

MX+: Pushing the Limits of Microscaling Formats for Efficient Large Language Model Serving

Jungi Lee
 Seoul National University
 Seoul, Republic of Korea
 jungi.lee@snu.ac.kr

Junyong Park
 Seoul National University
 Seoul, Republic of Korea
 junyong.park@snu.ac.kr

Soohyun Cha
 Seoul National University
 Seoul, Republic of Korea
 soohyun.cha@snu.ac.kr

Jaehoon Cho
 Seoul National University
 Seoul, Republic of Korea
 jaehoon.cho@snu.ac.kr

Jaewoong Sim
 Seoul National University
 Seoul, Republic of Korea
 jaewoong@snu.ac.kr

Abstract

Reduced-precision data formats are crucial for cost-effective serving of large language models (LLMs). While numerous reduced-precision formats have been introduced thus far, they often require intrusive modifications to the software frameworks or are rather unconventional for widespread adoption across hardware vendors. In this paper, we instead focus on recent *industry-driven* variants of block floating-point (BFP) formats and conduct a comprehensive analysis to push their limits for efficient LLM serving. Our analysis shows that existing ultra low-bit BFP variants struggle to provide reasonable language model performance due to outlier values in blocks. To address the outliers with BFPs, we propose MX+, a cost-effective and non-intrusive extension designed for seamless integration into the microscaling (MX) formats. MX+ builds on the key insight that the outlier does not need to use its exponent field in the element data type, which allows us to *repurpose* the exponent field as an extended mantissa to increase the precision of the outlier element. Our evaluation shows that MX+ achieves significantly higher model performance compared to the 4-bit MX format (MXFP4) with negligible storage overhead and slowdown, thus offering a compelling alternative to MXFP4 or MXFP6 for efficient LLM inference.

CCS Concepts

- Computer systems organization → Neural networks; Single instruction, multiple data;
- Hardware → Arithmetic and datapath circuits.

Keywords

Large Language Models, Microscaling Formats, GPUs

ACM Reference Format:

Jungi Lee, Junyong Park, Soohyun Cha, Jaehoon Cho, and Jaewoong Sim. 2025. MX+: Pushing the Limits of Microscaling Formats for Efficient Large Language Model Serving. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea*. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3725843.3756118>



This work is licensed under a Creative Commons Attribution 4.0 International License.
MICRO '25, Seoul, Republic of Korea
 © 2025 Copyright held by the owner/author(s).
 ACM ISBN 979-8-4007-1573-0/25/10
<https://doi.org/10.1145/3725843.3756118>

1 Introduction

Emerging services that leverage large language models (LLMs)—such as human-like chatbots and programming assistants—are increasingly impacting our daily lives, making LLMs crucial workloads for both software and hardware vendors today. Efficiently serving LLMs, however, presents significant challenges due to their substantial demand for compute and memory resources. In recent years, numerous low-bit quantization schemes and reduced-precision data formats have been introduced to alleviate compute and memory overheads. Yet, they typically require non-negligible changes to the software codebase for additional operations—such as per-channel scaling [63] and channel grouping [32]—or diverge too far from conventional integer or floating-point representations to be broadly adopted across computing platforms [15, 16].

To address these challenges, industry-leading companies, including AMD, Arm, Intel, Meta, Microsoft, NVIDIA, and Qualcomm, have recently collaborated to standardize an open and interoperable family of data formats and introduced Microscaling (MX) data formats [47], which build upon block floating-point (BFP) representations. However, there has been limited study on their effectiveness in serving LLMs or on comparisons with other BFP variants from the industry, such as Microsoft Floating Point (MSFP) [6] or shared microexponents [7].

In this paper, we begin by investigating *industry-driven* BFP variants and understanding their implications for low-bit LLM inference. Our analysis shows that MX achieves better language model performance than other BFP variants that employ similar bit widths. However, employing the 4-bit MX format (i.e., MXFP4) for *both* weights and activations results in a substantial reduction in model performance, limiting the maximum benefits of the MX formats. We observe that this is due to a small number of values whose magnitudes are significantly larger than others in LLM activations (called *outliers*), which undergo large quantization errors when converted into MXFP4.

