution with dynamic programming (lines 23-28).
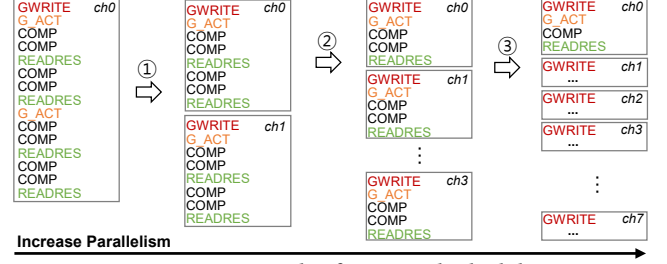
Figure 6. An example of command scheduling.
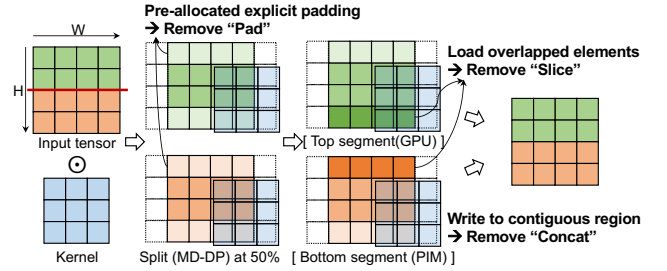
Figure 7. An example of memory optimization.

4.3 TVM Back-End for DRAM-PIM

TVM back-end for DRAM-PIM implements two optimization passes to increase command-level parallelism and reduce overheads. It also includes a code generator that generates DRAM-PIM commands for offloaded PIM nodes.

4.3.1 DRAM-PIM Command Generator. The DRAM-PIM back-end is designed to take transformed model graphs as input, generate PIM commands for PIM-offloaded nodes, and execute GPU and PIM kernels in parallel. We mark PIM-offloaded nodes by prefixing the node names and passing them as Relay IR attribute to trigger the DRAM back-end. We extend the TVM execution engine to launch GPU and PIM kernels in parallel while reusing the existing TVM mapping for GPU nodes to cuDNN, cuBLAS, or CUTLASS library calls.

The command generator includes a command scheduling pass to distribute PIM commands across channels to fully utilize all PIM compute units. This prevents channels from being idle when matrices to be placed in memory are too small, which is often the case for 1x1 CONV layers. Our scheduling mechanism distributes PIM commands at G_ACT (1), READRES (2), and COMP (3) granularity as shown in Fig. 6, which progressively increases channel-level parallelism.

4.3.2 Memory Optimizer. Splitting and pipelining nodes require input tensors to be sliced. When the kernel width or height is greater than one, additional input tensor elements around the splitting edges are needed to convolve with the kernel elements. Thus, we need an extra "Pad" operator before each input tensor. Also, after nodes are executed in the MD-DP mode, computation results on GPU and DRAM-PIM must be combined into a single tensor as input for subsequent layers. This requires an additional "Concat" operator to join split kernels. We found that "Slice", "Pad", and "Concat" operators incur significant data copy overheads, making most splitting attempts futile.
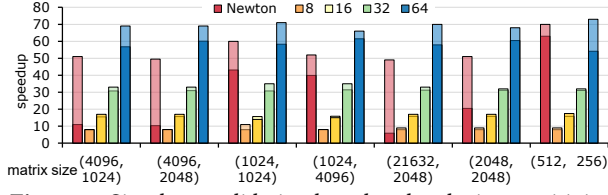
Figure 8. Simulator validation based on batch size sensitivity.

Therefore, we devised a memory layout optimization to eliminate the overheads. We assume the memory layout for CONV layers in NHWC format, and inference is done in a single batch size. As shown in Fig. 7, slicing/concatenating tensors in the input height dimension (H) is a no-op if two split input/output tensors are located in contiguous memory space. Additionally, if we pre-allocate the memory space of the size of the input tensor plus padding and initialize them to 0, we can eliminate the "Pad" operator by writing input data from a padding offset. Thus, during code generation, we allocate contiguous memory space including padding for input/output of CONV layers.

5 Methodology

We implemented ONNX graph transformation passes by using ONNX opset version 13 [1], and extended the TVM compiler version 0.8 (commit 7e376e2) [11] to support DRAM-PIM as a back-end using the Bring-Your-Own-Codegen (BYOC) interface [12, 61]. GPU kernels are based on CUDA 11.3.1 and cuDNN 8.2 libraries. PIMFlow allows an individual application to choose whether to enable the PIM capability in the GPU memory. If disabled, applications are compiled and executed to use all memory channels as regular load/store units for GPU. The execution mode and task size search phase is executed once per kernel prior to compilation to decide the execution mode, and search results are stored as a metadata log file for later compilations. We implemented the memory optimizer by modifying codes to lay out input matrices for cuDNN library calls and PIM kernels so that their addresses are contiguous when split. We plan to move the implementation to the compiler back-end and automate the memory address generation.

Evaluated models. We evaluated the single-batch inference time of five CNN models: *EfficientNetB0* (*ENetB0*) [57], *MnasNet* [56], *MobileNetV2* (*MBNetV2*) [51], *ResNet50* [25], and *VGG16* [53]. PIM candidate layers in these models include 1x1, 3x3, 5x5, and 7x7 CONV layers and FC layers. For pipelined execution, we used three subgraph patterns: 1x1-DW, DW-1x1, and 1x1-DW-1x1, where DW layers are executed on GPU while 1x1 layers are on DRAM-PIM.

GPU and DRAM-PIM simulators. We implemented a simulator for the Newton-based DRAM-PIM architecture by extending Ramulator (commit 4edcb0d) [34]. We modified the DRAM controller to process both GPU memory commands and PIM commands. We set the latency of the PIM commands based on parameter descriptions in [26], and adapted them for GDDR6 DRAM, as shown in Table 1. TVM DRAM-PIM

Table 1. DRAM configuration

Num of Ranks	1	Num of Column I/Os per row	32
Num of Banks	16	Column I/O bit width	256b
Global buffer size	4 KB	Num of Multipliers per bank	16
Timing Parameters (in clock cycles)			
t_{BL}: 2, t_{CL}: 11, t_{RP}: 11, t_{RCD}: 11, t_{CCD}: 2, t_{RAS}: 25			

back-end interfaces with this simulator to generate PIM command traces for PIM-offloaded layers and measures the trace execution time. We used Accel–Sim [32] (commit 000be7f) to generate and simulate GPU traces on NVIDIA GeForce RTX 2060 GPU. We enabled the "write-through" mode for GPU caches, which guarantees data coherence at the memory level for PIM commands and memory accesses from GPU.[2] We used AccelWatch [30] integrated with Accel–Sim to measure GPU energy consumption, and CACTI 7 [4] with the energy parameters adapted from [54] to measure PIM energy consumption.

