JIT-Q: Just-in-time Quantization with Processing-In-Memory for Efficient ML Training

Mohamed Assem Ibrahim Shaizeen Aga Ada Li Suchita Pati Mahzabeen Islam [1]

Abstract

Data format innovations have been critical for machine learning (ML) scaling, which in turn fuels ground-breaking ML capabilities. However, even in the presence of low-precision formats, model weights are often stored in both high-precision and low-precision during training. Furthermore, with emerging directional data-formats (e.g., MX9, MX6, etc.) multiple low-precision weight copies can be required. To lower memory capacity needs of weights, we explore just-in-time quantization (JIT-Q) where we only store high-precision weights in memory and generate low-precision weights only when needed. To perform JIT-Q efficiently, in this work, we evaluate emerging processing-in-memory (PIM) technology to execute quantization. With PIM, we can offload quantization to in-memory compute units enabling quantization to be performed without incurring costly data-movement while allowing quantization to be concurrent with accelerator computation. Our proposed PIM-offloaded quantization keeps up with GPU compute and delivers considerable capacity savings (up to 24%) at marginal throughput loss (up to 2.4%). Said memory capacity savings can unlock several benefits such as fitting a larger model in the same system, reducing model parallelism requirement, and improving overall ML training efficiency.

1 Introduction

Model scaling has been a critical component to unlocking disruptive machine learning (ML) capabilities. This scaling, however, has not been matched with commensurate memory capacity scaling (Rajbhandari et al., 2021). These conflicting trends have led to lower efficiency for ML due to increased reliance on distributed computing (communication overheads), lower batch-sizes (lower compute efficiency) and more. As such, techniques which optimize memory capacity needs of ML stand to lower these overheads and can lead to increased ML efficiency.

To tackle the memory capacity challenge, there has been considerable interest in quantization of tensors to (increasingly) low-precision numeric formats such as BF16 (Kalamkar et al., 2019), FP8 (Micikevicius et al., 2022), and beyond. By storing and computing on tensors in low-precision formats instead of single-precision (FP32) format, considerable capacity savings and also compute efficiency can be attained. As such, data-formats continue to be an active area of investigation, with emerging shared microexponents (MX) formats (Darvish Rouhani et al., 2023) further pushing the precision down to four bits.

While this data-format evolution has been a key lever in optimizing ML capacity needs, we believe there exists data redundancy in ML training which can be optimized for capacity savings. Specifically, state-of-the-art training techniques which employ low-precision formats, typically employ mixed-precision training (Micikevicius et al., 2017), wherein to attain good accuracy, weight tensors are maintained in both high-precision and low-precision in memory. The high-precision copy of weights accumulates the gradients after each optimizer step. At the same time, a quantized low-precision copy of these high-precision weights is maintained, which is employed in computations for forward and back-propagation (Figure 1 Ⓐ) phases. Further, with directional (MX) data-formats, which require quantization to be applied along the reduction dimension, two low-precision copies of weights, to be used in forward and back-propagation respectively, can be necessary (Figure 1 Ⓑ).

An observation we make in this work is that as the high-precision copy of weights is necessary for training, and the low-precision copy is derived from the high-precision copy, a mechanism to cheaply create a low-precision copy of weights, as and when needed, can obviate the need to store the low-precision copy in memory and thus save memory capacity. We term this *just-in-time* quantization (JIT-Q). Note that, as in Figure 1 Ⓒ, while a straightforward way to accomplish this is to read-in the high-precision weight copy for computations and quantize the weights before use at the core, this causes increased (and unnecessary) data-

[1] Advanced Micro Devices, Inc.. Correspondence to: Mohamed Assem Ibrahim <mohamed1.ibrahim@amd.com>.

Proceedings of the 7^{th} MLSys Conference, Santa Clara, CA, USA, 2024.

movement, which is the chief contributor to ML energy expenditure (Untether AI, 2023). Further, this also places quantization on the critical path. While this can be mitigated by scheduling quantization kernels ahead of time and in concurrence with main GPU computation, doing so causes even more data-movement as not only will the high-precision copy be read, the low-precision copy will also be written.

Instead, in this work, we propose harnessing emerging processing-in-memory (PIM) technology to perform the aforementioned JIT-Q of weight tensors. With recent functional PIM prototypes from multiple memory vendors (Lee et al., 2021; 2022), commands are broadcast to in-memory compute units and data is operated in-place in memory instead of moving data to the accelerator (e.g., GPU). While the accelerator coupled with memory can access one memory bank at a time over a shared data bus, by not using the shared data bus, PIM enables data from multiple banks to be operated on in tandem. This provides considerable memory bandwidth boost over that available to accelerator all the while enabling computation over data without incurring costly data-movement. In this work we focus on high-bandwidth memory (HBM) PIM (Lee et al., 2021), as HBM is coupled with GPUs, the most ubiquitous ML accelerators.

As depicted in Figure 1 Ⓓ, PIM enables harnessing capacity savings of JIT-Q of weights without incurring costly data-movement. To do so, we first deduce a quantization routine that can be offloaded to in-memory compute units in HBM. We identify data-placement considerations, for both scalar formats and directional blocked formats, and in-memory ALU augmentations that lead to an efficient quantization routine. Further, we discuss how this routine can be co-scheduled with the main GPU computation to deliver quantized low-precision weight tensors just-in-time to the GPU computation.

Our analysis across current and future models shows that our proposed PIM-offloaded quantization keeps up with main GPU computation and delivers considerable capacity savings (up to 24%) at marginal throughput loss (up to 2.4%). Resultant memory capacity savings unlock benefits that improve overall compute utilization; it enables devices to fit larger models and/or larger batch-sizes, and reduces reliance on model parallelism which requires additional communication.

Overall, our work makes the following key contributions:

- We propose just-in-time quantization (JIT-Q) for weights which enables storing of only high-precision copy of weights during training and creates low-precision copies just when they are needed.
- We evaluate the efficacy of emerging commercial processing-in-memory (PIM) solutions to perform JIT-Q. Offloading to PIM enables quantization of weights without incurring costly data-movement.
- We discuss data-placement and in-memory ALU augmentations necessary to offload quantization to PIM efficiently. Our evaluations show that our proposed PIM quantization routine can prepare low-precision weights copy just-in-time without stalling concurrent GPU computation.
- The proposed JIT-Q of weights delivers memory capacity savings of up to 24% which can be harnessed in many ways for efficient ML training.

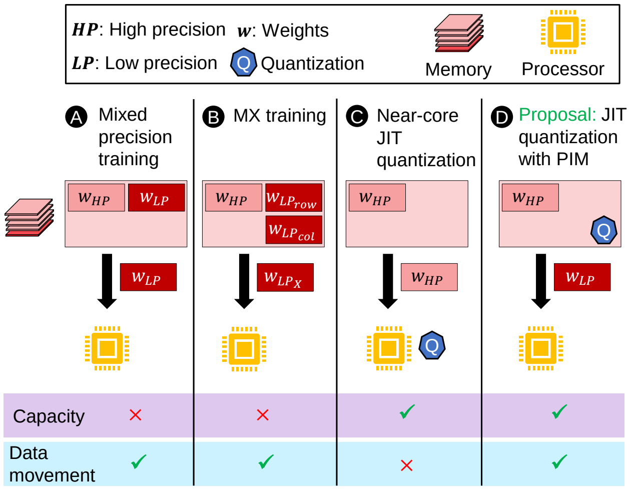

Figure 1. JIT-Q achieves capacity savings while maintaining the data-movement savings.