To effectively address the outliers in BFPs, we propose the MX+ format, a cost-effective and non-intrusive extension to MX, which enables LLM serving with *both* 4-bit weights and activations. The MX+ design builds on two key insights. First, in MX formats, the exponent of the largest magnitude value in a block is used for determining a *shared* scale, allowing the natural identification of an outlier element and its position within the block *without* additional computation or extra hardware logic. Second, for the outlier element

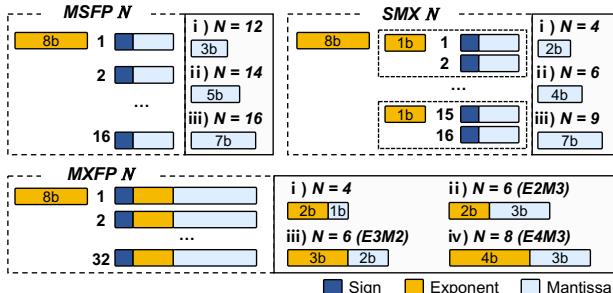


Figure 1: Industry-driven block floating-point formats.

in an MX block, we do not need to store its *own* exponent as it is *always* set to the maximum representable exponent of the element data type. This allows us to *repurpose* the exponent field to store more mantissa bits for the outlier element, which greatly helps increase the precision of the outlier in low-bit MX formats, where only one or a few mantissa bits exist (e.g., MXFP4).

The non-intrusive design of MX+ makes it easy to integrate into LLM frameworks and computing systems through both software and hardware approaches. On the software side, we implement MX+ and evaluate its accuracy and performance across various LLMs using the MX emulation library [39], NVIDIA CUTLASS [56], and the Triton compiler [57]. Additionally, we propose an approach to reduce the performance overhead of software-based MX+ integration by incorporating architectural support into acceleration units such as Tensor Cores in NVIDIA GPUs. Our evaluation shows that MX+ offers a significant improvement in model performance across a range of LLMs, achieving up to a +42.15% improvement for the 4-bit MX format, with a negligible slowdown under software integration or architectural support.

In summary, this paper makes the following contributions:

- We propose MX+, a non-intrusive extension to the MX formats. MX+ offers improved representation of outliers without additional user effort or complexity and achieves high model performance even with low-bit quantization.
- We integrate MX+ into existing software libraries and demonstrate that it incurs a marginal slowdown in LLM inference, *without* hardware modifications, particularly when token generation dominates inference time.
- We present a hardware design that enables direct MX+ computation within Tensor Cores without making intrusive modifications to the dot product pipeline, delivering near-MX performance while achieving higher model accuracy.

2 Preliminaries: Industry-Driven Block-Based Data Formats

By mapping weights and activations from higher precision to lower precision with coarser-grained representations, one can accelerate computation with simpler yet higher-throughput compute units while also achieving a more efficient use of memory bandwidth and capacity. For instance, the widely used uniform, symmetric integer quantization scheme maps a group of k floating-point numbers x_f to b -bit integers x_q using the scale factor s , as shown below:

$$s = \frac{\max(|x_f|)}{2^{b-1} - 1}; \quad x_q = \text{round}\left(\frac{x_f}{s}\right).$$

Table 1: Concrete MX-compliant formats.

Name	Element Data Type	Bits	Block Size (k)	Scale Format
MXFP8	E5M2	8	32	E8M0
	E4M3	8	32	E8M0
MXFP6	E3M2	6	32	E8M0
	E2M3	6	32	E8M0
MXFP4	E2M1	4	32	E8M0
MXINT8	INT8	8	32	E8M0

For integer quantization, dequantization involves multiplying the scale factor with the integer values, $s \cdot x_q$.

Block floating-point (BFP) formats, such as MSFP [6], bear some similarity to conventional integer quantization, but with the scale factor (s) *restricted* to powers of two. This restriction enables hardware to efficiently manage scaling or rescaling at a finer block (k) granularity, thereby allowing for more accurate representations of the original weight and activation tensors. Figure 1 provides a comparison of several industry-proposed block-based data formats, which we briefly explain below.

Microsoft Floating Point. Microsoft Floating Point (MSFP) is a variant of BFP formats, which was deployed in Project Brainwave [14]. An MSFP block comprises k number of elements, each with its own sign bit and mantissa bits, and a *shared* exponent used by all elements in the block. In a typical use case of MSFP [6], for instance, 16 elements in a floating-point tensor are grouped into a block with an 8-bit *shared* exponent, which is set to the exponent of the largest absolute value within the block. The mantissa of each element is obtained by right-shifting the original floating-point value by the difference between the shared exponent and its original exponent; therefore, there are no implicit leading bits in the MSFP mantissa. Note that MSFP formats are named based on their total bit width; for instance, MSFP12 has only four bits for the sign and mantissa, resulting in an average bit width of 4.5 bits per element.