Simulator validation. We validated our simulator using the matrix-vector kernel benchmarks evaluated in [26]. Fig. 8 is a reproduced version of Fig. 12 in [26] on our simulator, comparing the PIM and GPU performance with different batch sizes. We matched software and hardware configurations to the best of our knowledge; we used the HBM timing parameters based on [26] and CUTLASS v1.3 [31] for GPU kernels and NVIDIA Titan V GPU with 24 memory channels for GPU configurations. The experiment shows that our simulator performs 20.4x better than GPU (batch size = 1), which is a conservative speedup compared to 50x in [26] but closer to 10x in follow-up research [38]. While PIM performance scales consistently with matrix sizes, we found that GPU kernels show widely varying behaviors, depending on matrix sizes and library versions. Considering the inherently limited capacity for validation against a proprietary architecture, we focus on simulating a realistic version of PIM-enabled memory and using a widely-used cuDNN library for a fair evaluation.

Evaluated PIM offloading mechanisms.

- *Baseline*: GPU-only execution with a 32-channel memory.
- *Newton+*: The baseline Newton [26] with offloading support for CONV and FC layers (no mixed-parallel execution support) and command scheduling for multiple channels.
- *Newton++*: *Newton+* with the PIM command optimizations.
- *PIMFlow-md* and *PIMFlow-pl*: *Newton++* with mixed-parallel execution support for MD-DP (*PIMFlow-md*) and pipelining only (*PIMFlow-pl*).
- *PIMFlow*: PIMFlow with full optimizations and execution model support as proposed in Section 4.

6 Evaluation Results

In this section, we show how much performance and energy improvement *PIMFlow* can achieve by enabling CNN models on PIM-enabled GPU memory. We also present sensitivity

---

[2]We observed a 2.8% slowdown compared to the "write-back" mode (MobileNet), which may impact offloading decisions and splitting ratios but is tolerable considering the overall performance gain from PIM acceleration.
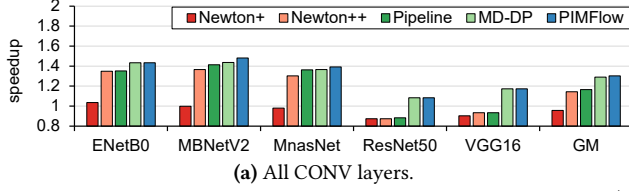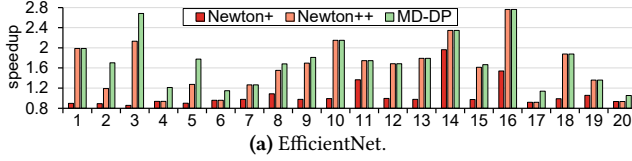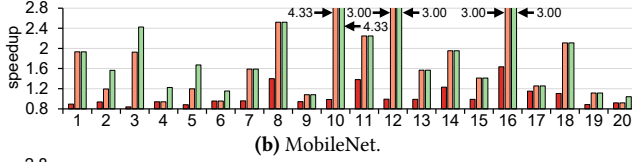
(a) All CONV layers.
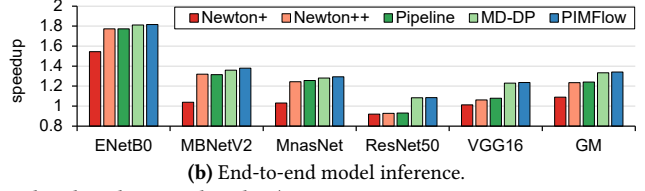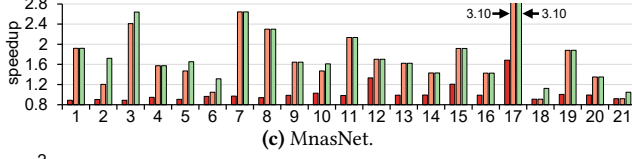

(b) End-to-end model inference.

Figure 9. Execution time (normalized to the GPU baseline).
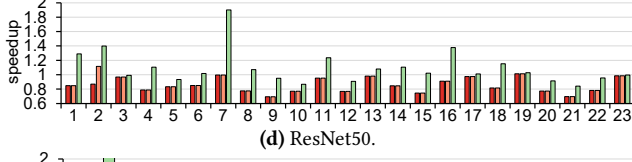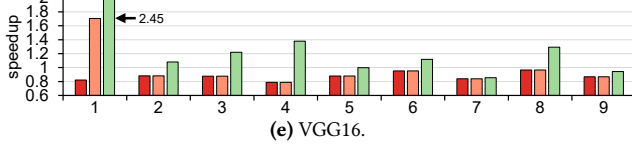

(a) EfficientNet.

(b) MobileNet.

(c) MnasNet.

(d) ResNet50.

(e) VGG16.

Figure 10. Layerwise performance breakdown for nodes executed in the MD-DP mode (normalized to the GPU baseline).

analysis results with different memory configurations, software parameters, and model sizes/types, to investigate the feasibility and performance impact of our design decisions.

6.1 CNN Model Performance

We measured the inference time of five CNN models compiled by *PIMFlow* and the other offloading mechanisms on simulated hardware configurations as listed in Section 5. For stable results, each simulation is repeated three times.