2 Background

2.1 Large Language Models

We focus in this work on Transformer-based (Vaswani et al., 2017) large language models (LLMs) given their applicability across domains and modalities (Tsimpoukelli et al., 2021; Sung et al., 2022; Alayrac et al., 2022). From a computational perspective, LLMs have two training phases, an expensive, but one-time, pre-training phase for general learning and another short task-specific fine-tuning phase. Post learning, LLMs are deployed for inference. The basic building block of an LLM is an *encoder* or a *decoder* layer. These layers are made of a multi-head attention sub-layer and a multi-layer perceptron (MLP) sub-layer. Operations in these sub-layers manifest as matrix multiplication operations (GEMMs) followed by a few element-wise and reduction operations (e.g., residual connection, layer normalization) which are often fused with the GEMMs. The encoder and decoder layers are similar except the decoder's attention sub-layer/GEMM input is masked, which causes different inference behavior computationally but does not affect training which is the focus of this work. Training phase involves forward and backward propagation through the layers, followed by parameter updates. In contrast, infer-

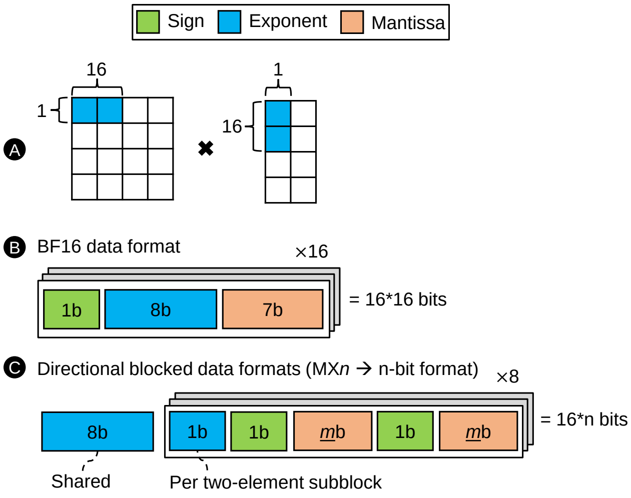

Figure 2. Directional data formats.

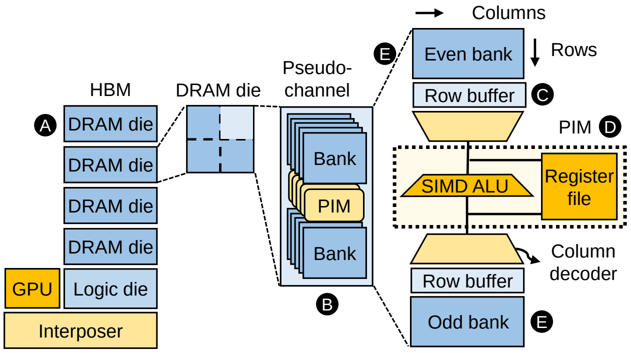

Figure 3. HBM PIM overview.

ence phase does not have backward propagation and weight update. Note, while we anchor on LLMs, ideas proposed in this work are applicable to other model architectures.

2.2 Directional Blocked Data Formats

Continued scaling of ML models has been a key ingredient to their disruptive capabilities. A critical fuel to this scaling is the evolution of low-precision data formats. Low-precision formats reduce memory capacity requirement and consequently data-movement along with delivering performance improvement by harnessing higher throughput compute. While an active area of research, in this paper, we focus on shared microexponents (MX) format (Darvish Rouhani et al., 2023), an emerging low-precision data format, because of the balance it achieves between maintaining model accuracy, improving hardware efficiency, while reducing software friction.

As shown in Figure 2, MX formats are directional blocked data formats that represent a block of N elements (N=16 BF16 elements in the figure). Instead of using a per-element sign, exponent, and mantissa bits as in scalar data formats Ⓑ, MX formats settles for only sign and mantissa bits per element Ⓒ. Based on the number of mantissa bits, there are three variants of MX formats to tailor to the different needs of training and inference. Specifically, MX9, MX6, and MX4 use 7, 4, and 2 mantissa bits per element, respectively. As for exponent bits, MX format uses two-levels of scaling factors (exponents) to reduce the negative effects of outliers and provides additional quantization noise reduction. That is, an 8b exponent is shared across *all* N elements, and a second-level 1b exponent is shared among a subblock of two elements. Finally, as shown in Ⓐ, for a GEMM operation where both interacting tensors are MX-quantized, to harness benefits of MX formats, the tensors have to be quantized along the reduction dimension (blocking/quantization along row for one, blocking/quantization along column for other).

2.3 Commercial PIM Solutions

To cater to increasing demands for memory bandwidth from emerging applications, memory vendors like Samsung and SK Hynix have proposed commercial processing-in-memory (PIM) solutions for DRAM based memories (Lee et al., 2021; Kim et al., 2022; Lee et al., 2022; 2019; He et al., 2020). With PIM, compute units are placed on the periphery of core DRAM structures, thus avoiding changes to internal DRAM structures and making commercialization easier. In a recent real PIM prototype (Samsung, 2022), Samsung has integrated PIM with high bandwidth memory (HBM), which is coupled with GPUs. Figure 3 depicts the PIM design we focus on in this work.

HBM provides high bandwidth and energy efficiency via high density interconnects and in-package 2.5D integration with the processor (JEDEC, 2013) Ⓐ. HBM memory access requires similar set of basic operations as conventional DRAM; however, HBM offers wider interface. Each HBM die comprises multiple pseudo-channels (pCHs) Ⓑ. Each pCH contains multiple banks sharing the data bus of the pCH. Further, similar to generic DRAM, a bank is comprised of multiple rows and columns. On a memory access from the GPU, a row worth data is brought into the per-bank row-buffer Ⓒ (incurring row activation overhead) and from there on column worth data (or multiples of it) can be accessed over the data bus from the GPU.

In the HBM-PIM prototype (Lee et al., 2021), each PIM unit includes a 256b wide SIMD PIM ALU (processing on multiple lanes in parallel) and limited number of register files to temporarily store data Ⓓ. To harness performance while managing area overheads, each PIM unit is shared by two banks (even and odd) Ⓔ. To reduce complexity, the PIM units do not have any instruction fetch capabilities, rather PIM commands are sent by the GPU to the pCH. Each PIM command is broadcast to all PIM units inside the

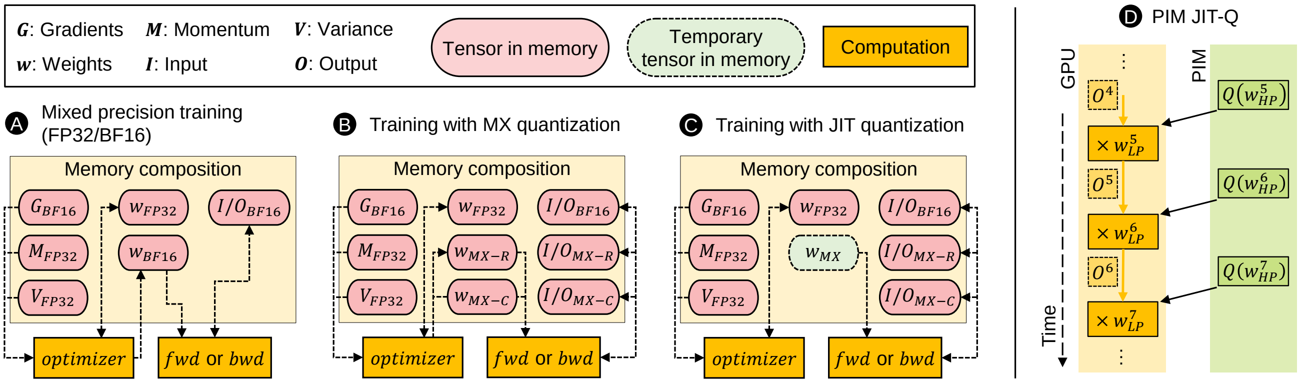

Figure 4. Memory composition, dataflow, and redundancy in state-of-the-art training techniques: mixed-precision training Ⓐ and training with MX-quantization Ⓑ. Proposed work optimizes weight redundancy with just-in-time quantization (JIT-Q) Ⓒ. Concurrent GPU and PIM execution for efficient JIT-Q of weights Ⓓ.

pCH and the PIM units operate in parallel. Therefore, PIM has bandwidth advantage over GPU (about 4-8×) - GPU memory accesses to different banks of a pCH get serialized over the memory interface, in contrast all PIM units can independently access the attached banks. To avail such bandwidth benefit, PIM operable data needs to be allocated in large physical pages encompassing all the channels and banks. For example, employing a 2MB page size covers current, and potential future, memory configurations. This bandwidth boost can be harnessed to offload bandwidth-intensive, low compute-to-byte, computations to PIM, while keeping compute-bound phases on GPU. Such collaboration is promising for workload acceleration (Aga et al., 2019).