Shared Microexponents. Shared microexponents (SMX) data formats [7] are a recent proposal similar to MSFP in that scale factors are shared by a group of elements and restricted to powers of two.¹ However, SMX employs a *multi-level scaling* approach, in contrast to the single-level scaling in MSFP. In its typical use case of *two-level scaling* [7], a group of 16 elements ($k_1=16$) shares a *first-level* scale factor s , which is an 8-bit shared exponent, while pairs of elements ($k_2=2$) *within* the group form a subgroup that shares a *second-level* scale (sub-scale) factor ss , represented by a one-bit shared microexponent for each subgroup.

Microscaling Formats. Microscaling (MX) formats [47] are another recent proposal developed in collaboration with multiple industry companies, with the objective of establishing open and interoperable data formats. An MX block consists of 32 elements ($k=32$) with a shared scale X , which is an 8-bit shared exponent, similar to the one used in MSFP or SMX. However, unlike MSFP or SMX, the element data type can be selected from five floating-point and one integer encodings, as shown in Table 1.

The integer data type (i.e., MXINT8) uses a two's complement encoding and has an implicit scale factor of 2^{-6} . For MXFP formats, each private element in an MX block has its *own* exponent bits,

¹We denote the shared microexponents data formats as SMX in this paper to distinguish it from the OCP proposal of Microscaling (MX) formats.

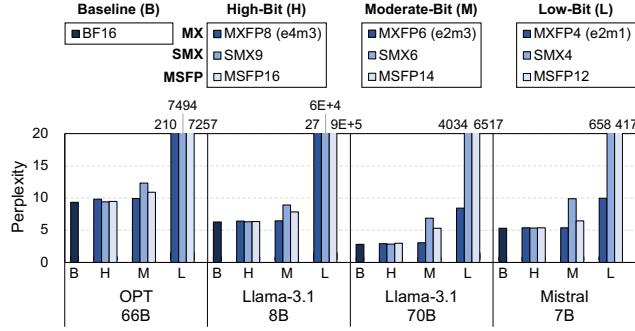


Figure 2: Perplexity with BF16 baseline (B), MSFP, SMX, and MX formats. We compare each format with varying bit widths: high (H), moderate (M), and low (L).

which makes each element effectively a floating-point number. In MX, the shared exponent and the corresponding scale factor X can be computed as follows:

$$\begin{aligned} \text{shared_exp} &= \max(\lfloor \log_2(|x_f|) \rfloor) - e_{\max}, \\ X &= 2^{\text{shared_exp}}, \end{aligned} \quad (1)$$

where e_{\max} is the maximum representable exponent value of the element data type. For instance, in MXFP4, the element data type is FP4, and each element has an individual 2-bit exponent with the exponent bias of 1. Thus, e_{\max} becomes 2 (i.e., $11_2 - 1$).

3 Serving LLMs with Block-Based Data Formats

In this section, we first compare the model performance of various LLMs when using the BFP variants discussed in Section 2. We then investigate the root cause of performance degradations with extremely low-bit formats and discuss how to better exploit low-bit BFP for LLM serving.

3.1 Model Performance with BFP Variants

Figure 2 shows the perplexity of various LLMs across different industry-driven BFP formats using the WikiText-2 dataset with the sequence length of 2048.² The baseline (B) uses Bfloat16 (BF16) as the default data format and performs matrix multiplications and element-wise operations in BF16, except for softmax, which uses FP32. For the evaluation with BFP variants, we follow the computation flow outlined in prior work [7, 52]; BF16 tensors are converted into MSFP, SMX, and MX for matrix multiplications, while element-wise operations use the same precision as the baseline (i.e., BF16 or FP32). We select the MSFP and SMX formats with average bits per element similar to those in MXFP4 (L), MXFP6 (M), and MXFP8 (H). The average bit widths of these BFP variants fall within the ranges of $4 \leq L \leq 4.5$, $6 \leq M \leq 6.5$, and $8.25 \leq H \leq 9$.