Fig. 9 presents (1) the execution time of all PIM-candidate CONV layers and (2) end-to-end model inference time for all evaluated models, normalized to the GPU baseline. In summary, *PIMFlow* with full MD-DP and pipelined execution support provides a 30% speedup on average for CONV layers against the GPU baseline, while outperforming all of the other offloading mechanisms. The performance gain is more significant – up to 48% with *ENetB0*, *MBNetV2*, and *MnasNet* than *ResNet50* and *VGG16* – since more compute-intensive CONV layers in the latter models do not provide as much speedup on DRAM-PIM as the former models. For
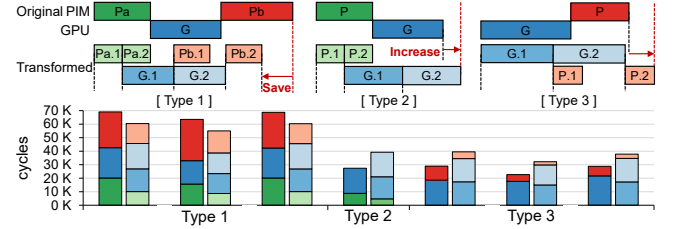

Figure 11. Layerwise performance breakdown for pipelining candidate subgraphs (left bars for nodes when executed in the MD-DP mode, right bars when pipelined).
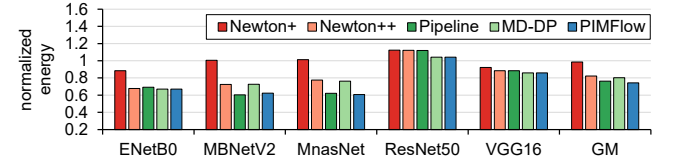

Figure 12. Energy consumption.

Table 2. The distribution of MD-DP splitting ratios

Split ratio to GPU (0: total offload)										
0	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**
41%	5%	6%	9%	3%	10%	5%	6%	8%	6%	0%

end-to-end inference time, *ENetB0* and *VGG16* gain 27% and 5% additional speedups on top of 43% and 17% for CONV layers, respectively, by offloading FC layers to the DRAM-PIM. Overall, the results show that compiler and runtime support by *PIMFlow* can enable CNN models on the DRAM-PIM hardware with substantial performance gain for both FC and CONV layers without impacting the hardware.

Newton+ vs. Newton++. *Newton++* outperforms *Newton+* by 20% on average for all CONV layers (up to 37% for *MBNetV2*), showing that optimizing PIM commands alone can boost PIM capabilities (more details in Section 6.2).

Newton++ vs. PIMFlow-md. Comparing the layerwise and modelwise execution times of *Newton++* and *PIMFlow-md* highlights the performance improvement enabled by the MD-DP execution mode (Fig. 9 and 10). The splitting ratio between GPU and DRAM-PIM varies depending on how memory-intensive a layer is and how it performs on GPU. Table 2 shows that 58% of the PIM-candidate layers are split across GPU and DRAM-PIM and executed in parallel, while 41% fully offload to DRAM-PIM (some of the layers were considered for pipelined execution as well, but executed in parallel based on search results). We observed that the MD-DP execution mode enables many layers that were not offloading candidates for *Newton++* to gain speedups with parallel execution on GPU and DRAM-PIM (e.g., layers 4, 6, 17, and 20 in *ENetB0*), resulting in a 13% speedup against
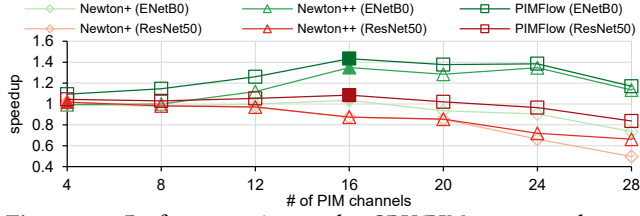
Figure 13. Performance impact by GPU/PIM memory channel ratios (32 in total). Filled shapes indicate the best performance for the given mechanism.
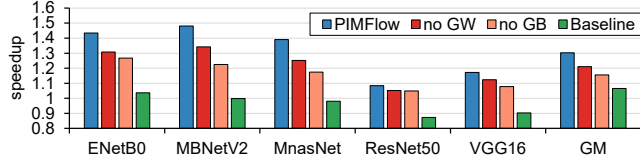

Figure 14. PIM-command optimization sensitivity (GW for GWRITE latency hiding and GB for multiple global buffers).
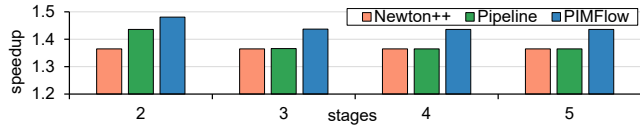

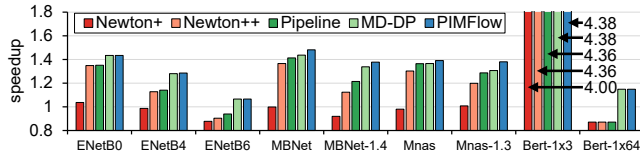**Figure 15.** Pipeline stage granularity sensitivity.

Figure 16. Model type and size sensitivity.

Newton++, on average. These are the target CONV layers identified in Section 3 whose performance on DRAM-PIM is not significantly better or worse than GPU. While full PIM offloading could also make a difference for these layers, parallel execution further reduced the critical execution path length with minimal parallelization overhead.

PIMFlow-md vs. PIMFlow-pl. In our current implementation, the MD-DP execution mode identifies all CONV layers except for DW as candidates while the pipelined execution mode uses pre-defined subgraph patterns to find a limited set of candidate layers. Their candidate layers partially overlap, so we analyzed how *PIMFlow-md* and *PIMFlow-pl* perform comparatively for the *PIMFlow-pl* patterns with a layerwise and stagewise breakdown. Fig. 11 presents pipelined subgraphs that show >10% speedup or <25% slowdown with *PIMFlow-pl* compared to *PIMFlow-md* in all evaluated models. We used three subgraph patterns for pipelining as shown on the top of Fig. 11, and only the Type 1 pattern outperforms the same nodes parallelized in MD-DP mode. This pattern can effectively overlap GPU kernels (G.1 and G.2) with PIM kernels (Pa.2 and Pb.1), while prologue and epilogue GPU kernel latencies are too high in Type 2 and 3 patterns.

PIMFlow-md/PIMFlow-pl vs. PIMFlow. *PIMFlow* with full MD-DP and pipelining support outperforms the *PIMFlow-md* and *PIMFlow-pl* variations by 1% and 12% on average, respectively. For *ENetB0*, *MBNetV2*, and *MnasNet*, *PIMFlow-md* and

PIMFlow-pl separately achieve significant speedups: 38% and 41% on average against the baseline (Fig. 9). When combined, *PIMFlow* improves the inference time by 43% on average for these models. The speedup numbers do not strictly add up due to having common candidate layers, as described above. For *ResNet50* and *VGG16* with a few to zero pipelining pattern matches, *PIMFlow* performs the same as *PIMFlow-md*.