3 Case for Just-in-time Quantization with PIM

3.1 Capacity - A Key Performance Determinator

Memory characteristics (capacity, bandwidth, latency) play a critical role in ML training efficiency. Specifically, in this work, we focus on capacity, which dictates the amount of data, and the associated computation, that gets mapped to an accelerator and as such computation efficiency. Model scaling in the recent past has surpassed capacity scaling, especially considering HBM capacity that is coupled with accelerators such as GPUs, which are commonly employed for ML training. This in turn has caused training state (optimizer state such as momentum, variance, gradient, weight tensors, and intermediate state such as input, output tensors) to be sharded (Narayanan et al., 2021; Zhao et al., 2023) across accelerators or offloaded away from HBM (Ren et al., 2021). Such tensor sharding/offloading can have considerable impact on ML training efficiency. Specifically, such mechanisms reduce overall compute efficiency; communication overhead to gather/scatter tensors, smaller tensors due to sliced weights and lower batch. Given its multi-dimensional impact, optimizing ML capacity needs is paramount to attaining better ML efficiency.

3.2 Opportunity: Eliminating Weight Redundancy

Prior works have optimized away memory redundancy in ML training to better support model scaling. As an example, works like ZeRo (Rajbhandari et al., 2020), store a single copy of optimizer state partitioned across accelerators in a distributed training setup. While these prior works have considerably reduced memory redundancy, in this work, we observe that there is still further scope to reduce memory redundancy for ML training. Consider memory composition of an accelerator for mixed-precision training (Micikevicius et al., 2017), the de facto training technique, as depicted in Figure 4 Ⓐ. With mixed-precision, weight tensors manifest redundancy as both high and low-precision copies of weights are stored in memory to be used in the optimizer computation and forward/back-propagation computations, respectively. This redundancy is worsened for training with directional blocked formats such as MX formats (described in Section 2.2) for two variations of low-precision weights, quantized along different dimensions, are required (Figure 4 Ⓑ). Note that, as high-precision weight tensors are necessary for effective training (to preserve small-valued updates), a cheap mechanism which creates low-precision weight tensors only when needed can eliminate weight tensor redundancy and deliver capacity savings.

3.3 PIM for Just-in-time Weight Quantization

The above identified opportunity can be harnessed with just-in-time quantization (JIT-Q) of weight tensors. That is, by storing only high-precision weight tensor in memory and creating the low-precision weight tensor only when needed (Figure 4 Ⓒ), weight tensor redundancy can be eliminated, and capacity savings can be harnessed. However, to truly realize the benefits of this approach, an efficient mechanism to create low-precision weight tensor is necessary. To that end, note first that a naive mechanism which reads in high-precision weight tensor during forward and back-propagation computation causes unnecessary data-movement and energy expenditure, and additionally adds quantization to critical path. While the latter of the problems can be tackled by co-scheduling quantization kernels ahead of time, such kernels, when executed by the accelerator will further add to data-movement overheads by having to write-back low-precision weights to memory, as well as contending with the concurrent main computation for compute/memory resources.

To tackle above challenges, in this work, we propose to harness processing-in-memory (PIM) technology and offload JIT-Q to in-memory compute units. With PIM, high-precision weight tensors are read by in-memory compute units, which quantize them to low-precision and store back the result to memory without incurring costly data-movement from memory to the processor. By co-scheduling PIM computation with main acceleration computation (Figure 4 Ⓓ), said low-precision tensors are only *temporarily* created just when needed and discarded thereafter. Overall, this enables JIT-Q of weights to harness capacity savings while avoiding costly data-movement. However, to do so, quantization should be efficiently offloaded to PIM so as to keep up with main acceleration computation. We discuss techniques to do so next.

4 Efficient JIT-Q with PIM

To efficiently offload a computation to PIM, a programmer first deduces a computation-conscious data mapping to exploit the strengths of the target PIM design (e.g., command broadcasts) while avoiding its shortcomings (e.g., no inter-bank communication and no cross SIMD compute). Next, a *PIM kernel* which expresses the computation orchestration on the in-memory compute units is launched. In this section, we discuss the key considerations for offloading JIT-Q computation to PIM in terms of compute orchestration, data mapping, and augmentations to existing PIM design.

4.1 PIM Quantization Routine

To initiate the required PIM computations on commercial PIM designs, host GPU launches *PIM kernels* (Lee et al., 2021; Samsung, 2022). These kernels are like existing GPU kernels except they issue *pim instructions*. A *pim instruction* effectively enqueues a *pim command* at the memory controller which in turn instructs PIM unit to execute either SIMD compute operation (e.g., add, multiply, etc.) or data-movement (moving data between row-buffer and register file) along with necessary row activation. Finally, to feed the independent memory channels in GPUs, different workgroups (groups of threads or thread blocks), within the PIM kernel, issue PIM commands to different channels, which in turn are broadcast to banks within a channel. In addition to PIM ALU functionality supported in current PIM prototypes (Lee et al., 2021), to effectively support quantization, we assume the PIM ALU to support two operations: compare two operands (*pim-CMP*) and single-bit intra-SIMD lane shift operation (*pim-bitSHIFT*). Given their simplicity and support for related functionality in existing PIM prototypes, such as activation functions in Hynix-PIM (Lee et al., 2022) and ReLU in Samsung-PIM (Lee et al., 2021), we believe this to be a reasonable assumption.

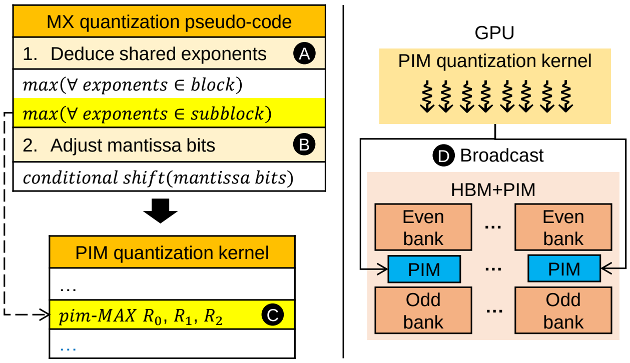
MX quantization pseudo-code
1. Deduce shared exponents Ⓐ
$max(\forall\ exponents \in block)$
$max(\forall\ exponents \in subblock)$
2. Adjust mantissa bits Ⓑ
$conditional\ shift(mantissa\ bits)$
PIM quantization kernel
...
$pim\text{-}MAX\ R_0, R_1, R_2$ Ⓒ
...
GPU
PIM quantization kernel
Ⓓ Broadcast
HBM+PIM
Even bank
...
Even bank
PIM
...
PIM
Odd bank
...
Odd bank

Figure 5. MX quantization pseudo-code and resultant PIM quantization kernel (left). PIM quantization kernel orchestration (right).

We describe this PIM routine for the MX format (Section 2.2) as tackling quantization for them is more involved than for scalar formats. For MX quantization from scalar data formats, the input tensor is broken into blocks of N elements (N=16). Then, for each N-element block, two key steps are performed as shown in Figure 5. First, the quantization routine computes the shared level-1 exponent using a reduction function (e.g., max) of all exponents in the N input elements Ⓐ. Then, using this level-1 exponent, level-2 exponents are deduced for every two-input element block. Second, the per-element mantissa bits are adjusted, using bit-level shift operations, to compute the m-bit mantissa per output element Ⓑ. We translate this pseudo-code to PIM quantization kernel via deducing the PIM instructions necessary to realize the computation. As an example, for exponent calculation, a series of *pim-MAX* commands Ⓒ are used, while for mantissa adjustment *pim-bitSHIFT* commands are used. Using our PIM kernel, we broadcast these commands to banks in a channel (and to multiple channels) to fully exploit PIM parallelism Ⓓ.