In general, MX outperforms or matches other BFP variants with similar bit widths. For the high-bit (H) formats, all BFP variants perform close to the baseline; while MXFP8 shows slightly higher perplexity than SMX9 or MSFP16, this is due to its lower average bits per element (8.25, compared to 9 and 8.5 in the other formats) and the use of reserved NaN representations, which are not supported by SMX or MSFP. In the moderate-bit (M) formats, however, SMX

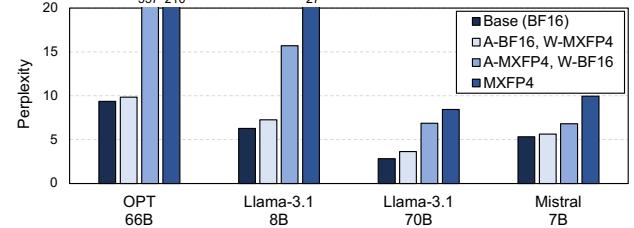


Figure 3: Perplexity of WikiText-2 across a mix of BF16 and MXFP4.

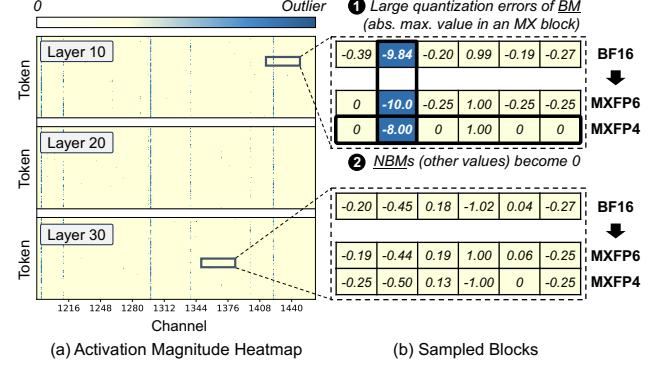


Figure 4: (a) Activation Magnitude Heatmap of the sample attention input tensors of Llama-3.1-8B. (b) BF16 values and their MXFP4 and MXFP6 representations. BF16 values are rounded to the second decimal place for brevity.

and MSFP14 begin to diverge, making them less effective for LLM serving scenarios, while MXFP6 remains close to the baseline. This is because, unlike SMX or MSFP, where exponents are shared among some or all elements in a block, each element in MXFP has its own exponent in addition to the *shared* one, allowing for more fine-grained value representations. In addition, MXFP employs an *implicit* leading 1 for normals, with sub-normals defined similarly to IEEE-754 floating-point formats, resulting in a larger effective bit width compared to other formats. However, when using the low-bit MX format (i.e., MXFP4), perplexity also begins to deviate from the baseline, even with MX. While it still significantly outperforms SMX4 and MSFP12, this makes the low-bit MX format less suitable for practical use in LLM serving, despite its potential for substantial bandwidth savings and computational efficiency.

3.2 Analysis on Low-Bit MX Format

To understand the underlying reasons for the reduction in model performance with the low-bit MX format, we conduct a further analysis on MXFP4. We first evaluate the perplexity when only either activation tensors (A) or weight tensors (W) are quantized to MXFP4, while the others use BF16. Figure 3 shows that quantizing weights (A-BF16, W-MXFP4) leads to a negligible increase in perplexity, whereas quantizing activations (A-MXFP4, W-BF16) substantially degrades model performance. This indicates that although MX employs fine-grained scaling (i.e., 32 elements per block) to reduce the impact of outlier values in tensors, low-bit MX does not effectively mitigate this for activation tensors.

To gain deeper insight into the root cause, we examine the MX blocks in the activation tensors. Figure 4(a) shows a heatmap of activation magnitudes from Llama-3.1-8B, while Figure 4(b) presents

²Perplexity is a widely used metric to assess the model performance of generative LLMs; lower values indicate better performance.

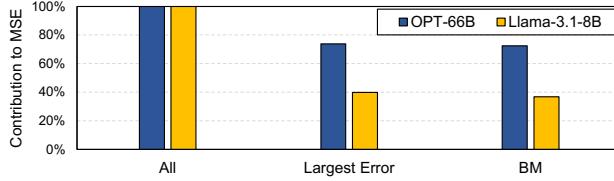


Figure 5: Contribution to MSE (%) from elements with the largest error in each MX block or from BM elements. We use the sampled attention input tensor of Layer 16.

two sample blocks with the original BF16 values and their MXFP4 and MXFP6 representations. Here, we refer to the *absolute maximum value* in an MX block as *Block Max (BM)*, while the other values are referred to as *Non-Block Max (NBM)*.