Energy consumption. Fig. 12 shows that both *Newton++* and *PIMFlow* consume significantly less energy than the GPU baseline by 18% and 26%, respectively. The result aligns with prior work that the fixed-function MAC logic in memory requires less power for the same computation than dense GPU cores and additionally saves power by reducing data transfers [3, 15, 26]. *Newton++* uses 17% less energy than *Newton+*, as the PIM-command optimizations reduces bank activation. The models with relatively small speedup on *PIM-Flow* (*Resnet50* and *VGG16*) show limited or negative energy gains due to the increased GPU static power for compute-intensive layers.

6.2 Sensitivity Study

GPU/PIM memory channel ratio. We experimented with different ratios of GPU-only and PIM-enabled memory channels to investigate how they impact overall model performance. Fig. 13 shows that as the number of PIM channels increases (and the number of GPU channels decreases) in a 32-channel memory, *PIMFlow* consistently improves model performance thanks to PIM acceleration, up to a certain point (16 PIM channels), then slows down as GPU kernel performance suffers from too few memory channels. The 16-16 channel division in our PIM-enabled GPU memory is derived from this experimental result. The positive impact of PIM acceleration is weaker and the negative impact of reduced GPU memory is more severe for *Newton+* and *Newton++*, especially for *ResNet50* with more compute-intensive layers than *ENetB0*, which reaffirms the contribution of *PIMFlow* to extending the scope of PIM utilization.

PIM-command optimizations. We isolated the performance impact of two PIM-command optimizations, GWRITE latency hiding and multiple global buffers. Fig. 14 shows that GWRITE latency hiding provides a 9% speedup on its own while multiple global buffers provide a 14% speedup against *Newton+*. *PIMFlow* achieves a 22% speedup on average by combining them; thus, we see that neither optimization absorbs or interferes with the effect of the other and independently contributes to the performance.

Pipeline stage granularity. Increasing the number of pipeline stages will reduce prologue and epilogue overheads, but increase kernel launch and communication overheads. We examined how the number of pipeline stages may impact performance, and found that having more than two stages leads to larger overheads than the performance gain from overlapped execution (Fig. 15).

Model type and size. We evaluated BERT [16] and scaled-up versions of *ENet*, *MBNetV2*, and *MnasNet*) to see how *PIMFlow* features work for DNN models of different types and sizes. For BERT, while *PIMFlow* performs the same as *Newton++* for small input (1x3) as used in [26], *PIMFlow* provides a 32% extra speedup over *Newton++* for 1x64 input by executing FC layers in the MD-DP mode. The *PIMFlow* acceleration for the mobile CNN models decreases as the model size increases, going down to 7% for *ENetB6* against the GPU baseline. This is because even with 1x1 CONV layers, the arithmetic intensity and data reuse increase, favoring GPU execution over PIM as the model size grows.

7 Discussion

Area overhead. The baseline DRAM-PIM architecture reported the area overhead for PIM-related logic to be 0.19 mm^2 [38] per bank. Our PIM-enabled memory introduced additional area overheads for a crossbar interconnect between channels and larger global buffers (4KB per channel). We estimate the area of global buffers to be 0.33 mm^2 based on [4], and the area of the crossbar and long links to be 1.53 mm^2 in 32-channel memory referring to [63], which are around 0.72% of the GPU die area in total.

Contention at memory controller. The GPU memory controller is extended to handle PIM commands too, which can increase contention at the controller. While a PIM channel reads activation data from GPU channels, the GPU memory controller cannot accept GPU memory commands. We simulated this contention by interleaving Accel–Sim memory commands with PIM command sequences to the DRAM-PIM simulator, and the slowdown due to the contention was negligible at 0.15% for *MBNetV2* and 0.22% for *Resnet50* compared to no-contention cases.

Compilation overhead. Compilation time is dominated by the hardware measurement time during the execution mode and task size search phase, which is proportional to the number of FC and CONV layers in a model. Though profiling on our simulators takes several hours, measurements on actual hardware are expected to finish within several minutes.

8 Related Work

Digital Processing-in-Memory DRAM. Many researchers have proposed PIM architectures for DL model acceleration [14, 15, 20, 26, 37, 52, 60]. [15] performed matrix multiplication in a systolic-array accelerator per MAC bank, which takes activations from Broadcast bank. [14] observed that small batch sizes result in memory-bound matrix operations during model inference, and processed these operations near memory while handling complex memory address mapping. [26, 38] accelerated matrix-vector multiplication with in-memory MAC logic and global buffers, while [37] introduced an HBM-based PIM architecture that supports MAC and elementwise operations at the bank level. Recent work tends to limit PIM compute capability to MAC or ele-

mentwise computations considering hardware constraints. PIMFlow is designed with such PIM architectures in mind, and thus it can be readily adapted to support them.

Software support for PIM hardware. To make PIM hardware accessible and fully utilized, it is important to provide software interfaces to mitigate the burden on programmers. [43] provided compiler support to generate PIM binaries, but required programmers to annotate programs with directives to identify PIM computations. [22, 58] built a cost model using profiling results and decided which computation to offload to PIM, while [62] profiled a subset of loop iterations to dynamically offload loops to PIM. [27, 33] implemented a software stack for GPU-PIM systems that uses a cost-benefit analysis of memory bandwidth consumption for offloading decisions. [59] allowed programmers to restructure target programs to optimize memory access patterns for PIM using a runtime memory traffic monitor. While these efforts could automate the offloading process, they did not proactively transform programs to generate more offloading opportunities as PIMFlow does.

Placement and scheduling mechanisms for heterogeneous systems. Scheduling workloads on heterogeneous systems is an open problem with intractable complexity [8, 29, 39, 41, 46, 48, 55]. [48] proposed a way to utilize the GPU with PIM-enabled memory. It analyzed GPU kernels by using metrics such as memory intensity and scheduled each kernel concurrently on both GPU and PIM cores. [41] proposed a hierarchical task scheduler for heterogeneous systems that divided a coarse-grained task into subtasks and scheduled them on accelerators. On CPU-GPU heterogeneous systems, [29] used online profiling to reduce load imbalance and increase parallelism for scheduled workloads across devices. While PIMFlow shares the goal of maximizing parallelism on heterogeneous systems with prior work, we focus specifically on PIM-aware compiler transformations and runtime support to maximize PIM utilization.

