Newton: A DRAM-maker's Accelerator-in-Memory (AiM) Architecture for Machine Learning

Mingxuan He
Electrical and Computer Engineering
Purdue University
West Lafayette, IN, U.S.A.
he238@purdue.edu

Choungki Song
DRAM Design
SK Hynix
Icheon, South Korea
choungki.song@sk.com

Ilkon Kim
DRAM Design
SK Hynix
Icheon, South Korea
ilkon.kim@sk.com

Chunseok Jeong
DRAM Design
SK Hynix
Icheon, South Korea
chunseok.jeong@sk.com

Seho Kim
DRAM Design
SK Hynix
Icheon, South Korea
seho5.kim@sk.com

Il Park
DRAM Design
SK Hynix
Icheon, South Korea
il.park@sk.com

Mithuna Thottethodi
Electrical and Computer Engineering
Purdue University
West Lafayette, IN, U.S.A.
mithuna@purdue.edu

T. N. Vijaykumar
Electrical and Computer Engineering
Purdue University
West Lafayette, IN, U.S.A.
vijay@ecn.purdue.edu

***Abstract*—Advances in machine learning (ML) have ignited hardware innovations for efficient execution of the ML models many of which are memory-bound (e.g., long short-term memories, multi-level perceptrons, and recurrent neural networks). Specifically, inference using these ML models with small batches, as would be the case at the Cloud edge, has little reuse of the large filters and is deeply memory-bound. Simultaneously, processing-in or -near memory (PIM or PNM) is promising unprecedented high-bandwidth connection between compute and memory. Fortunately, the memory-bound ML models are a good fit for PIM. We focus on digital PIM which provides higher bandwidth than PNM and does not incur the reliability issues of analog PIM. Previous PIM and PNM approaches advocate full processor cores which do not conform to PIM's severe area and power constraints. We describe Newton, a major DRAM maker's upcoming accelerator-in-memory (AiM) product for machine learning, which makes the following contributions: (1) To satisfy PIM's area constraints, Newton (a) places a minimal compute of only multiply-accumulate units and buffers in the DRAM which avoids the full-core area and power overheads of previous work and thus makes PIM feasible for the first time, and (b) employs a DRAM-like interface for the host to issue commands to the PIM compute. The PIM compute is rate-matched to the internal DRAM bandwidth and employs a non-intuitive, global input vector buffer shared by the entire channel to capture input reuse while amortizing buffer area cost. To the host, Newton's interface is indistinguishable from regular DRAM without any offloading overheads and PIM/non-PIM mode switching, and with the same deterministic latencies even for floating-point commands. (2) To prevent the PIM-host interface from becoming a bottleneck, we include three optimizations: commands which gang multiple compute operations both within a bank and across banks; complex, multi-step compute commands – both of which save critical command bandwidth; and targeted reduction of t_{FAW} overhead. (3) To capture output vector reuse with reasonable buffering, Newton employs an unusually-wide interleaved layout for the matrix. Our simulations running state-of-the-art neural networks show that building on a realistic HBM2E-like DRAM, Newton achieves 10x and 54x average speedup over a non-PIM system with infinite compute that perfectly uses the external DRAM bandwidth and a realistic GPU, respectively.**

***Index Terms*—processing-in-memory, fully-connected neural networks, DRAM**

I. Introduction

Machine learning (ML) is emerging as an important domain for processing vast amounts of data. While some ML inference workloads (e.g., visual processing) are compute-intensive (e.g., convolutional neural networks (CNNs)), many others (e.g., language and speech processing) are memory-bound (e.g., long sort-term memories (LSTMs), recurrent neural networks (RNNs), multi-layer perceptrons (MLPs), and the fully-connected classification layers in convolutional neural networks (CNNs)). Simultaneously, processing near or in memory (PNM or PIM) promises to provide unprecedented high-bandwidth and low-energy connection between compute and memory (PNM examples include 2.5-D integration via interposers employed by Hybrid Memory Cube [34] and High Bandwidth Memory [25]). As such, this application pull and technology push match perfectly.

While PIM is an old idea [7], [19], [22], [30], [33], [44], PNM is a more recent variant [3], [13], [15], [16], [24], [35]. Broadly, there are three variants depending upon where the compute is placed relative to the memory: the compute is (1) within each DRAM bank before the sense amplifiers (*analog-PIM*) [8], [29], [38], [39], [43]; (2) outside each DRAM bank just after the sense amplifiers (*digital-PIM*); and (3) after the global bus (either *PNM* or a host such as a CPU, GPU, or accelerator), as shown in Figure 1. While conventional DRAM allows its banks to be accessed in parallel, data can be retrieved from only one bank at a time due to the narrow, off-chip connection between DRAM and compute (e.g., narrow channel width). Analog-PIM and digital-PIM provide higher bandwidth than conventional DRAM by enabling compute to retrieve data from all the banks in parallel via wide, on-die connections. However, analog-PIM incurs the well-known issues of noise, scalability, and process variation which affects speed in digital logic but value accuracy in analog logic where values change with transistor parameters. To account

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00040

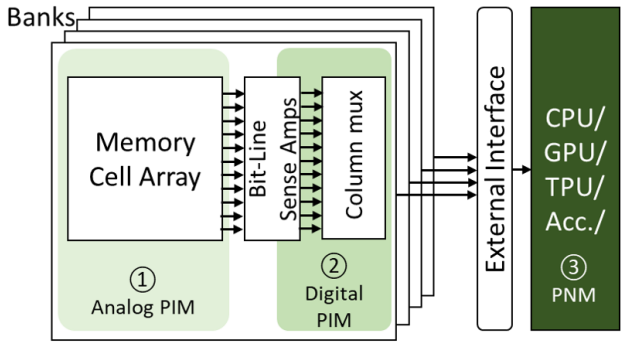

Fig. 1. Processing in/near Memory variants

for process variation, each chip may have to be trained separately to maintain accuracy. PNM has wider, in-package DRAM-compute connection than conventional DRAM but lower bandwidth than the PIM variants due to the lack of fully-parallel data retrieval from all the banks. Therefore, we focus on digital-PIM.

The bandwidth advantage of digital-PIM comes with significant constraints, involving hardware and applications. On the hardware front, compute area and power are constrained to avoid excessive DRAM density loss and thermal challenges (e.g., no more than 25% area overhead). Thus, buffering, wiring, and arithmetic logic for compute are limited. Though PIM's data bandwidth is high, the compute-memory command bandwidth remains constrained. Further, compute logic implemented using a DRAM process is slow though memory-bound applications would not be affected by the slow compute. PNM avoids these constraints by placing the compute logic in a separate die but provides lower bandwidth as mentioned above. Avoiding slow DRAM process via SRAM-based PIM [17] has worse capacity and energy.

On the application front, PIM can improve only regular, memory-bound and not compute-bound applications (i.e., applications with only a few operations per byte accessed of some large data). Further, a fundamental problem is that two-input-one-output operations with more than one large, low-temporal-reuse operand pose the difficulty that PIM's compute can be near only one of the operands, requiring massive data movement for the other operand(s) like non-PIM architectures and thereby losing PIM's bandwidth advantage. Thus, even among memory-bound applications, only those that have only one large, low-temporal-reuse operand can benefit from PIM (i.e., the other two are small with high temporal reuse and can be held in small buffers). Matrix-vector computation at the heart of inference with ML applications, such as LSTMs, RNNs, MLPs, and fully-connected classification layers in CNNs, have (1) one large input (filter matrix of size 10-100s MB), one small input (vector) and one small output (vector), and (2) only one multiply-accumulate (MAC) operation per matrix byte accessed. Thus, these regular and deeply memory-bound applications are a good fit for PIM. A key point is that any filter reuse means caching would reduce memory bandwidth demand obviating PIM or PNM solutions (i.e., any reuse means many operations per byte accessed such as the matrix-matrix multiplications in CNNs). Specifically, batching of several inputs in LSTMs, RNNs, or MLPs induces filter reuse which can be exploited by caching without PIM or PNM (i.e., matrix-vector multiplication becomes matrix-matrix multiplication with considerable matrix reuse). While ML inference in the cloud may enjoy sizable batches, inference at the edge (e.g., smartphones, hand-held devices, or even edge servers) typically would have small or even a size-of-one batches.

Previous PIM work does not identify this key application constraint. The lack of appropriate applications has been a problem for PIM until now. Further, previous PIM proposals employ full superscalar, vector/SIMD core(s) [7], [19], [22], [30], [33], [44]. However, PIM (analog or digital) is subject to severe area and power constraints due to which only the bare minimum compute hardware, such as some multiply-accumulate units (MACs) and buffers, would be acceptable (e.g., even such minimal hardware incurs around 20% area penalty).