Optimization. In contrast to scalar data format quantization, the shift amount can differ for each input element as it depends on the level-2 exponent and the per input element's original exponent. To efficiently support the different bit-level shift amounts, we augment the SIMD PIM ALU

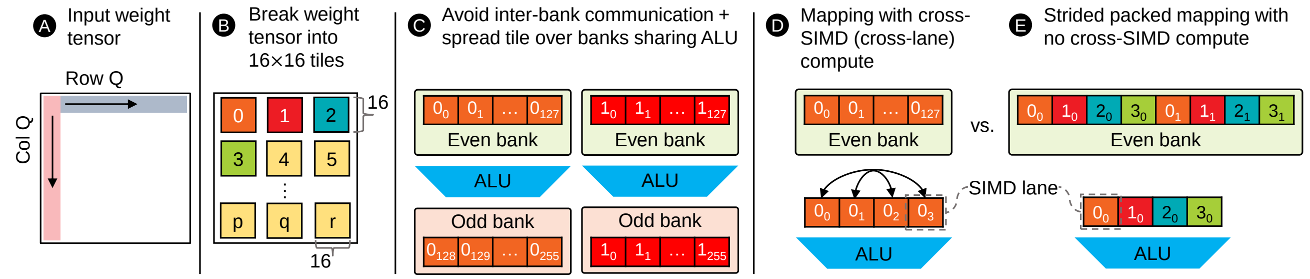

Figure 6. Placement of weight tensor in memory for efficient offloading of MX quantization to PIM.

with counter-based conditional intra-lane shifts. This stores the per-lane shift amount in a register, then per each *pim-bitSHIFT* instruction, it checks the stored shift amount per lane (S_i). If $S_i > 0$, then the shift is processed for the corresponding lane, then S_i is decremented. Otherwise, the shift is skipped for the corresponding lane. Without such support, three PIM instructions (*pim-CMP*, *pim-bitSHIFT*, and *pim-ADD*) are used for a single bit shift, which increases the PIM quantization time (evaluated in Section 5).

4.2 Weight Tensor Placement

Placement of data in memory is an important consideration for efficient computation offload to PIM for several reasons. First, our evaluated PIM design (Lee et al., 2021) places a PIM compute unit, with a SIMD ALU, per two DRAM banks. Therefore, unless any interacting elements in the offloaded computation are mapped to the same bank (or banks sharing a PIM unit), inter-bank communication overhead is incurred. With the absence of direct inter-bank communication in current commercial PIM designs, the GPU performs this communication by copying data from one bank to another, which is expensive. Second, PIM broadcasts the same command to multiple banks in the same pseudo-channel to achieve its significant memory bandwidth boost. Therefore, proper interleaving of input/output data across banks/channels is required to harness the broadcast feature of PIM. Finally, while the SIMD ALU in the PIM compute unit helps harness data parallelism, it lacks support for cross-lane computations, resulting in lane-level shift operations. With the limited number of metal layers in the current DRAM technology, supporting different levels of bit shifts (single-bit shift for mantissa bits processing and lane-level shift for cross-SIMD computations) is likely to be costly. Given these considerations, we discuss our data mapping tailored to JIT-Q of MX formats as shown in Figure 6.

Avoiding Inter-bank Communication. First, as MX formats require the input weight tensor to be quantized along the reduction dimension, supporting both row and column quantization is required as discussed in Section 2.2. Therefore, the overall quantization computation involves elements along both row and column dimensions Ⓐ. A naive MX-oblivious row-major mapping of the input tensor that divides the elements among the available PIM ALUs, to exploit the inherent parallelism of PIM, would map the row elements to same bank (or banks sharing a PIM compute unit) but not the column elements thus, triggering inter-bank communication during column quantization. To avoid this, we propose a tiled data mapping in which we break the input tensor into $N \times N$ tiles Ⓑ, where N is number of elements per MX block, and map each 2D tile to a single bank Ⓒ. The per-tile elements are mapped in a row-major fashion. We term this tiled mapping as *pim-jitq-tiled*. Using row-major mapping within a tile is beneficial for row quantization as it minimizes the number of occupied DRAM rows per MX block, reducing the row activations overhead. In contrast, it results in higher row activations overhead for the column quantization. To reduce this overhead, we exploit the sharing of the PIM ALU between pair of DRAM banks and propose to spread the input tile over the even and odd banks Ⓒ.

Harnessing PIM Parallelism. To unlock PIM's full potential, we process multiple independent tiles in different PIM units via command broadcasting. The large weight tensors in state-of-the-art LLMs guarantee that there are sufficient tiles to concurrently utilize all available PIM units.

Avoiding Cross-SIMD Compute. As discussed in Section 4.1, to compute the exponents on the MX block, the individual exponents of the elements are compared. With only *pim-jitq-tiled*, elements of the same MX block (or subset of) are mapped to same DRAM word, resulting in cross-SIMD computations Ⓓ. To avoid that, we propose to stride the tile across DRAM words, mapping each element to the same lane in each DRAM word Ⓔ. This ensures that elements of the same tile are aligned. To eliminate the memory waste due to utilizing a single SIMD lane, we pack elements from independent tiles in the same DRAM word Ⓔ. We term this mapping as *pim-jitq-strided*. As discussed above, the tensors from the evaluated LLMs have enough tiles to fully pack the SIMD lanes.

Note that, our proposed weight tensor placement spreads weight tensor across memory channels/banks and as such, as in baseline, can exploit memory parallelism effectively.

5 Evaluation

5.1 Methodology

5.1.1 *Performance Models*

We analyze performance using analytical models as PIM is currently only available as part of functional prototypes (Samsung, 2022). Additionally, we aim to study highly optimized GPU implementations of the evaluated LLMs to provision a stronger GPU baseline to show PIM benefits over. This makes relying on GPU simulators difficult and lends well to analytical models. Furthermore, using GPU simulators to model end-to-end ML training with large model sizes can be impractical as a single GEMM simulation can take several hours to multiple days (Avalos Baddouh et al., 2021; Villa et al., 2021) based on model/input size. Also, solutions based on kernel sampling (Avalos Baddouh et al., 2021) are insufficient given the need to simulate the overlapping impact of JIT-Q and other kernels (as discussed in Section 5.1.3). This makes simulating an entire transformer block with several kernels more challenging.

GPU Performance Model. Our baseline GPU performance is assumed to be max(GPU compute time, GPU memory time), where GPU compute time considers GEMM operations (multiplies and adds) with peak compute throughput, while memory time considers only reading GEMM's input tensors from HBM with 90% of peak memory bandwidth. In other words, we assume executing the vector operations in the transformer block, as well as writing the output of the transformer block's computation, to be free. This is because, in optimized implementations, we observe that the non-GEMM (element-wise) operations are increasingly getting fused with the GEMMs, to avoid kernel launch and global memory access overheads. Additionally, to have an even stronger GPU baseline, we assume zero-overhead concurrent execution of weight and input gradient computations in the backward propagation phase of training which results in shorter LLM execution time on GPU.

PIM Performance Model. We assume a PIM architecture in which the GPU issues PIM commands as special load/store accesses which bypass the caches and are issued in-order by the memory controller to multiple banks in parallel (Lee et al., 2021). We take a detailed DRAM command orchestration approach in which, for a given weight matrix, we consider necessary data mapping (Section 4.2) and orchestration (Section 4.1). Next, we deduce the exact DRAM commands needed to orchestrate the computation. We augment a detailed DRAM model for modeling PIM instruction timing that incorporates the PIM DRAM timing restrictions, including row activation overheads. We assume the parameters listed in Table 1. Note that we assume a PIM-aware GPU which can issue *pim-instructions* and *pim-commands* at issue-rate. With the available thread parallelism at the GPU, we believe this to be a reasonable assumption.

Table 1. Parameters for performance model (JEDEC, 2023).

#Banks per Stack (4-high)	512
Bandwidth per Pin	4.8 Gb/s
GPU Memory Bandwidth per Stack	614.4 GB/s
Row Buffer Size	1024 B

DRAM Parameters	tRP = 15ns, tCCDL=3.33ns,
	tRAS=33ns

PIM Parameters	#PIM Units per Stack = 256 #PIM Registers per ALU = 16

Table 2. Future LLMs. L = #layers, H = hidden dimension, A = #attention heads, P = #parameters, SL = sequence length, B = batch size, TP/PP = tensor/pipeline parallelism degree.

LLM	L	H	A	SL	B	TP	PP
future-1T	80	32K	128	4K	1	128	2
future-10T	200	64K	128	8K	1	128	8
future-100T	500	128K	128	16K	1	128	32

5.1.2 *Evaluated LLMs*

In this work, we evaluate LLMs of different hyperparameter combinations and distributed setups for training. Specifically, we evaluate behavior of models similar to the following LLMs (model name-parameter count): bert-345M (Devlin et al., 2019), gpt-2-1.5B (OpenAI, 2019), mega-lm-8.3B (Shoeybi et al., 2020), t-nlg-17B (Microsoft, 2020), gpt-3-175B (Brown et al., 2020), mega-nlg-530B (Smith et al., 2022), and palm-540B (Chowdhery et al., 2022). Additionally, given the trend of how LLMs evolve, we project three future LLMs of sizes 1 trillion (future-1T), 10 trillion (future-10T), and 100 trillion (future-100T) parameters in Table 2. For these future models, we scale the number of layers, hidden dimension, and sequence length.