9 Conclusion and Future Work

This paper presented our effort to expand the scope of PIM acceleration to CNN models, widely used as a backbone but out of the spotlight for PIM target applications. With the PIM-aware graph transformations, command-generating back-end, and mixed-parallel execution runtime specifically designed to accelerate convolutional layers on DRAM-PIM, PIMFlow showed strong potential for an optimizing software stack for commercial DRAM-PIM. For future work, we plan to apply an auto-tuning approach to our execution mode and task size search for more optimized code generation.

Data-Availability Statement

The data that support the findings of this study are openly available in Zenodo (DOI: 10.5281/zenodo.7548467) [2].

A Artifact Appendix

A.1 Abstract

Our artifact consists of four parts: (1) ONNX transformation passes, (2) hardware measurement scripts for the execution mode and task size search, (3) an extended TVM compiler with DRAM-PIM back-end, and (4) GPU and DRAM-PIM simulators. For pre-generated input data, we provide GPU traces for the CNN models evaluated in the paper. These traces can also be generated by using NVBit.[3] We provide modified binaries and source codes for the TVM compiler extended with the DRAM-PIM back-end.

Artifact evaluation and testing are streamlined with a top-level script (`pimflow`) that controls different features of PIMFlow with lower-level scripts (details in Section A.5). The reproduction of the results in this paper can be conducted on any platform that can run the simulators, as long as all traces are generated and executed with the same simulator configurations as the paper.

A.2 Artifact Check-list (Meta-information)

- **Algorithm:** Methods for (1) PIM-aware graph transformation passes to enable MD-DP (Multi-Device Data-Parallel) and pipelined execution of CONV layers, (2) layer-wise profiling of MD-DP and pipelining candidates on GPU and DRAM-PIM, (3) dynamic-programming based algorithm to obtain the optimal execution mode and task size, and (4) code-generating back-end for DRAM-PIM.
- **Program (Model):** *ENetB0*, *MBNetV2*, *MnasNet*, *ResNet50*, and *VGG16* as provided by Torchvision[4], and *Toy* is a small, in-house model with three CONV layers for quick testing.
- **Compilation:** Modified Apache TVM compiler based on v0.8 (commit: 7e376e2). Binaries and sources provided.
- **Data set:** GPU traces produced by GPGPU-Sim for the evaluated models are included. Traces can be re-generated by using NVBit, but the results may vary slightly due to inherently non-deterministic behaviors of kernel codes.
- **Run-time environment:** Linux (Ubuntu 20.04) with CUDA 11.3.1 and cuDNN 8 runtime for AMD64.
- **Hardware:** We recommend systems with NVIDIA GeForce RTX 2080 Ti GPU for reproducing the results in the paper. GPUs with the same architecture (Turing) should give comparable results.
- **Metrics:** Model inference time (GPU kernels).
- **Output:** Given a CNN model, its ONNX graph transformed by the PIM-aware graph transformation passes, GPU traces generated by NVBit, and DRAM-PIM kernels generated by TVM back-end.

- **Experiments:** Execution steps are described in A.5 and README at our archive.
- **How much disk space required (approximately)?:** Up to 65GB, including five model traces (15GB for TVM, Accel-SIM, GPGPU-Sim, Ramulator, and PIMFlow installation, and 10GB trace per model).
- **How much time is needed to prepare workflow (approximately)?:** Re-generating GPU traces will take up to 12 hours per model on systems with 8 NVIDIA GeForce RTX 2080 Ti GPUs.
- **How much time is needed to complete experiments (approximately)?:** Up to 8 hours of simulation time per model on Intel Xeon Gold 6248R CPU.
- **Publicly available?:** Yes.
- **Workflow framework used?:** No.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.7376801

A.3 Description

A.3.1 How Delivered. The PIMFlow compiler (ONNX passes and TVM back-end), GPU and DRAM-PIM simulators, and hardware measurement engine are available at the following link: https://doi.org/10.5281/zenodo.7376801 (DOI: 10.5281/zenodo.7376801). We also provide a GitHub repository (https://github.com/yongwonshin/PIMFlow).

A.3.2 Hardware Dependencies. We recommend systems with NVIDIA GeForce RTX 2080 Ti GPU for GPU trace generation. GPU with the same architecture (Turing) should give similar results. There is no additional hardware dependency for simulation execution (Accel-Sim and Ramulator) except for the executable platform requirement in their original versions.

A.3.3 Software Dependencies. We implemented and tested our codes on the Ubuntu 20.04 x86-64 system with CUDA 11.3.1 and cuDNN 8 library. Additional software dependencies include minimal prerequisites on Ubuntu for TVM, Accel-Sim, GPGPU-Sim, and Ramulator. The minimal versions required are Python v3.8, PyTorch version 1.11.0+cu113, Torchvision version 0.12.0+cu113, LLVM 13 including Clang, and ONNX operator set version 13. We strongly recommend using a docker image, "yongwonshin/pimflow:v0.1" or a later version, for installation. The docker image includes all software dependencies and prerequisites for experiments.

A.4 Installation

A.4.1 Docker Installation. *docker* and *nvidia-container-docker* packages are required in order to run our docker image. The following command will activate the image and install PIMFlow.

---

[3]https://github.com/NVlabs/NVBit
[4]https://pytorch.org/vision/stable/models.html

```
# execute docker container
docker run -it --gpus=all --rm \
  yongwonshin/pimflow:v0.1
# install prerequisites
./install.sh
```

A.4.2 Local Installation (Zenodo). The Zenodo archive provides Git patch files for TVM, Accel-Sim, GPGPUSIM, and Ramulator, and a compressed PIMFlow repository that includes all execution and profiling scripts, ONNX transformation passes, and GPU traces. Git patch files can be applied to each repository by using the following command:

```
# from commit: 000be7f
git am PIMFlow_accel-sim-framework.patch
# from commit: 13c6711
git am PIMFlow_gpgpu-sim_distribution.patch
# from commit: 4edcb0d
git am PIMFlow_ramulator.patch
# from commit: 7e376e2
git am PIMFlow_tvm.patch
```

The PIMFlow repository can be extracted by using the following command:

```
unzip PIMFlow_93035b5.zip
```

We also included DockerFile for manual docker build. The README at the archive has detailed installation instructions.