We observe that blocks in which the BM is significantly larger than the NBMs (e.g., the upper sampled block)—primarily due to the presence of an outlier in the MX block—tend to exhibit high quantization errors for two reasons. First, since MXFP4 allocates only 1 bit for the mantissa, large-magnitude BMs are susceptible to substantial deviation from their original values upon quantization (❶). Second, the exponent of a BM dictates a block-wide scale factor, as shown in Equation 1. As a result, when the BM is large, the shared scale factor also becomes large, which forces the other smaller elements (i.e., NBMs) to be represented less precisely due to the shared scale. For instance, most NBMs are quantized to zero after being divided by the shared scale factor (❷). In contrast, blocks without outliers (e.g., the lower sampled block) naturally have relatively low quantization errors.

Figure 5 shows the contribution to MSE (Mean Squared Error) from elements with the largest quantization error or from BM elements. We can see that more accurately representing the BM element in every MX block can reduce a significant portion of the quantization error. While theoretically any of the 32 elements in an MX block could have the largest error for the block, identifying the largest error element in every block introduces computational complexity without adding much benefit, as the BM element is often the largest contributor.

In summary, while it would be ideal to precisely represent all 32 elements in a block, this is not likely feasible with MXFP4 due to the limited number of mantissa bits. Instead, our analysis shows that focusing on better representation of the BM element alone can noticeably help improve model performance when using the low-bit MX format.

4 MX+: Enhancing the MX Formats

As discussed in Section 3.2, the largest magnitude in an MX block often experiences the highest quantization error among the elements within the block, which significantly hurts model performance particularly when the MX block contains an outlier. In this section, we propose MX+, a cost-effective extension designed for seamless integration into the MX formats for low-bit LLM serving.

4.1 MX+ Design

Our MX+ design revolves around three key considerations. First, it should not interfere with the design goals of MX; the extension needs to be managed *solely* within conversion kernels or hardware

Binary Encoding									
BF16									
-0.39								-0.20	0.99
-9.84								-0.19	-0.27
MXFP4									
0								1.00	0
-8.00								0	0
MXFP4+									
0								1.00	0
-10.0								0	0
0000								1110	0000
0000								0007	0000
0000								0000	0000
0000								0000	0000
0000								0000	0000
0000								0000	0000
Shared Scale									
2^1									

Figure 6: Comparison of MX and MX+ encodings (S: sign, E: exponent, M: mantissa). The BM element is boxed in blue. BF16 values are rounded to the second decimal place for brevity.

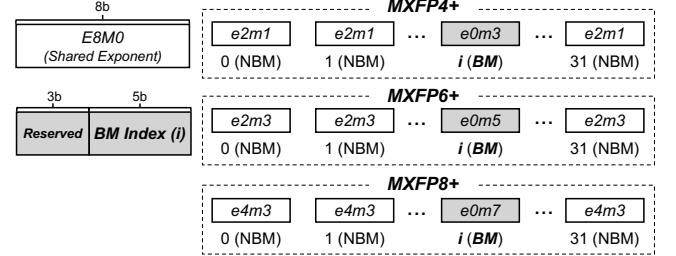


Figure 7: Data layout of MX+. We extend the mantissa bits instead of storing exponent bits in BM. An additional 8-bit is assigned per block to store the index of BM.

units for seamless integration with various frameworks, without requiring additional effort from end-users. Second, the extension needs to effectively handle outliers in blocks to mitigate quantization errors while aligning with the MX specification [47]. Third, the extension should not introduce large overheads in terms of storage and runtime latency.

MX+ builds on two key insights. First, the BM element in each MX block does not need to store its *own* exponent, as it is *always* set to the maximum representable exponent of the element data type, provided extremely small magnitude numbers below a threshold are flushed to zero. This allows us to safely *repurpose* the exponent field as an *extended mantissa* to more precisely represent the BM. Second, the BM element is also *naturally* identified during conversion from higher precision to MX for computing the shared scale. Thus, no additional computation is required to identify the BM element in each MX block.