To address the above issues, we propose *Newton*, an accelerator-in-memory (AiM) architecture for machine learning. Newton is the architecture of a major DRAM maker's upcoming AiM product which is a culmination of the decades of work on PIM. We make the following contributions:

- Newton (a) places a minimal compute of *only* multiply-accumulate units and buffers which, unlike the previous PIM and PNM proposals, does not incur the *full-core* area and power overheads and thus makes PIM feasible for the first time, and (b) employs a DRAM command-like interface for the host to issue commands to the PIM compute. The PIM compute is rate-matched to the DRAM's internal bandwidth, and employs a non-intuitive feature: a global input vector buffer shared by the entire channel to capture full input reuse while amortizing the DRAM row-wide input buffer area cost over the channel. Newton's interface avoids the overhead and granularity issues of offloading-based accelerators, so that Newton is indistinguishable from regular DRAM to the host – no kernel launch delay (key for small batches), the same deterministic latencies as regular DRAM commands even for floating-point compute commands, and no PIM/non-PIM mode switching for Newton or the host. However, even with its significant bandwidth advantage, naive AiM without the optimizations listed below performs only 48% faster than a GPU.
- To prevent the PIM-host interface from becoming a bottleneck, we include three optimizations: (1) a single compute command to *gang* multiple compute operations, both within a DRAM bank and across *all* banks, (2) *complex* compute commands which is a departure from simple DRAM read/write commands (e.g., a single compute command triggers (a) a broadcast of the input vector from the global buffer, (b) column-read of the filter matrix, and (c) multiply-add) – both of which save critical command

bandwidth, and (3) targeted reduction of timing overhead that reduces command bandwidth (e.g., t_{FAW}).

- The input vector is multiplied by every matrix row to produce the output vector (i.e., the input and output vectors have high reuse but the matrix has no reuse). To capture this reuse fully with only limited buffering, Newton employs an unusually-wide interleaved layout for the filter matrix (DRAM row-wide). This interleaving reduces the output vector write traffic with minimal output buffering.

Our simulations running state-of-the-art neural networks show that building on a realistic HBM2E-like DRAM Newton achieves 10x and 54x average speedup over a non-PIM system with infinite compute that perfectly uses the external DRAM bandwidth and a realistic GPU, respectively. While our evaluation is based on HBM2E-like DRAM, Newton is applicable to other DRAMs, including DDR, LPDDR, and GDDR families.

II. Background

A. Conventional DRAM architecture

We provide a brief overview of a typical DRAM architecture. We describe only the high-level details needed to understand our AiM architecture (i.e., this description is not intended to cover all the details of the DRAM architecture). We assume one channel and one rank for a modern DRAM architecture which employs many internal DRAM banks (e.g., 16). A DRAM row-miss read access starts by activating the row in a specific bank and latching the data in the bit-line sense-amplifiers (BLSAs) of the bank. The activation is followed by a DRAM column access of n bits from the row buffer (e..g, n = 256 bits or 32 bytes). These bits are routed from the bank to the global bus and then onto the off-chip memory bus. while the banks operate in parallel in conventional DRAM, the data retrieval from different banks are serialized through the global bus (only one set of n bits). While HBM makes wider column accesses than conventional DRAM (e.g., 256 bits versus 64 bits), the data retrieval from the banks is still serial like conventional DRAM. In contrast, PIM places the compute directly next to each bank which exposes the full internal bandwidth across the banks. To avoid any confusion between DRAM rows (columns) and matrix rows (columns), we will qualify the terms rows and columns with DRAM or matrix as appropriate.

Conventional systems making cache block accesses to memory employ cache block-interleaving so that consecutive cache blocks are mapped to adjacent channels to enable parallel accesses at cache block granularity. Channel parallelism simply results in the bandwidth being multiplied, in any of conventional, HBM, or PIM.

B. Workload

Modern LSTMs, RNNs, MLPs, and fully-connected classification layers in DNNs can be viewed as a matrix-vector product as shown in Figure 2. Figure 2 illustrates the key computation for one output element (highlighted in red). Because

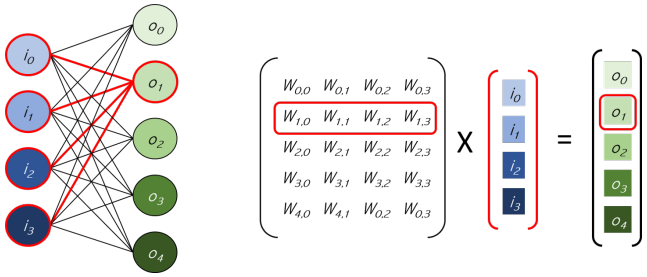

Fig. 2. A Fully-connected Neural Network Layer

each output computation requires the pairwise multiplication of the model weights and the inputs (and the summation of such products), the model weights can be viewed as a matrix where each matrix row contributes to the corresponding output cell. While, Figure 2 shows a small example, at typical sizes, one layer in a model may operate on an input of 4K elements (a 4Kx1 column vector) to produce an output column vector of 1K elements for which the matrix size is effectively 1K (rows) x 4K (columns). A neural network activation function (e.g., ReLU, sigmoid, and tanh), distinct from DRAM row activation, is applied to the output vector elements. A full model, where each layer's output is the next layer's input, may employ many layers so that the total model size may be large (e.g., 340M elements in Google's BERT [10]). Because matrix-vector multiplication has no data reuse of the matrix elements (only one operation per element) and because the large matrices cannot fit on-chip, these workloads are deeply memory-bound. Fortunately, these regular workloads satisfy the constraint of only one operand being large making them a good fit for PIM. To avoid terminology that may fit some of these ML models but not others, we will use the general term matrix-vector multiplication as the main computation instead of specific terms like filter-input multiplication.

III. Newton

Recall from Section I that, first, to keep area and power overheads acceptable Newton employs (a) a minimal compute of *only* multiply-accumulate units and buffers which avoids *full-core* area and power overheads of previous work and thus makes PIM feasible for the first time, and (b) a DRAM-like command interface for the host CPU to issue commands to the PIM compute. The PIM compute is rate-matched to the internal DRAM bandwidth, and employs a global input vector buffer shared by the entire channel to capture input reuse while amortizing the buffer area over the channel. Newton's interface makes it indistinguishable from regular DRAM to the host, avoiding offloading overheads and PIM/non-PIM mode switching, and ensuring the same deterministic latencies as regular DRAM commands even for floating-point compute commands. Second, to prevent the PIM-host interface from becoming a bottleneck, three key optimizations of the interface are: (a) commands which gang multiple compute operations both within a bank and across all banks, and (b) complex, multi-step compute commands to save critical command

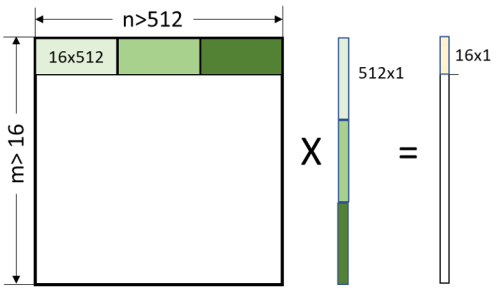

Fig. 3. AiM Tiling for DRAM with 16 banks and 1 KB DRAM row size

bandwidth; (c) deterministic operation even for floating–point compute commands to avoid handshake overhead; and (d) targeted reduction of command overhead. Finally, to exploit output vector reuse with minimal output buffering, Newton employs a DRAM row-wide interleaved layout of the filter matrix, We start with a single channel for simplicity and extend to multiple channels later.