A.4.3 Local Installation (GitHub). The following four GitHub repositories must be cloned in order to compile and run PIMFlow:

```
https://github.com/yongwonshin/PIMFlow_tvm.git
https://github.com/yongwonshin/PIMFlow_accel-sim-
framework.git
https://github.com/yongwonshin/PIMFlow_gpgpu-sim
_distribution.git
https://github.com/yongwonshin/PIMFlow_ramulator.git
https://github.com/yongwonshin/PIMFlow.git
```

An installation guide is included in each repository, and an overall environment setup guide for evaluation is provided by the README file in the PIMFlow GitHub repository.

A.5 Experiment Workflow

The overall experiment workflow consists of three steps: candidate profiling, candidate search and selection, and end-to-end execution. You can jump to Step 2 if you reuse previously profiled data, and to Step 3 if you have already computed the optimal graph. We use the *Toy* network for the following installation sequence. The <net> option can be *efficientnet-v1-b0*, *mobilenet-v2*, *mnasnet-1.0*, *resnet-50*, or *vgg-16*.

Step 1: Profile each CONV layer using MD-DP or pipelining transformation pass.

```
./pimflow -m=profile -t=split -n=<net>
./pimflow -m=profile -t=pipeline -n=<net>
```

Step 2: Compute the optimal ONNX graph based on the result from Step 1.

```
./pimflow -m=solve -n=<net>
```

Step 3: Execute the transformed model. Use the `--gpu_only` option for GPU execution time.

```
./pimflow -m=run --gpu_only -n=<net>
./pimflow -m=run -n=<net>
```

A.6 Evaluation and Expected Result

After steps 1 and 2, the resulting ONNX graph and GPU/DRAM-PIM traces for the graph are created in `PIMFlow`. Also, profiling results are saved in `PIMFlow/layerwise` and `PIMFlow/pipeline` for MD-DP and pipelined executions, respectively. Executing the following command will run the traces and generate an execution time graph for all PIM-candidate CONV layers with four offloading mechanisms (Fig. 17). The *<policy>* option can be *Newton+*, *Newton++*, *Pipeline*, *MDDP*, or *PIMFlow*.

```
# Convolution-only result
./pimflow -m=stat --conv_only -n=<net>
# End-to-end result
./pimflow -m=stat -n=<net> --policy=<policy>
```
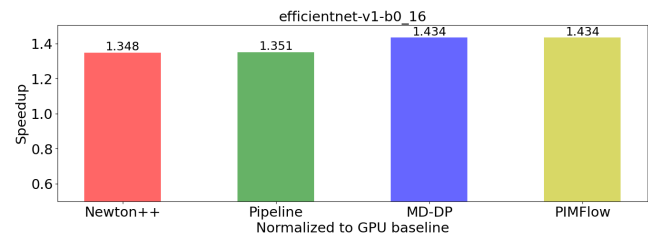

Figure 17. Example output graph (normalized to the GPU baseline).

A.7 Experiment Customization

Profiling scripts are customizable with MD-DP split ratios and pipeline patterns that are different from what is used in the paper. The number of memory channels used for DRAM-PIM hardware can also be customizable for hardware design space exploration. The main execution script can take as input other CNN/DNN models that were not evaluated in the paper and optimize them with PIMFlow.

References

[1] 2019. Open Neural Network Exchange. https://onnx.ai/.
[2] 2022. PIMFlow: Compiler and Runtime Support for CNN Models on Processing-in-Memory DRAM. https://doi.org/10.5281/zenodo.7548467.
[3] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 336–348. https://doi.org/10.1145/2749469.2750385
[4] Rajeev Balasubramanian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Trans. Archit. Code Optim.* 14, 2, Article 14 (jun 2017), 25 pages. https://doi.org/10.1145/3085572
[5] Amirali Boroumand, Saugata Ghose, Berkin Akin, Ravi Narayanaswami, Geraldo F. Oliveira, Xiaoyu Ma, Eric Shiu, and Onur Mutlu. 2021. Google Neural Network Models for Edge Devices: Analyzing and Mitigating Machine Learning Inference Bottlenecks. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 159–172. https://doi.org/10.1109/PACT52795.2021.00019
[6] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu.

2018. Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 316–331. https://doi.org/10.1145/3173162.3173177

[7] E. O. Brigham and R. E. Morrow. 1967. The fast Fourier transform. *IEEE Spectrum* 4, 12 (1967), 63–70. https://doi.org/10.1109/MSPEC.1967.5217220

[8] Louis-Claude Canon, Loris Marchal, Bertrand Simon, and Frédéric Vivien. 2020. Online Scheduling of Task Graphs on Heterogeneous Platforms. *IEEE Transactions on Parallel and Distributed Systems* 31, 3 (2020), 721–732. https://doi.org/10.1109/TPDS.2019.2942909

[9] Kevin K. Chang. 2017. Understanding and Improving the Latency of DRAM-Based Memory Systems. https://doi.org/10.48550/ARXIV.1712.08304

[10] Kevin K. Chang, Abhijith Kashyap, Hasan Hassan, Saugata Ghose, Kevin Hsieh, Donghyuk Lee, Tianshi Li, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. 2016. Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization. *SIGMETRICS Perform. Eval. Rev.* 44, 1 (jun 2016), 323–336. https://doi.org/10.1145/2964791.2901453

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*.

[12] Zhi Chen, Cody Hao Yu, Trevor Morris, Jorn Tuyls, Yi-Hsiang Lai, Jared Roesch, Elliott Delaye, Vin Sharma, and Yida Wang. 2021. Bring Your Own Codegen to Deep Learning Compiler. *CoRR* abs/2105.03215 (2021).

[13] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[14] Benjamin Y. Cho, Jeageun Jung, and Mattan Erez. 2021. Accelerating Bandwidth-Bound Deep Learning Inference with Main-Memory Accelerators. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 44, 14 pages.

[15] Seunghwan Cho, Haerang Choi, Eunhyeok Park, Hyunsung Shin, and Sungjoo Yoo. 2020. McDRAM v2: In-Dynamic Random Access Memory Systolic Array Accelerator to Address the Large Model Problem in Deep Neural Networks on the Edge. *IEEE Access* (2020). https://doi.org/10.1109/ACCESS.2020.3011265

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805

[17] Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steele, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen, Chang Woo Kang, Ihn Kim, and Gokhan Daglikoca. 2002. The Architecture of the DIVA Processing-in-Memory Chip. https://doi.org/10.1145/514191.514197

[18] M. Gokhale, B. Holmes, and K. Iobst. 1995. Processing in memory: the Terasys massively parallel PIM array. *Computer* (1995). https://doi.org/10.1109/2.375174

[19] Gene H Golub and Charles F Van Loan. 2013. *Matrix computations*. JHU press.