Figure 6 illustrates an example of binary encoding for MXFP4 and MXFP4+ using the sampled block presented in Figure 4. As shown in the figure, the exponent field of the BM element is always set to the maximum representable value (i.e., 11_2) in MXFP4. MXFP4+ repurposes these bits to store additional mantissa, thereby providing a more accurate representation of the original BM value. Note that MX+ does not alter the shared scale.

Similar to prior work [52], we flush values with extremely small magnitudes to zero to simplify conversion and enable the MX+ extension. Specifically, if the exponent of the BM ($\lfloor \log_2(\text{BM}) \rfloor$) is less than or equal to $-127 + e_{\max}$, we set all elements in the block to zero. This is because, in such cases, the shared exponent gets clamped at its lower bound of -127, which results in the exponent field of the element data type being set less than e_{\max} . To represent this case, we extend the shared exponent encoding by reserving a special value: a biased shared exponent of zero indicates that all elements in the block are zero.

4.2 Data Layout of MX+

Figure 7 shows the data layout of MX+ for three possible types: MXFP4+, MXFP6+, and MXFP8+, each of which is an extension to MXFP4, MXFP6 (E2M3), and MXFP8 (E4M3), respectively. An additional 8 bits are assigned to each MX block, where five bits are used to store the index of the BM element within the block. The remaining three bits are reserved, which can be utilized for further optimizations or to support future MX specifications that define formats with block sizes other than 32 elements.

NBM values are converted to conventional MX element data types such as E2M1, E2M3, and E4M3, while the BM value is stored with more mantissa bits such as E0M3, E0M5, and E0M7. We do not explicitly store the exponent of BM since it will always be the maximum of the given element data type (i.e., 2 for E2M1 and E2M3; 8 for E4M3), as shown in Equation 1. Thus, while using the same bit width as NBMs, BMs are effectively represented as E2M3, E2M5, and E4M7 for MXFP4+, MXFP6+, and MXFP8+.

Note that since all elements use the same bit width, MX+ does not lead to unaligned memory access. This design also incurs a negligible overhead in terms of compute and memory costs, as BMs are already identified during conversion to the MX format. The additional bits increase the average bit width by only 0.25 (e.g., from 4.25 to 4.5 for MXFP4). Similar to the shared scale [47], the index metadata does not need to be stored contiguously with the element data or the shared scale. It can also be compressed or pruned away for the repeated values.

4.3 Potential Use of Reserved Bits

While MX+ greatly reduces block-wise quantization error by representing the BM element more precisely, NBM elements also contribute to quantization error. As discussed earlier, since the shared scale is determined by the BM, NBMs may be represented even less precisely than they would be with their own scales. To show an example of exploiting the reserved bits, we also consider a variant of MX+, referred to as MX++, and evaluate its accuracy.

MX++ decouples the shared scale of NBMs from that of the BM by utilizing the reserved bits, often enabling NBMs to be mapped to a finer quantization grid compared to MX+. Specifically, NBMs employ a shared scale that is smaller than or equal to the shared scale for the BM, with the difference between their shared exponents encoded into the reserved three bits in Figure 7. However, directly applying the shared exponent computation from Equation 1 to NBMs may increase quantization error because NBM elements could saturate to the maximum magnitude of the element data type after scaling. We thus define the smallest feasible shared exponent e for NBMs to avoid saturation as follows:

$$e = \max_2(\lfloor \log_2(|x_f|) \rfloor) - e_{\max} + 1,$$

where \max_2 identifies the second-largest exponent in a given MX block. Without the offset of 1, the first two terms would represent the shared exponent of an MX block without the BM (Equation 1), which may introduce additional error. Consider the example elements in Figure 6. Without the offset, e equals -3 ($= -1 - 2$, where -1 is the exponent of 0.99), and the value 0.99 is scaled to 7.92 ($= 0.99 \div 2^{-3}$) and saturated to the maximum representable value of 6.0 in MXFP4. With the offset of 1, however, the value is scaled to

3.96 and remains within the representable range. The final shared exponent for NBMs is then determined by applying the clipping function $CLIP(x, \{min, max\})$:

$$\text{shared_exp}_{\text{new}} = CLIP(e, \{\text{shared_exp} - 7, \text{shared_exp}\}),$$

where shared_exp and $\text{shared_exp}_{\text{new}}$ denote the shared exponents for the BM and NBMs in MX++. The lower bound ensures that the difference from the BM's shared exponent (shared_exp) fits within 3 bits. The upper bound addresses the case where the exponents of the BM and the largest NBM are identical and thus e exceeds shared_exp due to the offset. Revisiting the previous example, $\text{shared_exp}_{\text{new}}$ of -2 enables the NBM value -0.39 to scale to -1.56 and map to -1.5, whereas it was previously quantized to zero with the shared_exp of 1.