A. Interleaved layout for the filter matrix

While the matrix is resident in the DRAM, the input vector is broadcast to the banks. Each element of the output vector is computed by multiplying a matrix row and the input vector. The input vector is multiplied by every matrix row (i.e., the input vector has high reuse but the matrix has no reuse). To capture this reuse in full, Newton employs a simple interleaved layout for the matrix. Because each matrix row is accessed one DRAM column access at a time, the idea is to hold a *chunk* of the input vector elements and completely reuse the input elements with all the matrix rows before moving on to the next set of input elements so that the same input elements are never re-fetched. Accordingly, the matrix rows are laid out in a chunk-interleaved manner, where the first matrix row's first chunk is followed by the second matrix row's first chunk, and so on. Upon filling the DRAM row of one bank, this interleaving continues to the next bank for maximizing reuse (e.g., each 1-KB DRAM row has a chunk of one of the matrix rows). Figure 3 assumes 16 banks and DRAM rows with 16x512 bits = 8 Kb = 1 KB so that 1 KB each of the first 16 matrix rows are mapped to the 16 banks. We discuss the schedule of the computation later in Section III-C. If there are more matrix rows than the banks then the interleaving continues in later DRAM rows in the banks. The first chunk of all the matrix rows is followed by the second chunk of all the matrix rows, and so on. One would expect the chunk width to equal the column-access width to capture column-access parallelism while keeping the *input* buffering as small as possible (e.g., 16 16-bit elements, which is 256 bits). However, each chunk is actually as wide as a DRAM row to reduce the *output* buffering as explained in Section III-C; input buffering reduction is also explained in Section III-C. We assume 16-bit floating-point data because our customers and partners stipulate that recommendation systems, unlike CNNs,

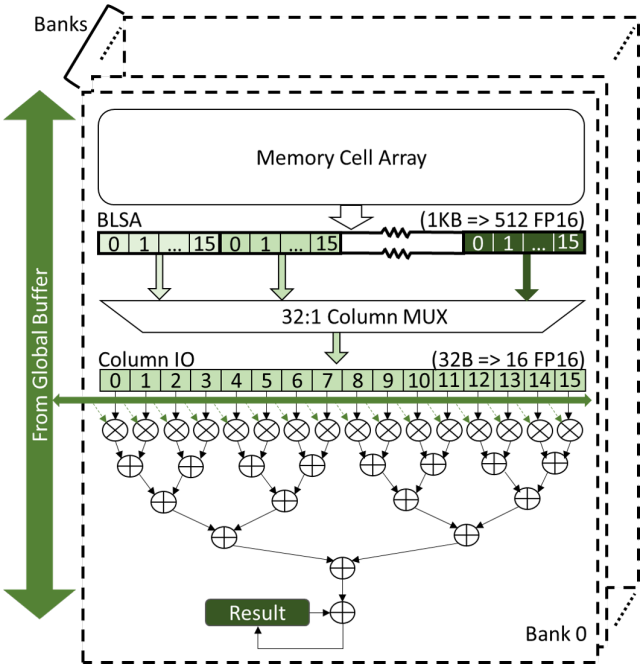

Fig. 4. Newton Datapath

need high accuracy (e.g., 0.1% accuracy difference matters for revenue).

Finally, AiM memory can be used as normal memory and can hold non-AiM data. However, for ease of timing explained later in Section III-C, AiM and non-AiM data can be in the same bank but not in the same DRAM row.

B. Newton organization

Newton employs k multipliers per bank to match each bank's column-access bandwidth (e.g., k = 16), as shown in Figure 4. The multipliers are implemented using DRAM-technology transistors. Because of area constraints, Newton does not employs as many multipliers as needed to cover an entire DRAM row whose data is accessed anyway only a DRAM column access at a time as a trade-off in DRAM design between access bandwidth and area (not specific to AiM). Newton's strategy captures all the internal DRAM bandwidth as the multipliers are rate-matched to the column-access bandwidth (i.e., all the column access data multiplications are complete under the column access latency).

The matrix row and the input vector chunks are multiplied (e.g., 16 elements at a time in any given bank) and reduced through a pipelined adder tree (Figure 4). The host retrieves the latched result of the tree. It may seem that this reduction tree may incur extra area compared to an approach where the matrix is laid out in column-major format so that the different matrix rows in each column access would be multiplied by the input vector (e.g., each DRAM row would hold matrix rows in an element-interleaved manner where the first element of the 16 matrix rows would be followed by the matrix rows' second element, and so on). Then, there would simply be 16

multiply-accumulates (requiring 16 multipliers and 16 adders) to consume each DRAM column access data However, both the tree and column-format approaches require 16 multipliers and 16 adders (a 16-to-1 adder tree needs 15 adders plus one adder for the accumulation as shown in Figure 4). In fact, the column major approach requires 16 accumulator latches (as opposed to only one accumulator latch with the adder tree approach).

In addition to the modest area advantage due to fewer latches, the adder-tree approach has one additional advantage. If the matrix rows are fewer than the multipliers, which in the future may be in the thousands, then the column-format approach may result in idle multipliers and loss of throughput.

Avoiding this idling requires the matrix rows to be partitioned into sub-vectors and distributed among multiple banks. This partitioning needs a reduction across the banks for the final result. In the former approach, underutilization occurs only in the case that the number of matrix rows is smaller than the total number of banks (across all memory channels). As such, because the number of matrix rows (512+) are typically more than the total number of banks (256-384, in aggressive 16-24 channel memory systems), the latter approach's unfavorable case is more likely than the former's. Hence, the former approach (the adder tree) is better.

At a high level, a typical matrix-vector multiplication involves (1) parallel (but staggered) DRAM row activations in all the banks (for maximum parallelism), (2) broadcast of the input vector chunk, (3) access of the matrix row chunk, (4) multiply the chunks and reduce. Recall from Section III-A that the layout enables one input vector chunk to be reused fully before moving to the next chunk. Like the input vector, the output vector also has reuse: the element-by-element product of a matrix row and input vector updates the same output vector element. However, capturing the output reuse requires accessing all the chunks in the input vector to multiply with the full matrix row. In such a scenario, additionally achieving full input reuse would require buffering the entire input vector which may not be feasible (the full input vector may be longer than a DRAM row). As such, limited buffering implies fully capturing either full input or full output reuse but not both (largely symmetric), or some partial input and partial output reuse.

Newton employs DRAM-row wide chunks, as mentioned in Section III-A. To capture full input reuse, a DRAM-row wide buffer holds the input vector chunk. To reduce the area, the buffer is global and shared by all the banks in a channel. The 16 parallel multiply operations per bank require the corresponding input vector elements, called *sub-chunks*, to be read from the global buffer and broadcast to all the banks directly into the multiplier inputs without any further per-bank latching to save area. The relatively-slow DRAM-process transistors accommodate these global buffer reads and broadcasts. The adder tree result per bank, a single output vector element, held in the result latch accumulates the entire input vector chunk's result over several rounds of multiplications and additions. Thus, only a single-element latch is required for an entire

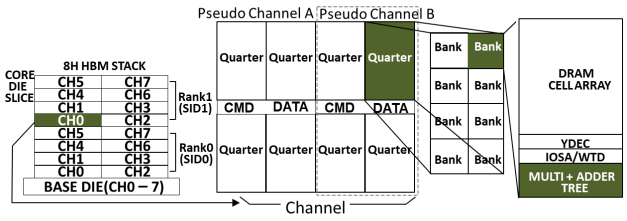

Fig. 5. Abstract Floor Plan

input chunk's result. This small output buffering is the reason for the DRAM row-wide interleaving. At the same time, by globally sharing the input vector buffer across all the banks in a channel, the input buffer cost is reduced.

Figure 5 illustrates an abstract floorplan of Newton implemented on HBM2E. Within HBM2E's channel and bank organization, the per-bank compute logic (highlighted in dark-green) is shown for one bank. The per-channel global buffer is placed in the peripheral area.

C. Newton operation

Newton's operation may be viewed as imposing a tiling on the iteration space of the matrix-vector product computation as shown in the example in Figure 3. The example assumes a 16-bank DRAM with a 1-KB row (i.e., 1 KB = 8 Kb = 512 x 16 bits = 512 `bfloat16` elements per DRAM row). Each 16×512 tile represents the computation granularity that corresponds to one row of 512 elements for each of the 16 banks. The 512 elements in one chunk are consumed one sub-chunk (16 `bfloat16` words) at a time in the innermost loop to produce and write out one partial output vector element (see Algorithm 1). In the next outer loop, the computation proceeds with the tiles moving down the rows in column-major fashion (while holding the input vector chunk for full input reuse) which produces the partial output vector elements read out at the end of each DRAM-row. Finally the outermost loop moves across the matrix columns, outputting more partial output vector elements each of which is reduced by the host to the appropriate output vector element.

The host reads the result latches from all the banks in parallel and concatenated together into a sub-chunk (e.g., 16-bit results from 16 banks concatenated to form 256 bits in the output vector). The DRAM-row wide chunks (Section III-A) lower the output write traffic with minimal output buffering by ensuring that the host reads results only once per full DRAM row. The wider the chunk the lower the traffic but the more the input buffering whose cost is amortized over the entire channel by employing a global buffer. If the matrix row is wider than the chunk, then the host reduces multiple chunks' partial results all of which contribute to the same output vector element. Finally, the host processor applies the neural network

Algorithm 1 Newton's Tiled MV computation

```
1: function MVPRODUCT(InputVectorV, MatrixM, m, n)
2:     numChunks = n/512                          ▷ Number of chunks
3:     C[1..numChunks] ← split(V)                 ▷ Split vector to chunks
4:     for i ∈ 1..numChunks do                    ▷ Outermost loop.
5:         GlobalBuffer ← C[i]
6:         r = m/16                               ▷ Number of vertical tile positions
7:         for j ∈ 1..r do
           ▷ Compute 1 DRAM row x Global Buffer per bank
8:             for all b ∈ 1..numBanks do
9:                 Results[b] ← ComputeTile(Tile j, Row b)
10:            end for
11:            TileResult ← ReadResultsFromAllBanks()
12:                ▷ Tile result sent for accumulation at host
13:        end for
14:    end for
15: end function
```

activation functions (distinct from DRAM row activation) to the final reduced outputs (Section II-B).