5.1.3 *Evaluated Metrics*

JIT-Q Slack. To showcase the efficacy of offloading JIT-Q to PIM, we evaluate JIT-Q *slack* as shown in Figure 7. We define JIT-Q slack as the difference between the execution of GEMM i on GPU, and the quantization time of the weights of the next GEMM i+1 on PIM 🅐. Bigger slack indicates faster quantization on PIM. Throughout this section, for simplicity, we model the next transformer block quantization instead of the next GEMM as shown in 🅑. That said, JIT-Q at the operator level should at least retain the capacity savings gained at the transformer block level.

Memory Capacity Savings. We evaluate the capacity sav-

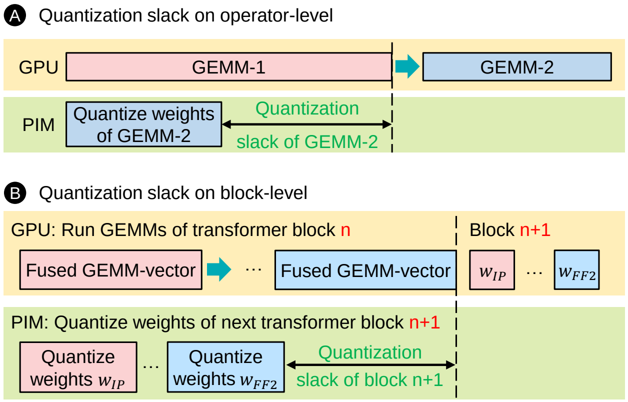
A Quantization slack on operator-level
GPU
GEMM-1
GEMM-2
PIM
Quantize weights of GEMM-2
Quantization slack of GEMM-2
B Quantization slack on block-level
GPU: Run GEMMs of transformer block n
Block n+1
Fused GEMM-vector
Fused GEMM-vector
w_{IP}
w_{FF2}
PIM: Quantize weights of next transformer block n+1
Quantize weights w_{IP}
Quantize weights w_{FF2}
Quantization slack of block n+1

Figure 7. Quantization slack indicates PIM ability to perform JIT-Q without stalling concurrent GPU computation.

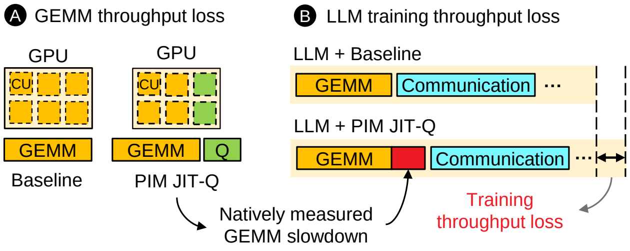

Figure 8. Compute interference between GPU and PIM execution can negatively impact training throughput.

ings unlocked by JIT-Q on PIM due to storing a single temporary copy of the quantized weights of the next transformer block. For that, we estimate the capacity consumed by weights, gradients, activations, and optimizer state based on the following. We assume a mixed precision training setup similar to FP8-based training (Wang et al., 2018; Mellempudi et al., 2019; Graphcore, 2023) with BF16 high-precision master weights and optimizer state, MXn low-precision weights, MXn activations for GEMMs, and FP8 otherwise. Additionally, for weights, we assume two copies for row and column quantization. Finally, for activations, we assume a selective recomputation strategy (Korthikanti et al., 2022). We also discuss implications with other mixed precision setups (e.g., FP32 master weights) in Section 5.6.

Training Throughput Loss. As PIM quantization kernel is orchestrated by the GPU, running other training computation on GPU concurrently will require GPU compute resources (compute units or CUs or GPU cores) to be shared amongst the two as shown in Figure 8. This loss of compute resources, along with contention for shared HBM resource can slowdown the concurrent GPU computation Ⓐ. This in turn, can lead to training throughput loss Ⓑ.

To evaluate this loss in LLM training throughput, we natively measure how the interference between GPU and PIM execution can slowdown the execution on GPU. Our system setup consists of an AMD Instinct™ MI210 Accelerator comprising GPU with 104 CUs and four stacks of HBM2E memory for a total capacity of 64GB and a peak memory bandwidth of 1638.4 GB/s (AMD, 2023a). First, we use Omniperf (AMD, 2023b), a system performance profiling tool for machine learning/HPC workloads running on AMD MI GPUs, to measure the slowdown of GPU compute while disabling 16 out of available 104 CUs. Our experiments show that a single CU can sustain the necessary PIM command issue rate for two physical HBM channels. As such, with 32 channels in our system, we set aside 16 CUs to orchestrate PIM kernel. Note that software and hardware optimizations can lower the CUs needed for PIM orchestration, but we make a conservative assumption of 16 CUs to study worst-case resource requirement for PIM. Second, we scale our modeled GPU compute time by the measured slowdown. Specifically, the execution on GPUs observes a slowdown only while concurrently running with PIM quantization. For example, with a large JIT-Q slack, the GPU and PIM execution interfere for a small portion of the execution, in which the GPU computations observe slowdown. Third, we estimate the LLM training time and its breakdown using an analytical performance model for LLMs as proposed in (Pati et al., 2023). Finally, with the updated GPU execution slowdown, we estimate the training throughput loss.

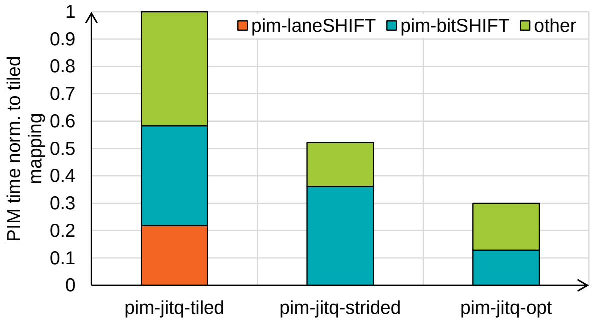

Figure 9. PIM row quantization time of the different weight tensor placements and optimizations (for BF16 to MX6).

5.2 Evaluating PIM Mapping & Orchestration

In this section, we evaluate the performance of our proposed PIM JIT-Q mapping and orchestration. Specifically, we compare the tiled mapping (*pim-jitq-tiled* in Section 4.2), strided tiled mapping (*pim-jitq-strided* in Section 4.2), and strided mapping with the optimization in Section 4.1 (*pim-jitq-opt*). Figure 9 depicts the PIM quantization time of these PIM flavors normalized to baseline tiled mapping (*pim-jitq-tiled*) on the y-axis for a representative LLM. Further, we break down the PIM quantization time into that spent executing

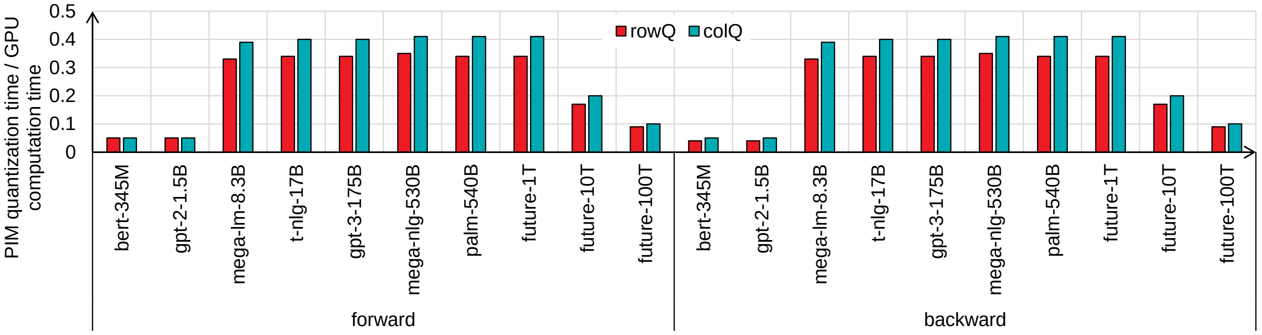

Figure 10. PIM JIT-Q slack (for BF16 to MX6).

shift-related PIM instructions (*pim-laneSHIFT* and *pim-bitSHIFT*), and rest of the time as *other* (contains DRAM row-open overhead, data-movement from register to row-buffer, compare, etc.). We observe that *pim-jitq-strided* is superior to *pim-jitq-tiled* (48% lower quantization time) because of avoiding lane-level shifts (i.e., *pim-laneSHIFT* instructions). Also, using *pim-jitq-strided* not only eliminates *pim-laneSHIFT*, but reduces other PIM instructions as well. For example, by packing multiple tiles with strided mapping, fewer *pim-CMP* instructions are used to process the packed tiles compared to unpacked mapping. Also, utilizing the augmented PIM design *pim-jitq-opt* (Section 4.1) further reduces PIM quantization time to 70% lower compared to *pim-jitq-tiled* as it reduces the number of instructions required for bit-level shifting to a single PIM instruction.