[20] Peng Gu, Xinfeng Xie, Shuangchen Li, Dimin Niu, Hongzhong Zheng, Krishna T Malladi, and Yuan Xie. 2020. DLUX: A LUT-based near-bank accelerator for data center deep learning training workloads. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40, 8 (2020), 1586–1599.

[21] Daniel Hackenberg, Guido Juckeland, and Holger Brunst. 2012. Performance analysis of multi-level parallelism: inter-node, intra-node and hardware accelerators. *Concurrency and Computation: Practice and Experience* 24, 1 (2012), 62–72.

[22] Ramyad Hadidi, Lifeng Nai, Hyojong Kim, and Hyesoon Kim. 2017. CAIRO: A compiler-assisted technique for enabling instruction-level

offloading of processing-in-memory. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 4 (2017), 1–25.

[23] Nastaran Hajinazar, Geraldo F. Oliveira, Sven Gregorio, João Dinis Ferreira, Nika Mansouri Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gómez-Luna, and Onur Mutlu. 2021. SIMDRAM: A Framework for Bit-Serial SIMD Processing Using DRAM. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 329–345. https://doi.org/10.1145/3445814.3446749

[24] Hyungkyu Ham, Hyunuk Cho, Minjae Kim, Jueon Park, Jeongmin Hong, Hyojin Sung, Eunhyeok Park, Euicheol Lim, and Gwangsun Kim. 2021. Near-Data Processing in Memory Expander for DNN Acceleration on GPUs. *IEEE Computer Architecture Letters* 20, 2 (2021), 171–174. https://doi.org/10.1109/LCA.2021.3126450

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*. https://doi.org/10.1109/CVPR.2016.90

[26] Mingxuan He, Choungki Song, Ilkon Kim, Chunseok Jeong, Seho Kim, Il Park, Mithuna Thottethodi, and T. N. Vijaykumar. 2020. Newton: A DRAM-maker's Accelerator-in-Memory (AiM) Architecture for Machine Learning. In *Proc. ACM/IEEE 48th Annu. Int. Symp. Microarchit.* https://doi.org/10.1109/MICRO50266.2020.00040

[27] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W. Keckler. 2016. Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent near-Data Processing in GPU Systems. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA '16)*. IEEE Press, 204–216. https://doi.org/10.1109/ISCA.2016.27

[28] Yuan-Ting Hu, Jia-Bin Huang, and Alexander Schwing. 2017. Maskrnn: Instance level video object segmentation. *Advances in neural information processing systems* 30 (2017).

[29] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Chunling Hu, Brian T. Lewis, and Keshav Pingali. 2014. Adaptive heterogeneous scheduling for integrated GPUs. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 151–162. https://doi.org/10.1145/2628071.2628088

[30] Vijay Kandiah, Scott Peverelle, Mahmoud Khairy, Junrui Pan, Amogh Manjunath, Timothy G. Rogers, Tor M. Aamodt, and Nikos Hardavellas. 2021. AccelWattch: A Power Modeling Framework for Modern GPUs. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 738–753. https://doi.org/10.1145/3466752.3480063

[31] Andrew Kerr, Haicheng Wu, Manish Gupta, Dustyn Blasig, Pradeep Ramini, Duane Merrill, Aniket Shivam, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Matt Nicely. 2022. *CUTLASS*. https://github.com/NVIDIA/cutlass

[32] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *ISCA*. https://doi.org/10.1109/ISCA45697.2020.00047

[33] Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, and Kevin Hsieh. 2017. Toward Standardized Near-Data Processing with Unrestricted Data Placement for GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '17)*. Association for Computing Machinery, New York, NY, USA, Article 24, 12 pages. https://doi.org/10.1145/3126908.3126965

[34] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Comput. Archit. Lett.* (2016). https://doi.org/10.1109/LCA.2015.2414456

[35] Andrew Lavin and Scott Gray. 2015. Fast Algorithms for Convolutional Neural Networks. https://doi.org/10.48550/ARXIV.1509.09308

[36] Sunjung Lee, Jaewan Choi, Wonkyung Jung, Byeongho Kim, Jaehyun Park, Hweesoo Kim, and Jung Ho Ahn. 2022. MVP: An Efficient CNN Accelerator with Matrix, Vector, and Processing-Near-Memory Units. *ACM Trans. Des. Autom. Electron. Syst.* 27, 5, Article 42 (jun 2022), 25 pages. https://doi.org/10.1145/3497745

[37] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee,

Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Jinhyun Kim, O Seongil, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim. 2021. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology : Industrial Product. In *ISCA*. https://doi.org/10.1109/ISCA52012.2021.00013

[38] Seongju Lee, Kyuyoung Kim, Sanghoon Oh, Joonhong Park, Gimoon Hong, Dongyoon Ka, Kyudong Hwang, Jeongje Park, Kyeongpil Kang, Jungyeon Kim, Junyeol Jeon, Nahsung Kim, Yongkee Kwon, Kornijcuk Vladimir, Woojae Shin, Jongsoon Won, Minkyu Lee, Hyunha Joo, Haerang Choi, Jaewook Lee, Donguc Ko, Younggun Jun, Keewon Cho, Ilwoong Kim, Choungki Song, Chunseok Jeong, Daehan Kwon, Jieun Jang, Il Park, Junhyun Chun, and Joohwan Cho. 2022. A 1ynm 1.25V 8Gb, 16Gb/s/pin GDDR6-based Accelerator-in-Memory supporting 1TFLOPS MAC Operation and Various Activation Functions for Deep-Learning Applications. In *Proc. IEEE Int. Solid-State Circuits Conf.*

[39] Zhe Ma, Francky Catthoor, and Johan Vounckx. 2005. Hierarchical Task Scheduler for Interleaving Subtasks on Heterogeneous Multiprocessor Platforms. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference* (Shanghai, China) *(ASP-DAC '05)*. Association for Computing Machinery, New York, NY, USA, 952–955. https://doi.org/10.1145/1120725.1120765

[40] Sébastien Marcel and Yann Rodriguez. 2010. Torchvision the Machine-Vision Package of Torch. In *Proceedings of the 18th ACM International Conference on Multimedia* (Firenze, Italy) *(MM '10)*. Association for Computing Machinery, New York, NY, USA, 1485–1488. https://doi.org/10.1145/1873951.1874254