Unlike neural network activation function which can be applied as and when elements of the vector are computed, the scaling factor for batch normalization depends on the range of values in the full vector (i.e., maximum, minimum). Even so, most of the batch-normalization latency is hidden under Newton computation by (1) determining the range of values of the vector on the host as they are produced, and (2) exposing the normalization latency of only the first tile after the vector is produced before launching the next layer's MV computation on Newton with that normalized tile while other tiles are normalized under Newton's compute latency.

An alternative schedule to decrease the output traffic is to discard the interleaving and instead lay out a full matrix row in one DRAM bank occupying contiguous DRAM rows if necessary. The next matrix row goes to the next bank and so on, wrapping around to the first bank. With this layout, each input vector chunk is broadcast to the banks as before with two differences: (1) On the positive side, the result is accumulated for the entire matrix row, not just one chunk (DRAM row) as in the interleaving scheme, reducing output traffic. (2) On the negative side, each input vector chunk is used by all the banks but has to be re-fetched for the next set of matrix rows across all the banks unlike the interleaving scheme where each chunk is reused fully for all the matrix rows. This corresponds to a row-major traversal of the tiled computation (instead of the column-major traversal that captures reuse). We evaluate this alternative, called Newton-no-reuse. The input traffic rise in Newton-no-reuse far exceeds the output traffic fall – the entire input vector chunk (an entire DRAM row) has to be refetched per matrix DRAM row versus one sub-chunk of output read out per matrix DRAM row, causing significant performance drop as we show in Section V-B. Finally, in this variant, the neural network activation functions are implemented as look-up tables. Newton employs a single look up table per channel. To apply the neural activation to the results in different banks, the table is conceptually multi-ported. The neural activation is applied to the final result before being read by the host.

TABLE I
Additional commands to support Newton

Command	Operation
COMP#	Ganged multiply of sub-chunk# in all banks
READRES	Read the Result latches of all banks
GWRITE#	WRITE sub-chunk# to the Global Buffer
G_ACT#	Ganged activation of 4-bank cluster#

We also explored an option in between the extremes of full input reuse and no input reuse. The new option reuses the input vector among four matrix rows in each of all the banks by providing four result latches per bank. Compared to the full-reuse variant, this option avoids the per-DRAM row output traffic while incurring input fetch once every four matrix rows. However, the output traffic is so low that even in the benchmarks with fewer than four matrix rows per bank (i.e., small matrices) where the full-reuse variant has no advantage over this option, the former performs virtually similarly to the latter while avoiding the latter's extra result latches. Therefore, we do not pursue this option further.

D. Newton's commands

DRAM commands must be separated by a specified delay (e.g., 4 cycles). The global buffer for the input vector chunk (1 DRAM row-wide) is loaded one column-access width at a time (`GWRITE` in Figure 7). Though the loading takes many commands, the performance impact is low due to amortization over all the matrix rows. Newton's commands are summarized in Table I. If the row activation commands were for individual banks, the commands would be staggered in time one after another. Further, the row activations would be constrained by the usual four-activation timing window (t_{FAW}) for power reasons. These constraints imply that the row activations would be stretched over a long period which is exposed at each bank. Newton addresses this problem by ganging four activations using one command within t_{FAW} constraints (G_ACT in Figure 7). Due to area constraints, DRAM rows are not double-buffered causing the last row activation latency to be exposed (the other activations are overlapped).

We also consider reducing the t_{FAW} overhead. Like most modern integrated systems, DRAMs use many internal voltages that are generated from external voltage supplies (e.g., VDD, VPPE) as shown in Figure 6. For example, the main internal voltages are VCORE (the DRAM cell core voltage) and VPP (DRAM cell wordline). Many concurrent ACT command operations cause severe internal voltage drop due to the increased current flow, requiring long delays to recover the internal voltage levels to the desired target voltage. Newton's ganging of activating four banks in a single command exacerbates the voltage drop.

Such internal voltage recovery times are closely related to several DRAM timing parameters such as t_{FAW} (our focus), t_{AA}, t_{WR}, t_{RP}, and t_{RCD}. To reduce t_{FAW}, the internal LDO (Low Dropout) regulator and the DC-DC pump driver strength must be increased to enable higher currents and faster voltage

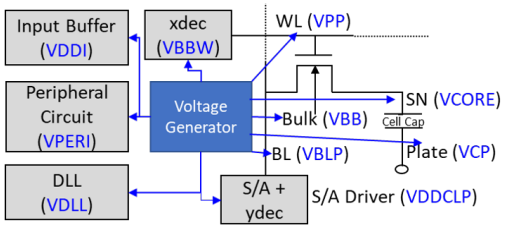

Fig. 6. DRAM internal voltages and its impact on t_{FAW}

recovery. Therefore, improving t_{FAW} comes with the cost of higher die area. Newton's t_{FAW} delays are based on the circuit-level estimates of the required internal voltage generator drive ability to achieve aggressive performance goals with acceptable area costs. These aggressive optimizations in Newton are justified in spite of their die area costs because Newton is expected to be sold at a higher price point because of the acceleration it offers. Such optimizations would not be considered in normal DRAMs.

Next, each compute command involves (i) reading a sub-chunk from the global input buffer, (ii) making a column access for the corresponding filter sub-chunk, and (iii) performing the multiplication. While typical DRAM commands are simple and involve only one operation (a read or a write), employing a simple command for each of the three steps would cause significant pressure on the command bandwidth. Instead, Newton employs a single, complex command with a parameter for the sub-chunk to capture all three steps (COMP in Table I). Nevertheless, consuming a DRAM row would require many compute commands (e.g., assuming 256-bit column access, a 1-KB DRAM row requires 32 commands). Even though the banks can operate in parallel, if each bank were to require a separate set of compute commands, the command bandwidth would be saturated. Instead, we observe that all the banks operate on the same input sub-chunk. Consequently, a single command gangs the compute operations in all the banks (e.g., COMP in Figure 7. Finally, the result from all the banks can be retrieved in a parallel manner and simply concatenated using just one command (READRES in Figure 7).

Though implemented with DRAM-process transistors (which are slower than logic-process transistors), the multiplications and each stage of the adder tree pipeline complete before the next compute command's column access to rate-match the DRAM's internal bandwidth. *Thus, in the time a conventional DRAM reads a row from one bank, AiM completes the arithmetic operations of a row in all the banks.* This timing, achieved by many circuit-level innovations, gives the same deterministic behavior and the same latencies as regular DRAM commands, which is a key advantage of Newton's interface.

Our description so far assumes one channel for simplicity. With multiple (pseudo) channels, Newton's per-channel operation and timing are simply repeated in parallel across the (pseudo) channels. There are a few remaining timing-related issues: (1) AiM and non-AiM (conventional) data cannot

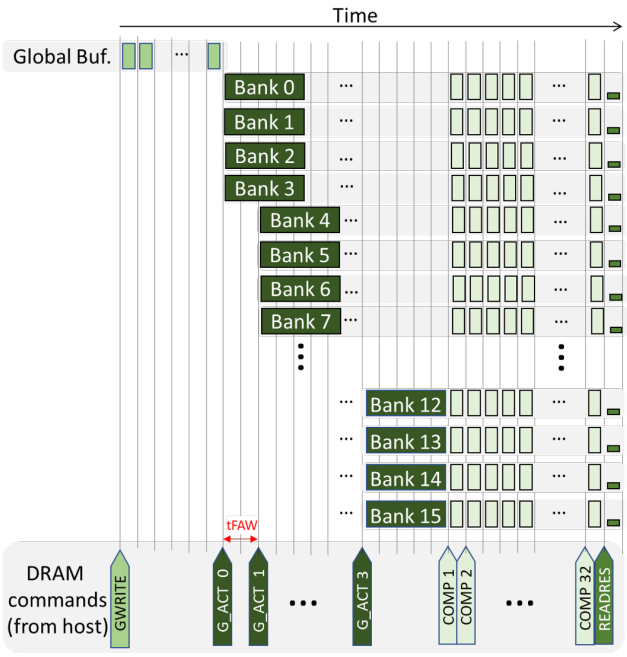

Fig. 7. Newton computation timing (One DRAM row across all banks)

reside in the same DRAM row. While non-AiM commmands can interleave with AiM commands to the same bank, the former are guaranteed to access a different row than the AiM commands. Consequently, a pre-charge is needed on the bank before the non-AiM row can be accessed, in which time the AiM operations are guaranteed to complete ensuring that non-AiM commands cannot interfere with AiM operations within the same bank.