5.3 PIM Quantizes without Stalling GPU

Next, we evaluate if, via offload to PIM, weights can be quantized without stalling the GPU execution using the JIT-Q slack metric (Section 5.1.3). Figure 10 shows the JIT-Q slack for BF16 to MX6 as PIM quantization time normalized to GPU compute time (y-axis, lower is better) for the evaluated LLMs (x-axis). We observe that quantization with PIM exhibits sufficient slack for both forward and backward phases. Specifically, on average, we observe that the GPU computation time is at least 2.4× the quantization time with PIM. The large slack holds for narrower MX4 and wider MX9 formats with GPU time at least 1.6× and 3.7× PIM time, respectively (not shown).

We also observe that the varying slack across the evaluated LLMs is due to their sequence length (SL) and batch-size (B), which affect the quantization work on PIM and the compute work on GPU differently. Specifically, increasing SL or B, as in bert-340M, gpt-2-1.5B, or models larger than future-1T, increases the input tensor size, but does not impact weight tensors. Thus, GEMM compute increases, while quantization time on weights remains the same.

Additionally, we evaluate the performance of row and column quantization. We observe row quantization to have better slack compared to column quantization under forward and backward phases. This is due to the tiled data mapping discussed in Section 4.2. Specifically, with the row-major mapping of the tile in a bank, column elements end up in multiple DRAM rows resulting in additional DRAM row opens for column quantization. This results in an 18% increase in column quantization time compared to row quantization.

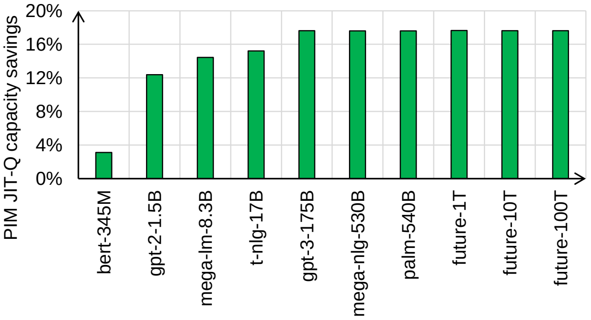

Figure 11. PIM JIT-Q capacity savings (for BF16 to MX6).

5.4 Capacity Savings of PIM JIT-Q

Figure 11 shows capacity savings for BF16 to MX6. We observe a significant capacity savings of 12.5%, on average, which continues to hold for future LLMs with trillions of parameters (up to 17.6%). For MX4 (not shown), the average capacity savings drops to 9% (up to 12.5%) as the overall capacity overheads of the low-precision weights gets smaller. In contrast, for MX9 (not shown), the average capacity savings increase to 16.8% (up to 24.2%).

The capacity savings unlock multiple benefits (not in tandem) such as training larger LLMs with the same available resources. For example, with MX6, it enables training a 20% larger model compared to gpt-3-175B. Also, given that tensor slicing degree is largely dictated by underlying

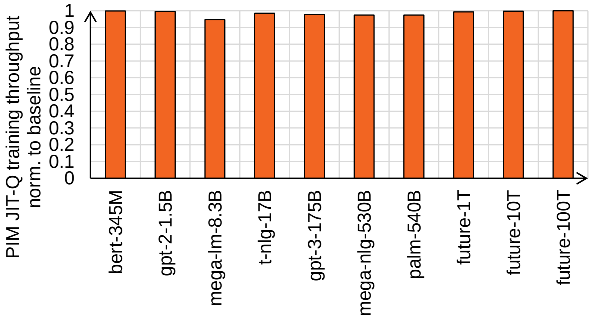

Figure 12. PIM JIT-Q training throughput loss (for BF16 to MX6).

memory capacity, the capacity savings can lower the tensor-slicing degree needed. This can lower the overall training cost, as well as potentially improving computation efficiency because of running larger GEMMs (less tensor slicing) per GPU and reducing model-parallel related communication. For example, the saved capacity enables the use of 12.5% reduction in the tensor-slicing degree. Further, the savings can allow larger batch-sizes to be used which in turn can improve computation efficiency. For example, we estimate an increase in the per-GPU batch-size from two to four for a gpt-3-175B-like model because of the capacity savings JIT-Q provides during training. Finally, with the capacity savings, more activations can be stored which can decrease the frequency of forward phase re-computation.

Finally, we also evaluate a baseline that maintains a single copy of the low-precision weights, either row or column, and observe that JIT-Q delivers capacity savings of 4.8% (up to 6.6%), 6.7% (up to 9.7%), and 9.3% (up to 13.8%) for MX4, MX6, and MX9, respectively, averaged across the evaluated LLMs. Note that such baseline would incur higher data-movement due to reading the high-precision copy to generate the required low-precision copy.

5.5 Effect of PIM JIT-Q on Training Throughput

Assuming the methodology in Section 5.1.3, we estimate the training throughput loss shown in Figure 12. We observe, for BF16 to MX6 quantization that PIM delivers capacity savings at small training throughput loss of 1.6% on average (up to 5% for mega-lm-8.3B). For BF16 to MX4 (not shown), the average throughput loss is 2.4% due to the increased overlap between GPU and PIM. In contrast, the average throughput loss for BF16 to MX9 is 1.1% (not shown). It is worth noting that the reduction in forward phase re-computation enabled by the capacity savings can positively affect training throughput and possibly regain/improve throughput loss due to interference. Further, larger batch-sizes, enabled by the capacity savings due to JIT-Q, can also help recover this loss. Finally, in future PIM designs that offload PIM orchestration to dedicated engines instead of GPU, this compute interference will not pose a challenge.

5.6 PIM JIT-Q with FP32 Master Weights

The single bit shifts to deduce the output MX mantissa bits (*pim-bitSHIFT*) are proportional to the number of mantissa bits in the input scalar format. Therefore, for FP32 master weights, the bit-level shift commands increase, which increases PIM quantization time, and subsequently affects the quantization slack. Specifically, on average, we observe that the GPU compute time is at most 0.51× and 0.76× of the PIM quantization time for MX4 and MX6, respectively, while maintaining the slack for MX9. That said, although PIM is not able to quantize the weights for the next transformer block i+1, PIM JIT-Q can prepare for i+2 instead. In other words, we maintain the quantized weights for the next two transformer blocks instead of one. Compared to keeping the low-precision copies for all the weight tensors in the baseline, JIT-Q delivers capacity savings of 5.6%, 7.9%, and 11.1% at throughput loss of 4.4%, 4.2%, and 3.6% for MX4, MX6, and MX9, respectively, averaged across the evaluated LLMs.

6 Discussion

Master Weights Sharding. Works such as Zero-Redundancy parallelism (ZeRO) (Rajbhandari et al., 2020) and FSDP (Zhao et al., 2023) shard the model states (parameters, gradients, and optimizer state) across GPUs instead of replicating them. The required tensor is reconstructed (communicated) on-demand before computations. Our proposed JIT-Q can be utilized per each (PIM-enabled) GPU to locally quantize its shard of the master weights before communicating the parameters. That said, for directional blocked formats (MX), the partitioning of the master weight tensors should be in tiled fashion (Section 4.2). Finally, JIT-Q can also be used for or in conjunction with recent works that further quantize weights (and gradients) to reduce overall communication volume (Wang et al., 2023).