[41] Narasinga Rao Miniskar, Frank Liu, Aaron R. Young, Dwaipayan Chakraborty, and Jeffrey S. Vetter. 2021. A Hierarchical Task Scheduler for Heterogeneous Computing. In *High Performance Computing: 36th International Conference, ISC High Performance 2021, Virtual Event, June 24 – July 2, 2021, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 57–76. https://doi.org/10.1007/978-3-030-78713-4_4

[42] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C.-Y. Cher, C. H. A. Costa, J. Doi, C. Evangelinos, B. M. Fleischer, T. W. Fox, D. S. Gallo, L. Grinberg, J. A. Gunnels, A. C. Jacob, P. Jacob, H. M. Jacobson, T. Karkhanis, C. Kim, J. H. Moreno, J. K. O'Brien, M. Ohmacht, Y. Park, D. A. Prener, B. S. Rosenburg, K. D. Ryu, O. Sallenave, M. J. Serrano, P. D. M. Siegl, K. Sugavanam, and Z. Sura. 2015. Active Memory Cube: A processing-in-memory architecture for exascale systems. *IBM Journal of Research and Development* 59, 2/3 (2015), 17:1–17:14. https://doi.org/10.1147/JRD.2015.2409732

[43] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C.-Y. Cher, C. H. A. Costa, J. Doi, C. Evangelinos, B. M. Fleischer, T. W. Fox, D. S. Gallo, L. Grinberg, J. A. Gunnels, A. C. Jacob, P. Jacob, H. M. Jacobson, T. Karkhanis, C. Kim, J. H. Moreno, J. K. O'Brien, M. Ohmacht, Y. Park, D. A. Prener, B. S. Rosenburg, K. D. Ryu, O. Sallenave, M. J. Serrano, P. D. M. Siegl, K. Sugavanam, and Z. Sura. 2015. Active Memory Cube: A processing-in-memory architecture for exascale systems. *IBM Journal of Research and Development* 59, 2/3 (2015), 17:1–17:14. https://doi.org/10.1147/JRD.2015.2409732

[44] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *CoRR* (2019). arXiv:1906.00091 http://arxiv.org/abs/1906.00091

[45] Ataberk Olgun, Juan Gómez Luna, Konstantinos Kanellopoulos, Behzad Salami, Hasan Hassan, Oğuz Ergin, and Onur Mutlu. 2021. PiDRAM: A Holistic End-to-end FPGA-based Framework for Processing-in-DRAM. arXiv:2111.00082 [cs.AR]

[46] Michael Orr and Oliver Sinnen. 2021. Optimal task scheduling for partially heterogeneous systems. *Parallel Comput.* 107 (2021), 102815. https://doi.org/10.1016/j.parco.2021.102815

[47] Alexandros Papakonstantinou, Yun Liang, John A. Stratton, Karthik Gururaj, Deming Chen, Wen-Mei W. Hwu, and Jason Cong. 2011. Multilevel Granularity Parallelism Synthesis on FPGAs. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. 178–185. https://doi.org/10.1109/FCCM.2011.29

[48] Ashutosh Pattnaik, Xulong Tang, Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, and Chita R. Das. 2016. Scheduling techniques for GPU architectures with processing-in-memory capabilities. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 31–44. https://doi.org/10.1145/2967938.2967940

[49] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 779–788.

[50] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*. Springer, 234–241.

[51] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *CVPR*. https://doi.org/10.1109/CVPR.2018.00474

[52] Hyunsung Shin, Dongyoung Kim, Eunhyeok Park, Sungho Park, Yongsik Park, and Sungjoo Yoo. 2018. McDRAM: Low Latency and Energy-Efficient Matrix Computations in DRAM. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.* (2018). https://doi.org/10.1109/TCAD.2018.2857044

[53] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. https://doi.org/10.48550/ARXIV.1409.1556

[54] Kyle Spafford, Jeremy Meredith, and Jeffrey Vetter. 2010. Maestro: Data Orchestration and Tuning for OpenCL Devices. In *Euro-Par 2010 - Parallel Processing*, Pasqua D'Ambra, Mario Guarracino, and Domenico Talia (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 275–286.

[55] Jaspar Subhlok and Gary Vondran. 1996. Optimal Latency-Throughput Tradeoffs for Data Parallel Pipelines. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures* (Padua, Italy) *(SPAA '96)*. Association for Computing Machinery, New York, NY, USA, 62–71. https://doi.org/10.1145/237502.237508

[56] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. 2019. MnasNet: Platform-Aware Neural Architecture Search for Mobile. In *CVPR*. https://doi.org/10.1109/CVPR.2019.00293

[57] Mingxing Tan and Quoc V. Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. *CoRR* (2019). arXiv:1905.11946 http://arxiv.org/abs/1905.11946

[58] Yizhou Wei, Minxuan Zhou, Sihang Liu, Korakit Seemakhupt, Tajana Rosing, and Samira Khan. 2022. PIMProf: An Automated Program Profiler for Processing-in-Memory Offloading Decisions. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 855–860. https://doi.org/10.23919/DATE54114.2022.9774560

[59] Yudong Wu, Mingyao Shen, Yi-Hui Chen, and Yuanyuan Zhou. 2020. Tuning applications for efficient GPU offloading to in-memory processing. In *Proceedings of the 34th ACM International Conference on Supercomputing*. 1–12.

[60] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. 2021. Spacea: Sparse matrix vector multiplication on processing-in-memory accelerator. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 570–583.

[61] Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, and Yibo Zhu. 2022. Bolt: Bridging the Gap between Auto-tuners and Hardware-native Performance. In *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu (Eds.), Vol. 4. 204–216. https://proceedings.mlsys.org/paper/2022/file/38b3eff8baf56627478ec76a704e9b52-Paper.pdf

[62] Liang Yan, Mingzhe Zhang, Rujia Wang, Xiaoming Chen, Xingqi Zou, Xiaoyang Lu, Yinhe Han, and Xian-He Sun. 2021. CoPIM: A Concurrency-aware PIM Workload Offloading Architecture for Graph Applications. In *2021 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. 1–6. https://doi.org/10.1109/ISLPED52811.2021.9502483

[63] Xia Zhao, Magnus Jahre, and Lieven Eeckhout. 2020. Selective Replication in Memory-Side GPU Caches. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 967–980. https://doi.org/10.1109/MICRO50266.2020.00082