(2) The adder tree takes more than 4 cycles to complete though there is pipelining so that another set of additions can start 4 cycles after the previous set. Due to the completion latency, the host memory controller has to insert an appropriate delay before issuing an adder-tree result read (for neural network activation or for read out to host).

(3) The number of the matrix rows may not be a multiple of the number of banks leaving some banks free when the last row in the other banks are still performing AiM operations. The free banks cannot perform non-AiM operations until all the banks are done with AiM operations.

(4) The current Newton design can process only one ML model at a time in a bank or even a channel. Different models can operate simultaneously in different channels.

E. Other concerns

Though we focus on an HBM-based implementation, Newton's key ideas are applicable to other DRAM families such as LPDDR, DDR, and GDDR, with low-level differences based on the internal bandwidth, impact on density, and implementation (e.g., number of MACs for rate matching). Further, there may be other commercial considerations that determine whether AiM is offered in these families.

Newton does not interfere with virtual memory. Address translation can proceed similarly to conventional DRAM. The Newton commands are based on physical addresses as are conventional DRAM commands. While addresses within a virtual memory page are guaranteed to be contiguous, physical pages are often not contiguous even when the virtual pages are. However, our matrix layout expects physical address contiguity. To that end, we use super pages to allocate the matrix guaranteeing physical address contiguity.

An issue with DRAM refresh is that Newton operations often take longer to complete for an entire DRAM row than conventional DRAM reads and writes. Because the result latch accumulates the result for the entire DRAM row, a refresh request in the middle of an Newton operation on a row would change the row and disrupt the operation. A simple fix is to delay the Newton operation if a refresh request would mature within the operation's latency. The memory controller simply waits for the pending refresh to mature, sends the refresh command, and then sends the Newton command. This strategy is identical to how pending refresh requests within conventional DRAM read or write latency are handled.

Finally, DRAMs rely on error correction codes (ECC) for transient errors. However, the ECC is computed and checked by the memory controller and not the DRAM itself whereas AiM computation occurs within the DRAM. Fortunately, only the matrix resides in the DRAM for long periods of time with the possibility of collecting transient errors. The input and output vectors are fetched and read out, respectively, at least once per input, which is frequent enough to lower their susceptibility. To address the matrix's susceptibility. we envision re-loading the matrix, and thereby discarding any errors, from a non-AiM copy every so often for a small bandwidth overhead (e.g., once per 1000 inputs).

F. A simple performance model

In an *ideal non-PIM* system, we start by conservatively assuming that (1) the off-die processing is limited only by the memory bandwidth and the arithmetic is hidden completely under the memory transfers; and (2) the input and output vectors are small enough to be held in the compute chip without any memory access (hence our interleaving is not relevant to the ideal non-PIM). We focus on one DRAM row (a matrix row chunk) read which requires the DRAM row to be activated. Upon activation, column accesses, separated by t_{RRD}, sequentially retrieve the row's data. Assume that the DRAM row (chunk) size is *row* and the column access width (sub-chunk) is *col_io*. The number of columns (*col*) is then given by *row*/*col_io*. In the ideal non-PIM, the long latency of retrieving the entire DRAM row *completely hides* the activation latency of a DRAM row in the next bank as well as any t_{FAW}-related delays (hence our t_{FAW} optimization is not relevant to the ideal non-PIM). Thus, ideal non-PIM's effective time for *one* DRAM row,

$$t_{ideal-non-PIM} = col * t_{CCD}$$

TABLE II
Benchmarks

Workload	Matrix	Vector
GNMT LSTMs1 [45]	4096×1024	1024×1
GNMT LSTMs2 [45]	4096×2048	2048×1
BERTs1 [10]	1024×1024	1024×1
BERTs2 [10]	1024×4096	4096×1
BERTs3 [10]	4096×1024	1024×1
AlexNetL6 [26]	21632×2048	2048×1
AlexNetL7 [26]	2048×2048	2048×1
DLRMs1 [31]	512×256	256×1

In Newton, a DRAM row in each group of four banks is activated which require separate commands separated by the inter-command delay. The t_{FAW} constraint applies across the banks in groups of four, so that activating a row in each group of four banks contributes $\max(t_{RRD}, t_{FAW})*(n/4-1)$, assuming n banks. (The last bankgroup's t_{FAW} is not relevant as no further activations are needed.) Within this constraint, the activations occur in parallel across the (groups of) banks. After the last bank group's activation with a delay of t_{ACT} is complete, all the banks make column accesses in parallel to retrieve and operate on their data. The processing is rate matched to the retrieval rate. Because the input vector chunk is fetched only once per all the matrix rows (Section III-A), we ignore input reads for model simplicity (our simulations in Section V faithfully capture the input fetch). The output vector chunk concatenated across all the banks is read out in one column access per DRAM row in all the banks. Again, we ignore the output in our model (but not in our simulations) for simplicity. Thus, Newton's time for processing one DRAM row in *all* the banks,

$$t_{Newton} = max(t_{RRD}, t_{FAW}) * (n/4-1) + t_{ACT} + col * t_{CCD}$$

Newton's speedup is $n * t_{ideal-non-PIM}/t_{Newton}$. Let

$$o = (max(t_{RRD}, t_{FAW}) * (n/4-1) + t_{ACT})/(col * t_{CCD})$$

which is the ratio of the activation overheads to the data retrieval time in Newton. Then, Newton's speedup is

$$n/(o+1)$$

We investigated overlapping some of the overhead and found that the lack of slack in the command bandwidth and the area overhead of an extra row buffer are impediments to achieving overlap.

IV. Methodology

Benchmarks: Our benchmarks include six matrix-vector (MV) product computations (which is the memory-bound computation Newton targets) chosen from popular machine learning domains such as natural language processing (GNMT, BERT), and recommendation systems (DLRM, see Table II). For each of the workloads, we identified the (possibly more than one set of) MV dimensions in the workloads (e.g., see multiple rows for BERT-large in Table II). For these workloads, the fully-connected (FC) layers account for more than 99% of the run time. For completeness, we also include

TABLE III
DRAM Configuration (HBM2E-like)

Num of Ranks	1
Num of Banks	16
Num of Rows in each bank	32768
Num of Column I/Os per row	32
Column I/O bit width	256b (16 `bfloat16`)
Num of Multipliers per bank	16
Timing Parameters (in nanoseconds)	
t_{AA} = 22–29 ns; t_{RP} = 14 ns; t_{RCD} = 14 ns; t_{RAS} = 33 ns	

the two FC layers of AlexNet, a CNN. The layers account for a smaller fraction of the inference time (e.g., 15%) but a vast majority of the model parameters, and therefore power and bandwidth. Note that CNNs' convolutional layers are compute-bound and hence unsuited for *any PIM* (not just Newton). However, if a platform running CNNs happens to have Newton, then their FC layers can run on Newton.

Newton Simulator: Our Newton simulator is based on the cycle-level DRAMsim2 [36] simulator. Though our simple performance model (Section III-F) abstracted away some details (such as loading the global buffer and reading the results), our simulator models all those details. In addition to the basic DRAM parameters (e.g., banks, row/column widths) that is similar to HBM2E [27], our modified Newton simulator includes the following key changes.

- Global buffer: The buffer that holds a row-wide chunk of the input vector and allows for direct connectivity to the multiplier inputs (with 16-way fanout for 16 banks).
- Result buffer: One scalar `bfloat16` register that holds an output per bank.
- Command Support: We add support for Newton commands as shown in Table I.

Modeling non-PIM architectures in simulation: To compare Newton against the best possible non-PIM architectures we consider an ideal non-PIM host with unlimited compute bandwidth – *Ideal Non-PIM*.

To model an upper-bound on performance of *any* non-PIM architecture including PNM proposals (e.g., [3], [13], [15], [16], [24], [35]) and traditional systems (GPU, TPU, and multicores), *Ideal Non-PIM* assumes infinite compute bandwidth and is limited only by the DRAM's external bandwidth. Thus its execution time is modeled as the time to transfer DRAM data to the host. Against this ideal system, any speedups achieved by Newton can only be higher against realistic non-PIM architectures including CPUs, GPUs, TPUs, or any imaginable custom non-PIM (PNM or traditional) accelerator. Previous PIM proposals use full cores which impose high area overheads and hence are not compared.