Master Weights Placement. In scenarios where master weights are offloaded to CPU memory, and only the low-precision copy is sent to/resides in GPU (Ren et al., 2021; Rajbhandari et al., 2021), JIT-Q can be done on PIM-enabled CPU memory to eliminate the persistent low-precision weight copies in GPU memory. In this case, JIT-Q only changes when to send the low-precision weights from CPU to GPU, not the CPU-GPU traffic volume.

Blocked Formats Variants. We discuss MX formats as an exemple of blocked data formats. As such, JIT-Q neither relies on nor prescribes MX formats and is applicable wherever weights are maintained at multiple precisions (e.g., mixed precision training with BF16 and FP32 weight tensors (Micikevicius et al., 2017)). That said, PIM routines for other block sizes with more/fewer exponent levels (Rouhani et al., 2023) and different sub-block granularity can be deduced and employed.

Scalar-to-scalar JIT-Q. Quantization from one scalar format to another (e.g., from FP32 to BF16), used in existing training setups, is simpler compared to MX quantization. This is because each input is independent (no blocking) making the computation element-wise, thus even more PIM-amenable. Also, it does not involve computing the shared exponents, and the bit shift amount is the same across all the inputs. Therefore, we can simplify our PIM quantization routine and mapping to perform such quantization.

Weight Tensor Placement Implications. The proposed tiled data placement (Section 4.2) is key to harnessing PIM bandwidth boost for MX formats and has minor impact on data read/write performance. This is realized by modifying the weight update stage during training. Specifically, while baseline calculates and writes both high and low-precision weight tensors to memory, JIT-Q only requires writing high-precision weight tensors to memory. This simplifies and lowers data-movement in weight update stage. Additionally, these tiled writes of the high-precision weight tensor result in their interleaving across channels/banks in memory. Therefore, the efficiency of the writes is not adversely affected. Furthermore, with such interleaving, the parallelism extracted from memory is preserved on reading the weight tensors. Finally, for ML training employing scalar-to-scalar quantization, JIT-Q has no special placement implications or requirements other than allocating data in large physical pages (Section 2.3).

7 Related Work

Several ML techniques are employed to reduce per-device memory requirements. Distributed techniques help increase the effective memory capacity by slicing model parameters across multiple devices; pipeline parallelism maps layers to devices (Huang et al., 2019; Narayanan et al., 2019), tensor/sequence parallelism slices individual layers (Narayanan et al., 2021; Li et al., 2022), and ZeRO/FSDP shards model states across data-parallel devices (Rajbhandari et al., 2020; Zhao et al., 2023). Other works offload model states to heterogeneous system memory (e.g., CPU, NVMe) (Ren et al., 2021; Rajbhandari et al., 2021). Finally, activation checkpointing trades-off computation for capacity (Chen et al., 2016; Korthikanti et al., 2022). JIT-Q reduces the reliance on such techniques and can also be used in conjunction to limit the required distributed device count, communication overheads, and extraneous compute.

Similarly, compression can provide memory savings (Nikolić et al., 2022; Zadeh et al., 2022) or reduce communication (Wang et al., 2023; Bai et al., 2021; Zhang et al., 2022; Rhu et al., 2018). One class of compression is quantization which is used for both training and inference (Micikevicius et al., 2022; Noune et al., 2022; Darvish Rouhani et al., 2023). While we focus on MX formats, our proposal can be extended to other data formats.

Additionally, ML algorithm-aware techniques such as model pruning (Anwar et al., 2017) and knowledge distillation (Hinton et al., 2015) reduce capacity needs by removing unnecessary model parameters. However, these are employed during deployment and do not help with capacity savings during training. Furthermore, these techniques require algorithmic understanding and/or additional training to preserve model accuracy.

Finally, many works leverage PIM's data-movement and performance benefits to accelerate ML (Lee et al., 2021; Pati et al., 2022; Oliveira et al., 2022; Aga et al., 2019). To the best of our knowledge, this is the first work which showcases PIM's ability to limit capacity requirements via efficient JIT quantization.

8 Conclusion

Memory capacity is a key determinator of ML efficiency. This work proposes harnessing processing-in-memory (PIM) for just-in-time quantization of weight tensors. This eliminates weight tensor redundancy to deliver memory capacity savings. We show that our proposed PIM quantization routine can keep up with GPU computation and deliver up to 24% memory capacity savings at marginal throughput loss (up to 2.4%) for LLM training. Resultant capacity savings can be harnessed to train larger models, to reduce the number of GPUs required for training, or for better compute efficiency.

Acknowledgements

The authors thank Nuwan Jayasena and the anonymous MLSys reviewers for helping improve the paper. AMD, the AMD Arrow logo, AMD Instinct, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

References

Aga, S., Jayasena, N., and Ignatowski, M. Co-ML: A Case for Collaborative ML Acceleration Using near-Data Processing. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*, 2019.

Alayrac, J.-B., Donahue, J., Luc, P., Miech, A., Barr, I., Hasson, Y., Lenc, K., Mensch, A., Millican, K., Reynolds, M., et al. Flamingo: A Visual Language Model for Few-shot Learning. *Advances in Neural Information Processing Systems*, 2022.

AMD. AMD Instinct™ MI210 Accelerator. https://www.amd.com/en/products/server-accelerators/amd-instinct-mi210, 2023a.

AMD. Omniperf. https://github.com/AMDResearch/omniperf, 2023b.

Anwar, S., Hwang, K., and Sung, W. Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 2017.

Avalos Baddouh, C., Khairy, M., Green, R. N., Payer, M., and Rogers, T. G. Principal Kernel Analysis: A Tractable Methodology to Simulate Scaled GPU Workloads. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021.

Bai, Y., Li, C., Zhou, Q., Yi, J., Gong, P., Yan, F., Chen, R., and Xu, Y. Gradient Compression Supercharged High-Performance Data Parallel DNN Training. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, 2020.

Chen, T., Xu, B., Zhang, C., and Guestrin, C. Training Deep Nets with Sublinear Memory Cost. *arXiv*, 2016.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N., Prabhakaran, V., Reif, E., Du, N., Hutchinson, B., Pope, R., Bradbury, J., Austin, J., Isard, M., Gur-Ari, G., Yin, P., Duke, T., Levskaya, A., Ghemawat, S., Dev, S., Michalewski, H., Garcia, X., Misra, V., Robinson, K., Fedus, L., Zhou, D., Ippolito, D., Luan, D., Lim, H., Zoph, B., Spiridonov, A., Sepassi, R., Dohan, D., Agrawal, S., Omernick, M., Dai, A. M., Pillai, T. S., Pellat, M., Lewkowycz, A., Moreira, E., Child, R., Polozov, O., Lee, K., Zhou, Z., Wang, X., Saeta, B., Diaz, M., Firat, O., Catasta, M., Wei, J., Meier-Hellstern, K., Eck, D., Dean, J., Petrov, S., and Fiedel, N. PaLM: Scaling Language Modeling with Pathways. *arXiv*, 2022.

Darvish Rouhani, B., Zhao, R., Elango, V., Shafipour, R., Hall, M., Mesmakhosroshahi, M., More, A., Melnick, L., Golub, M., Varatkar, G., Shao, L., Kolhe, G., Melts, D., Klar, J., L'Heureux, R., Perry, M., Burger, D., Chung, E., Deng, Z. S., Naghshineh, S., Park, J., and Naumov, M. With Shared Microexponents, A Little Shifting Goes a Long Way. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2023.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv*, 2019.

Graphcore. Mixed-Precision Arithmetic for AI: A Hardware Perspective. https://docs.graphcore.ai/projects/ai-float-white-paper/en/latest/ai-float.html, May 2023.

He, M., Song, C., Kim, I., Jeong, C., Kim, S., Park, I., Thottethodi, M., and Vijaykumar, T. Newton: A DRAM-maker's accelerator-in-memory (AiM) architecture for machine learning. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

Hinton, G., Vinyals, O., and Dean, J. Distilling the knowledge in a neural network. *arXiv*, 2015.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, M. X., Chen, D., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., and Chen, Z. *GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism*. 2019.

JEDEC. High Bandwidth Memory (HBM) DRAM. https://www.jedec.org/standards-documents/docs/jesd235a, 2013.

JEDEC. High Bandwidth Memory (HBM3) DRAM. https://www.jedec.org/standards-documents/docs/jesd238a, 2023.