We use a common DRAM configuration similar to HBM-2E for both *Ideal Non-PIM* and Newton, as shown in Table III (only a subset of timing parameters and in some cases, ranges instead of actual values are shown to protect proprietary information). Our configuration uses an 8-high stack with 8 channels, 2 pseudo channels, and 16 banks per channel for a total capacity of 128 Gb. Each bank is organized as a memory cell array with 32K rows and 8K columns. Each row of 8K bits (or 1K bytes) can be accessed at a 256-bit (32-byte) column I/O granularity to which Newton's 16 multipliers per bank are rate-matched.

GPU simulation: We use GPGPUsim [6] (version 4.0), a widely-used cycle-level simulator for GPUs, to model a realistic, high-performance non-PIM host (as opposed to the unrealistic *Ideal Non-PIM* discussed above). We configure GPGPUsim as a Titan-V, a high-end model with 80 CUDA cores and 24 memory channels. We modify GPGPUsim's DRAM and its configuration to operate with the same timing parameters as Newton. On the software front, we use Cutlass-1.3 [23], a high-performance, open-source CUDA library for linear algebra.

In the simulation, we noticed that Cutlass's run time has a high constant overhead that hurts the GPU's performance for our (relatively-light) kernels. To address this problem, we ran several iterations of matrix-vector computation and isolated the incremental cost of each matrix-vector multiplication. Thus, we eliminate the constant overheads from the matrix-vector computation. (Note that this elimination is conservative in that it *minimizes* the GPU execution time; including any part of the overheads makes the GPU worse.)

Average Power Modeling: Newton's power consumption differs from that of *Ideal Non-PIM* in three key ways. First, Newton incurs compute power in the multipliers and adders that *Ideal Non-PIM* does not. Our internal models show that Newton when performing the all-bank parallel computation (i.e., when executing the COMP command) consumes about 4x as much power as *Ideal Non-PIM* when reading DRAM at peak bandwidth (e.g., consecutive column accesses of the same DRAM row). Of course, *Ideal Non-PIM* would incur compute power which we ignore for our evaluations – an advantage for *Ideal Non-PIM*. Second, *Ideal Non-PIM* incurs power to transfer the *entire* matrix over the external interface (DRAM PHY), which Newton completely avoids. Instead, Newton incur transfer power only for (1) the partial results sent to the host for final accumulation, and (2) the input vector as it is loaded in chunks to the global buffer However, the cost of Newton's transfers are dwarfed by that of *Ideal Non-PIM*'s matrix transfers. Finally, Newton incurs additional power to hold banks open for longer (to ensure that all banks are open before compute begins). In contrast, *Ideal Non-PIM* can open a page and immediately transfer the contents without waiting. We model all these components to compute the average power using DRAMSim2. We do not show power parameters which are proprietary.

V. Results

Our simulations show: (1) Newton achieves 54x speedup over a Titan V-like GPU; (2) Newton is 10x faster than any non-PIM architecture; (3) all of Newton's key optimizations contribute to its speedups; (4) Newton's speedups increase with number of banks as do the compute and internal DRAM bandwidths), although not linearly due to the Amdahl's Law effect of the activation overheads (o in Section III-F), (5) as expected, Newton's performance advantage diminishes at

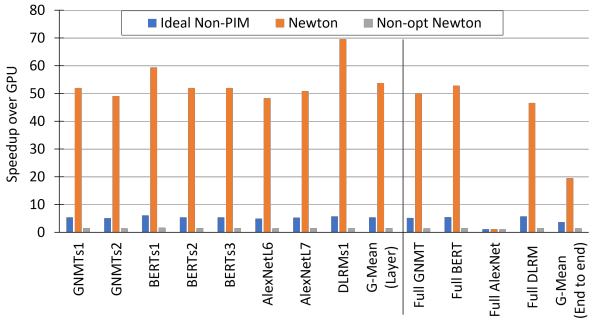

Fig. 8. Speedup

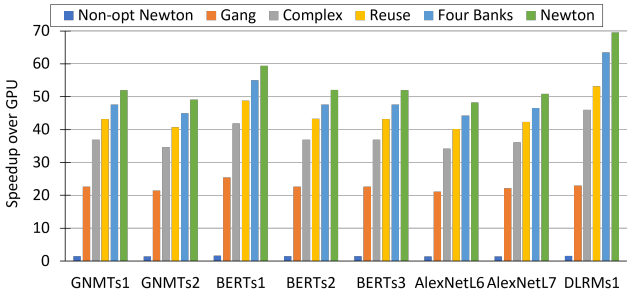

Fig. 9. Isolating Newton's optimizations

larger batch sizes as non-PNM architectures can exploit data reuse, and (6) Newton achieves higher energy efficiency than *Ideal Non-PIM*.

A. Performance

Figure 8 shows the speedup of Newton over a Titan V-like GPU with our 24-channel, (16-banks/channel) HBM-2E-like DRAM configuration for (1) individual layers of our benchmarks (X-axis left) and (2) end-to-end runs of our full benchmarks – i.e., GNMT, BERT, AlexNet, and DLRM (X-axis right). The end-to-end runs include activation functions and batch normalization as mentioned in Section III-C. For each benchmark, we also show Non-opt-Newton which does not include any of Newton's optimizations: ganged, complex commands, interleaved layout and tiling, and improved t_{FAW} (Section III-D). Figure 8 also includes *Ideal Non-PIM* as an upper bound on performance achievable by any non-PIM architecture (including PNM proposals). Newton achieves 54x speedup over the GPU when averaged across all individual layers (geometric mean on the left). In contrast, even the unrealistic *Ideal Non-PIM*'s speedup is limited to 5.4x on average, which motivates Newton. While *Ideal Non-PIM* is bound by the external bandwidth, Newton's compute is rate-matched to the internal bandwidth. In the time required for one external DRAM row transfer, Newton can consume one DRAM row in all the banks. Finally, Non-opt-Newton's modest speedup – 48% over the GPU (lower than even *Ideal Non-PIM*) – highlights the value of Newton's key optimizations (which we isolate next).

Newton achieves 10x speedup over *Ideal Non-PIM*, whereas 16 banks/channel offer a speedup opportunity of 16x. However, not all DRAM operations are possible in parallel due to staggered DRAM row activations and t_{FAW} constraints (Section III-F). Plugging in Newton's parameters into our performance model in Section III-F, the predicted speedup is 9.8x which closely matches (within 2%) the measured speedup of 10x. The prediction is lower than the measurement because the performance model ignores refresh effects, but our simulations do not. Due to its slower runs, *Ideal Non-PIM* tends to see more interruptions due to refresh than Newton, which slightly improves Newton's speedup. This effect is especially pronounced in *DLRMs1* – Facebook's Deep Learning Recommendation Model [31].

Not surprisingly, GNMT, BERT, and DLRM – the NLP and recommendation DNNs – retain high speedups in the end-to-end evaluation (in the right section of Figure 8). While Newton gets a boost in its speedup (70x) by finishing the single layers of DLRM within the refresh window, the end-to-end run of DLRM on Newton does see intervening refresh resulting in a lower (but still significant) speedup of 47x. AlexNet, a CNN and not a target for Newton due to being compute-bound, has modest end-to-end speedup of only 1.2x. Recall from Section IV that the acceleration of CNNs' FC layers is a free benefit in systems with Newton memory; and is not a key target. While the overall mean speedup including AlexNet, as shown in Figure 8, is 20x, the mean speedup for the key target applications (BERT, GNMT and DLRM) is 49x.

B. Isolating individual optimizations

Newton leverages many optimizations in hardware and software (listed below). To understand the impact of each of these optimizations in isolation, Figure 9 shows Newton's speedup (over a Titan V-like GPU) as we *progressively* add the optimizations *one at a time* leading up to the full Newton design. Starting with a non-optimized version (Non-opt-Newton), we progressively add (1) all-bank ganged compute commands (*gang*), (2) complex multi-step compute commands (*complex*), (3) reuse via tiling and interleaved layout for the filter matrix (*reuse*), (4) four-bank ganged activations (*four bank*), and (5) aggressive t_{FAW} (which results in full Newton). Without any optimizations, Non-opt-Newton performance is severely stifled despite having the same number compute and internal bandwidths as Newton; the speedup is merely 48% over the GPU. Each of Newton's optimizations significantly improves performance. The ganged computation strategy (which yields the largest improvement) reduces command bandwidth requirements by 16x which causes a significant improvement in performance. The use of complex commands offers an additional 3x reduction in command bandwidth. Each of the remaining optimizations offer further improvements leading to the full Newton design's 54x speedup on average.

C. Sensitivity to Number of Banks

Figure 10 shows the speedup of Newton over a Titan V-like GPU (Y-axis) for our benchmarks (groups of bars on

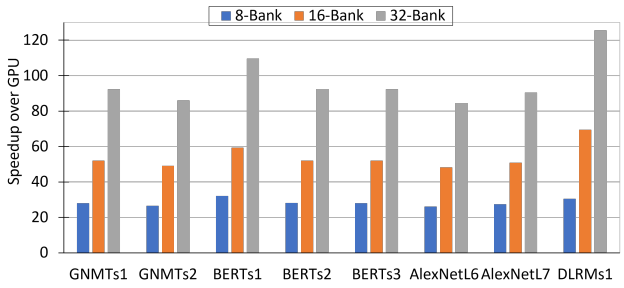

Fig. 10. Sensitivity to number of banks

the X-axis) as we vary the number of banks (individual bars in each group). While the compute bandwidth increases linearly with number of banks, the Amdahl's Law effect of the activation overheads (o in Section III-F) dampens the speedup. Newton's speedup goes from 28x (at 8 banks/channel) to 54x (at 16 banks) and finally to 96x (at 32 banks). Note, 16 banks is a typical number for DRAM design; changing the number of banks impacts area and power. Finally, if more parallelism is needed, adding channels remains an option. With additional channels, Newton benefits from the best of both worlds – increased compute parallelism without exacerbating the Amdahl's Law bottleneck. Note that additional channels improves the baseline DRAM bandwidth as well; to exploit such higher bandwidth, however, non-PIM designs may need additional compute resources as well.

D. Sensitivity to Batch Size

Newton's sensitivity to batch size is important to understand because k-way batching effectively changes the matrix-vector ($[M \times N] \times [N \times 1]$) product to a matrix-matrix ($[M \times N] \times [N \times k]$) product with significant reuse. Large batches' high reuse results in the workload being compute-bound (and thus unsuitable for *any* PIM architecture, not just Newton). However, supporting small-batch inference is important (say 8-way batches), as also argued in [20], especially for acceleration at the edge (Section I).

Figure 11 and Figure 12 compare the performance of Newton with *Ideal Non-PIM* (Y-axis) and Titan V-like GPU, respectively. In each graph the performance shown on the Y-axis is normalized to that of the Titan V-like GPU with batch size of 1, for our benchmarks (groups of bars on the X-axis). For each benchmark, we vary the batch size (individual bars). Matrix elements see k-way reuse with k-way batching, which non-PIM architectures can exploit to achieve higher performance (e.g., via caching). Newton's performance remains unchanged with batch size because Newton's compute cannot exploit the reuse to improve performance. In contrast, *Ideal Non-PIM*'s performance (Figure 11 improves with the batch size, so that *Ideal Non-PIM* nearly catches up with Newton at $k = 8$. At $k = 16$, *Ideal Non-PIM* is 1.6x faster than Newton. However, this crossover point is an artifact of the idealized nature (i.e., infinite compute) of *Ideal Non-PIM*. The realistic GPU comparison in Figure 12 shows that a large batch size of 64 is needed for the GPU to outperform Newton.

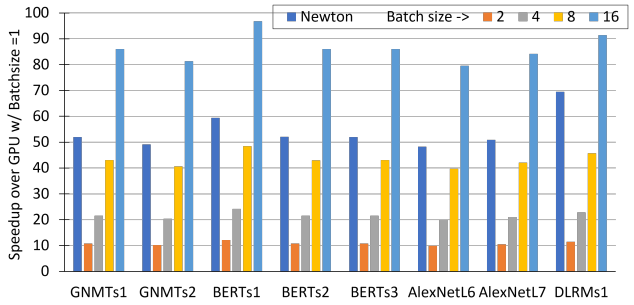

Fig. 11. Sensitivity to batch size (*Ideal Non-PIM*)

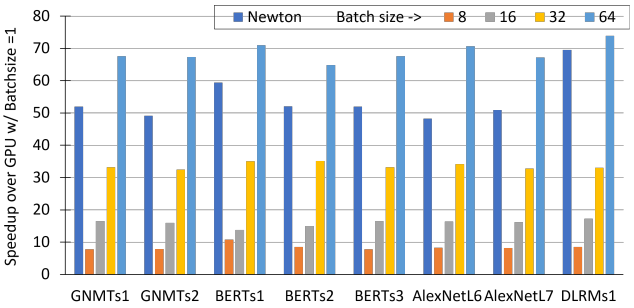

Fig. 12. Sensitivity to batch size (GPU-comparison)

Newton remains significantly faster at smaller batch sizes of 8 and lower.

This graph also implies that CNNs are unsuitable for PIM in general. CNNs have enormous filter reuse due to convolutional sliding even at low batch sizes. As such, Newton does not target CNNs.

E. Power Comparison

Given that Newton achieves 10x speedup over any non-PIM architecture, it is to be expected that Newton incurs higher average power consumption than conventional DRAM. However, any non-PIM architecture (like CPUs or GPUs) would incur additional compute power. Consequently, Newton's power is likely to remain lower in aggregate. Further, by avoiding the external transfers of the matrix data, Newton achieves higher energy efficiency. Figure 13 shows Newton's average power normalized to that of conventional DRAM (Y-axis) for each of our benchmarks (X-axis). Newton, which achieves 10x speedup over any non-PIM system, consumes only 2.8x more power on average (rightmost bar) than conventional DRAM (entirely ignoring non-PIM's compute power and external transfer power), which illustrates Newton's energy efficiency.

VI. Related Work

Application characteristics:: Previous PIM and PNM proposals explore general workloads [7], [15], [19], [35], vector workloads [9], [30], [33], MapReduce workloads [15], [32], graph workloads [3], neural networks [8], [39], [43], key-value search [18], and data reorganization [4]. These workloads do not satisfy one or more of PIM's key constraints – regularity, memory-boundedness and only one large operand.

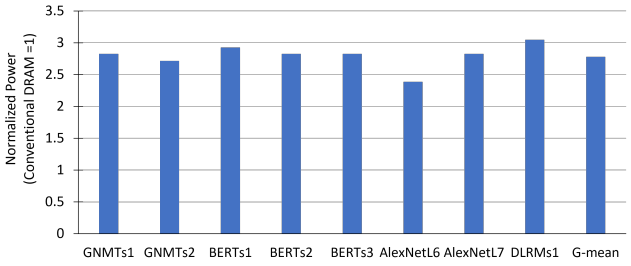

Fig. 13. Average Power

Architecture:: As discussed in Section I, analog-PIM architectures [8], [29], [38], [39], [43] incur the reliability and process variation issues associated with analog computation. Previous digital-PIM and PNM proposals employ uniprocessors [7], vectors [33], VLIW [30], GPGPUs [9], multicores [11], [19], and many-cores [2], [3], [13], [15], [16], [24], [35]. These architectures incur the area overhead of these full processors with a subset of the overheads for instruction processing and datapath control, complex superscalar pipelines, multi-level cache hierarchies, cache coherence, address translation, and interconnection network. In contrast, Newton implements a simple SIMD datapath and avoids datapath control overheads by exposing a DRAM-like interface for the host CPU to control the datapath.

Circuit-level optimizations for PIM-based ML accelerators explore binary/ternary reconfigurability [5], options for MAC unit placement in McDRAM [41], and variable bit-precision [28], [40], These optimizations may improve the circuit-level implementation of Newton. The AiM work at Hynix started in 2016 and proceeded concurrently with the McDRAM work, as shown by a few presentations on Hynix's AiM architecture which predate the codename Newton [42]. (Industry product projects typically do not provide much public documentation.) Being an industry product effort, the Newton project took longer for publication than the academic McDRAM project. While McDRAM's focus is evaluating various circuit-level placement options for the MACs in a PIM-based ML accelerator, this paper's focus is on architecture-level issues such as the command interface optimizations and input reuse.

Other proposals [1], [12], [14] explore processing in cache which does not address the memory bandwidth bottleneck seen by cache misses. In contrast, PIM, in general, and AiM, in particular, exploit higher internal memory bandwidth. Other work [21], [37] explores sparse matrix computation for large, sparse matrices, whereas AiM targets dense matrix-vector multiplication.

VII. Conclusion

While PIM promises unprecedented high-bandwidth connection between compute and memory, PIM imposes severe area and power constraints. Further, PIM fits only regular, memory-bound workloads with only one large, low-reuse operand held in memory arrays while the other operands are small and held in buffers. Fortunately, many memory-bound ML models that perform matrix-vector computations fit this constraint. Previous approaches advocate full cores which do not conform to PIM's area and power constraints.