Kalamkar, D. D., Mudigere, D., Mellempudi, N., Das, D., Banerjee, K., Avancha, S., Vooturi, D. T., Jammalamadaka, N., Huang, J., Yuen, H., Yang, J., Park, J., Heinecke, A., Georganas, E., Srinivasan, S., Kundu, A., Smelyanskiy, M., Kaul, B., and Dubey, P. A Study of BFLOAT16 for Deep Learning Training. *arXiv*, 2019.

Kim, J. H., Kang, S.-H., Lee, S., Kim, H., Ro, Y., Lee, S., Wang, D., Choi, J., So, J., Cho, Y., et al. Aquabolt-XL HBM2-PIM, LPDDR5-PIM with in-memory processing, and AXDIMM with acceleration buffer. *IEEE Micro*, 2022.

Korthikanti, V., Casper, J., Lym, S., McAfee, L., Andersch, M., Shoeybi, M., and Catanzaro, B. Reducing Activation Recomputation in Large Transformer Models. *arXiv*, 2022.

Lee, S., Kang, S.-h., Lee, J., Kim, H., Lee, E., Seo, S., Yoon, H., Lee, S., Lim, K., Shin, H., Kim, J., Seongil, O., Iyer, A., Wang, D., Sohn, K., and Kim, N. S. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology: Industrial Product. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2021.

Lee, S., Kim, K., Oh, S., Park, J., Hong, G., Ka, D., Hwang, K., Park, J., Kang, K., Kim, J., Jeon, J., Kim, N., Kwon, Y., Vladimir, K., Shin, W., Won, J., Lee, M., Joo, H., Choi, H., Lee, J., Ko, D., Jun, Y., Cho, K., Kim, I., Song, C., Jeong, C., Kwon, D., Jang, J., Park, I., Chun, J., and Cho, J. 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 *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, 2022.

Lee, W. J., Kim, C. H., Paik, Y., Park, J., Park, I., and Kim, S. W. Design of processing-“inside”-memory optimized for dram behaviors. *IEEE Access*, 2019.

Li, S., Xue, F., Baranwal, C., Li, Y., and You, Y. Sequence Parallelism: Long Sequence Training from System Perspective. *arXiv*, 2022.

Mellempudi, N., Srinivasan, S., Das, D., and Kaul, B. Mixed Precision Training With 8-bit Floating Point. *arXiv*, 2019.

Micikevicius, P., Narang, S., Alben, J., Diamos, G. F., Elsen, E., García, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., and Wu, H. Mixed Precision Training. *arXiv*, 2017.

Micikevicius, P., Stosic, D., Burgess, N., Cornea, M., Dubey, P., Grisenthwaite, R., Ha, S., Heinecke, A., Judd, P., Kamalu, J., Mellempudi, N., Oberman, S., Shoeybi, M., Siu, M., and Wu, H. FP8 Formats for Deep Learning. *arXiv*, 2022.

Microsoft. Turing-NLG: A 17-billion-parameter language model by Microsoft. https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/, Feb 2020.

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V. A., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., and Zaharia, M. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. *arXiv*, 2021.

Nikolić, M., Sanchez, E. T., Wang, J., Zadeh, A. H., Mahmoud, M., Abdelhadi, A., and Moshovos, A. Schrödinger’s FP: Dynamic Adaptation of Floating-Point Containers for Deep Learning Training. *arXiv*, 2022.

Noune, B., Jones, P., Justus, D., Masters, D., and Luschi, C. 8-bit Numerical Formats for Deep Neural Networks. *arXiv*, 2022.

Oliveira, G. F., Gómez-Luna, J., Ghose, S., Boroumand, A., and Mutlu, O. Accelerating Neural Network Inference With Processing-in-DRAM: From the Edge to the Cloud. *IEEE Micro*, 2022.

OpenAI. Language Models are Unsupervised Multitask Learners. `https://openai.com/research/better-language-models`, Feb 2019.

Pati, S., Aga, S., Jayasena, N., and Sinclair, M. D. Demystifying BERT: System Design Implications. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2022.

Pati, S., Aga, S., Islam, M., Jayasena, N., and Sinclair, M. D. Tale of Two Cs: Computation vs. Communication Scaling for Future Transformers on Future Hardware. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2023.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. ZeRO: Memory Optimizations toward Training Trillion Parameter Models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

Rajbhandari, S., Ruwase, O., Rasley, J., Smith, S., and He, Y. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021.

Ren, J., Rajbhandari, S., Aminabadi, R. Y., Ruwase, O., Yang, S., Zhang, M., Li, D., and He, Y. ZeRO-Offload: Democratizing Billion-Scale Model Training. *arXiv*, 2021.

Rhu, M., O’Connor, M., Chatterjee, N., Pool, J., Kwon, Y., and Keckler, S. W. Compressing DMA Engine: Leveraging Activation Sparsity for Training Deep Neural Networks. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

Rouhani, B. D., Zhao, R., More, A., Hall, M., Khodamoradi, A., Deng, S., Choudhary, D., Cornea, M., Dellinger, E., Denolf, K., Dusan, S., Elango, V., Golub, M., Heinecke, A., James-Roxby, P., Jani, D., Kolhe, G., Langhammer, M., Li, A., Melnick, L., Mesmakhosroshahi, M., Rodriguez, A., Schulte, M., Shafipour, R., Shao, L., Siu, M., Dubey, P., Micikevicius, P., Naumov, M., Verilli, C., Wittig, R., and Chung, E. Microscaling Data Formats for Deep Learning. *arXiv*, 2023.

Samsung. Samsung Electronics Semiconductor Unveils Cutting-edge Memory Technology to Accelerate Next-generation AI. https://semiconductor.samsung.com/newsroom/tech-blog/samsung-electronics-semiconductor-unveils-cutting-edge-memory-technology-to-accelerate-next-generation-ai/, 2022.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv*, 2020.

Smith, S., Patwary, M., Norick, B., LeGresley, P., Rajbhandari, S., Casper, J., Liu, Z., Prabhumoye, S., Zerveas, G., Korthikanti, V., Zhang, E., Child, R., Aminabadi, R. Y., Bernauer, J., Song, X., Shoeybi, M., He, Y., Houston, M., Tiwary, S., and Catanzaro, B. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model. *arXiv*, 2022.

Sung, Y.-L., Cho, J., and Bansal, M. Vl-adapter: Parameter-efficient Transfer Learning for Vision-and-Language Tasks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022.

Tsimpoukelli, M., Menick, J. L., Cabi, S., Eslami, S., Vinyals, O., and Hill, F. Multimodal Few-shot Learning with Frozen Language Models. *Advances in Neural Information Processing Systems*, 2021.

Untether AI. Untether AI. https://www.untether.ai/technology, 2023.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention Is All You Need. In *Proceedings of the International Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

Villa, O., Lustig, D., Yan, Z., Bolotin, E., Fu, Y., Chatterjee, N., Jiang, N., and Nellans, D. Need for Speed: Experiences Building a Trustworthy System-Level GPU Simulator. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

Wang, G., Qin, H., Jacobs, S. A., Holmes, C., Rajbhandari, S., Ruwase, O., Yan, F., Yang, L., and He, Y. ZeRO++: Extremely Efficient Collective Communication for Giant Model Training. *arXiv*, 2023.

Wang, N., Choi, J., Brand, D., Chen, C.-Y., and Gopalakrishnan, K. Training Deep Neural Networks with 8-bit Floating Point Numbers. *arXiv*, 2018.

Zadeh, A. H., Mahmoud, M., Abdelhadi, A., and Moshovos, A. Mokey: Enabling Narrow Fixed-Point Inference for out-of-the-Box Floating-Point Transformer Models. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2022.

Zhang, Z., Zheng, S., Wang, Y., Chiu, J., Karypis, G., Chilimbi, T., Li, M., and Jin, X. MiCS: Near-Linear Scaling for Training Gigantic Model on Public Cloud. *Proceedings of the VLDB Endowment*, 2022.

Zhao, Y., Gu, A., Varma, R., Luo, L., Huang, C.-C., Xu, M., Wright, L., Shojanazeri, H., Ott, M., Shleifer, S., Desmaison, A., Balioglu, C., Damania, P., Nguyen, B., Chauhan, G., Hao, Y., Mathews, A., and Li, S. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *arXiv*, 2023.