We described the architecture and workload of Newton, a major DRAM maker's upcoming accelerator-in-memory (AiM) product for machine learning. We addressed the above issues through these contributions: (1) To satisfy PIM's area constraints, Newton (a) places a minimal compute of only multiply-accumulate units and buffers in the DRAM which avoids the full-core area and power overheads of previous work and thus makes PIM feasible for the first time, and (b) employs a DRAM-like interface for the host to issue commands to the PIM compute. The PIM compute is rate-matched to the internal DRAM bandwidth and employs a non-intuitive, global input vector buffer shared by the entire channel to capture input reuse while amortizing buffer area cost. To the host, Newton's interface is indistinguishable from regular DRAM without any offloading overheads and PIM/non-PIM mode switching, and with the same deterministic latencies even for floating-point commands. (2) To prevent the PIM-host interface from becoming a bottleneck, we include three optimizations: commands which gang multiple compute operations both within a bank and across banks; complex, multi-step compute commands – both of which save critical command bandwidth; and targeted reduction of command overhead. (3) To capture output vector reuse with reasonable buffering, Newton employs an unusually-wide interleaved layout for the matrix.

Our simulations running state-of-the-art neural networks show that building on a realistic HBM2E-like DRAM Newton achieves 10x and 54x average speedup over a non-PIM system with infinite compute that perfectly uses the external DRAM bandwidth and a realistic GPU, respectively. Our results show that each of Newton's optimizations significantly contribute to performance improvement over a GPU, collectively taking the speedup from 48% to 54x. Further, Newton is more energy-efficient than non-PIM systems. While our evaluation is based on HBM2E-like DRAM, Newton is applicable to other DRAMs, including DDR, LPDDR, and GDDR families. These results make a compelling case for AiM architectures for emerging machine learning workloads that are memory-bound.

Acknowledgment

The Purdue authors on this research project were supported in part by funding from SK Hynix Inc.

References

[1] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 481–492.

[2] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, June 2015, pp. 336–348.

[3] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 105–117. [Online]. Available: http://doi.acm.org/10.1145/2749469.2750386

[4] B. Akin, F. Franchetti, and J. C. Hoe, "Data reorganization in memory using 3d-stacked dram," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 131–143. [Online]. Available: http://doi.acm.org/10.1145/2749469.2750397

[5] K. Ando, K. Ueyoshi, K. Orimo, H. Yonekawa, S. Sato, H. Nakahara, M. Ikebe, T. Asai, S. Takamaeda-Yamazaki, T. Kuroda, and M. Motomura, "Brein memory: A 13-layer 4.2 k neuron/0.8 m synapse binary/ternary reconfigurable in-memory deep neural network accelerator in 65 nm cmos," in *2017 Symposium on VLSI Circuits*, 2017, pp. C24–C25.

[6] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, April 2009, pp. 163–174.

[7] J. B. Brockman, S. Thoziyoor, S. K. Kuntz, and P. M. Kogge, "A low cost, multithreaded processing-in-memory system," in *Proceedings of the 3rd Workshop on Memory Performance Issues: In Conjunction with the 31st International Symposium on Computer Architecture*, ser. WMPI '04. New York, NY, USA: ACM, 2004, pp. 16–22. [Online]. Available: http://doi.acm.org/10.1145/1054943.1054946

[8] 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," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 27–39. [Online]. Available: https://doi.org/10.1109/ISCA.2016.13

[9] B. Y. Cho, W. S. Jeong, D. Oh, and W. W. Ro, "XSD: Accelerating MapReduce by Harnessing GPU inside SSD," in *1st Workshop on Near Data Processing (WoNDP 2013) In Conjunction with the 46th International Symposium on Microarchitecture*, 2013.

[10] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[11] R. G. Dreslinski, D. Fick, B. Giridhar, G. Kim, S. Seo, M. Fojtik, S. Satpathy, Y. Lee, D. Kim, N. Liu, M. Wieckowski, G. Chen, D. Sylvester, D. Blaauw, and T. Mudge, "Centip3de: A 64-core, 3d stacked near-threshold system," *IEEE Micro*, vol. 33, no. 2, pp. 8–16, March 2013.

[12] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. IEEE Press, 2018, p. 383–396. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00040

[13] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, Feb 2015, pp. 283–295.

[14] D. Fujiki, S. Mahlke, and R. Das, "Duality cache for data parallel acceleration," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 397–410. [Online]. Available: https://doi.org/10.1145/3307650.3322257

[15] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 2015, pp. 113–124.

[16] 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*, ser. ASPLOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 751–764. [Online]. Available: https://doi.org/10.1145/3037697.3037702

[17] M. Gokhale, B. Holmes, and K. Iobst, "Processing in memory: The terasys massively parallel pim array," *Computer*, vol. 28, no. 4, pp. 23–31, Apr. 1995. [Online]. Available: http://dx.doi.org/10.1109/2.375174

[18] Q. Guo, X. Guo, R. Patel, E. Ipek, and E. G. Friedman, "Ac-dimm: Associative computing with stt-mram," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 189–200. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485939

[19] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park, "Mapping irregular applications to diva, a pim-based data-intensive architecture," in *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, ser. SC '99. New York, NY, USA: ACM, 1999. [Online]. Available: http://doi.acm.org/10.1145/331532.331589

[20] 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 *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080246

[21] K. Kanellopoulos, N. Vijaykumar, C. Giannoula, R. Azizi, S. Koppula, N. M. Ghiasi, T. Shahroodi, J. G. Luna, and O. Mutlu, "Smash: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 600–614. [Online]. Available: https://doi.org/10.1145/3352460.3358286

[22] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, "Flexram: toward an advanced intelligent memory system," in *Computer Design, 1999. (ICCD '99) International Conference on*, 1999, pp. 192–201.

[23] A. Kerr, D. Merrill, J. Demouth, and J. Tran, "Cutlass: Fast linear algebra in cuda c++." https://devblogs.nvidia.com/cutlass-linear-algebra-cuda/.

[24] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. IEEE Press, 2016, p. 380–392. [Online]. Available: https://doi.org/10.1109/ISCA.2016.41

[25] J. Kim and K. Tran, "Hbm: Memory solution for bandwidth-hungry processors." Presented at 'Hot Chips: A Symposium on High Performance Chips', 2014.

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[27] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin, J. H. Cho, K. H. Kwon, M. J. Kim, J. Lee, K. W. Park, B. Chung, and S. Hong, "25.2 a 1.2v 8gb 8-channel 128gb/s high-bandwidth memory (hbm) stacked dram with effective microbump i/o test methods using 29nm process and tsv," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2014, pp. 432–433.

[28] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H. Yoo, "Unpu: A 50.6tops/w unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision," in *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, 2018, pp. 218–220.

[29] 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: Association for Computing Machinery, 2017, p. 288–301. [Online]. Available: https://doi.org/10.1145/3123939.3123977

[30] 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, "Active memory cube: A processing-in-memory architecture for exascale systems," *IBM Journal of Research and Development*, vol. 59, no. 2/3, pp. 17:1–17:14, March 2015.

[31] M. Naumov, D. Mudigere, H. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy, "Deep learning recommendation model for personalization and recommendation systems," *CoRR*, vol. abs/1906.00091, 2019. [Online]. Available: https://arxiv.org/abs/1906.00091

[32] Nitin, M. Thottethodi, and T. N. Vijaykumar, "Millipede: Die-stacked memory optimizations for big data machine learning analytics," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 160–171.

[33] 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, Mar. 1997. [Online]. Available: http://dx.doi.org/10.1109/40.592312

[34] J. T. Pawlowski, "Hybrid memory cube (hmc)," in *2011 IEEE Hot Chips 23 Symposium (HCS)*, Aug 2011, pp. 1–24.

[35] 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 2014, Monterey, CA, USA, March 23-25, 2014*, 2014, pp. 190–200. [Online]. Available: http://dx.doi.org/10.1109/ISPASS.2014.6844483

[36] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, 2011.

[37] F. Sadi, J. Sweeney, T. M. Low, J. C. Hoe, L. Pileggi, and F. Franchetti, "Efficient spmv operation for large and highly sparse matrices using scalable multi-way merge parallelization," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 347–358. [Online]. Available: https://doi.org/10.1145/3352460.3358330

[38] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 273–287. [Online]. Available: https://doi.org/10.1145/3123939.3124544

[39] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 14–26. [Online]. Available: https://doi.org/10.1109/ISCA.2016.12

[40] D. Shin, J. Lee, J. Lee, and H. Yoo, "14.2 dnpu: An 8.1tops/w reconfigurable cnn-rnn processor for general-purpose deep neural networks," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, 2017, pp. 240–241.

[41] 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.

[42] "Supporting documentation of early AiM work," https://engineering.purdue.edu/~mithuna/NewtonSupportingDocs/, SK Hynix Inc.

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

[44] H. S. Stone, "A logic-in-memory computer," *Computers, IEEE Transactions on*, vol. C-19, no. 1, pp. 73–78, Jan 1970.

[45] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, Łukasz Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, "Google's neural machine translation system: Bridging the gap between human and machine translation," 2016.