5 Software Integration of MX+ on GPUs

MX formats are increasingly integrated into existing DNN acceleration systems with software and hardware support. In systems lacking compute units for low-precision element data types in MX, MX blocks are typically converted to a higher-precision format supported by the hardware [21, 57, 61]. For example, data stored in MXFP4 can be converted to FP16 for computation on Intel Granite Rapids via software support [21]. In such scenarios, MX+ can also be easily supported with a minor modification to the conversion kernel as follows:

$$\begin{aligned} \text{output}_i &= (-1)^{s_i} \times 2^{\text{shared_exp}} \times E_i & 0 \leq i < 32, \\ \text{where } E_i &= \begin{cases} 2^{e_{\max}} \times m_i & \text{if } i = \text{BM Index, (MX+ Addition)} \\ 2^{e_i - b} \times m_i & \text{otherwise,} \end{cases} \end{aligned} \quad (2)$$

where i and b denote the element index within a block and the exponent bias of the given element data type. Also, s_i , m_i , and e_i represent the sign, mantissa, and private exponent of an input element i , respectively. Note that the mantissa m_i is captured differently between BM and NBM elements in MX+. For instance, in MXFP4+, the BM element has three effective bits for the mantissa, while the NBM elements have only one bit.

When the systems are equipped with compute units that natively support MX-compliant formats—such as Tensor Cores in the recently announced NVIDIA Blackwell GPUs [41]—element data in MX blocks can be processed directly within the compute unit *without* the need for format conversion. In this section, we present an approach to integrate the MX+ extension into a GPU system that supports MX precision formats *without* requiring any hardware modifications. For clarity, we focus on the scenario where activations are represented in MXFP4+ and weights in MXFP4, a configuration that achieves model performance comparable to the case where both are in MXFP4+, as we discuss in Section 7.2. However, the proposed approach can also be applied when both operands use MXFP4+ or other formats such as MXFP6+.

5.1 Challenges of Handling BM on GPUs

NVIDIA's Tensor Core performs matrix multiply-and-accumulate (MMA) operations ($D = A \times B + C$) for a *predefined* set of matrix tile shapes ($M \times N \times K$) with specific input and output data types. Tensor Core MMA operations are exposed to programmers through PTX

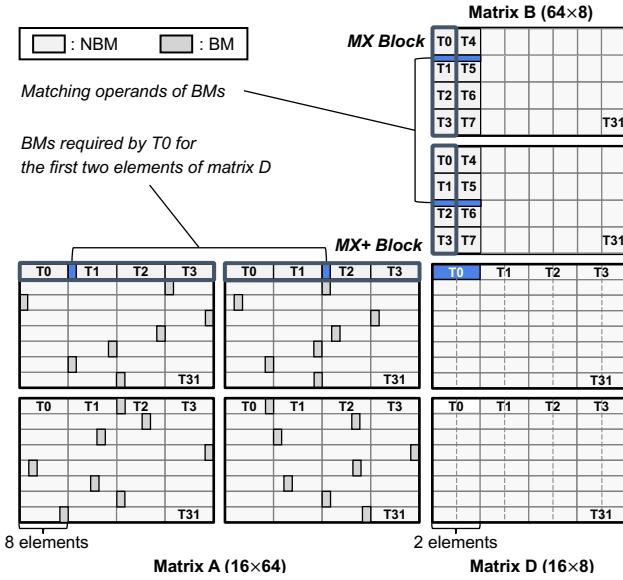


Figure 8: Elements of matrix A (MXFP4+), matrix B (MXFP4), and matrix D (FP32) are distributed across the 32 threads (Tx where x is a thread ID) in a warp.

instructions such as `wmma.mma`, `mma`, and `wgmma.mma_async`, which are translated into device-specific machine code (i.e., SASS instructions) such as HMMA (half-precision) and IMMA (integer). We discuss this section based on the following MXFP4 `mma` PTX instruction without loss of generality [46].

```
mma.m16n8k64.block_scale.f32. e2m1.e2m1.f32.ue8m0 D, A, B, C, EA, EB.  
                                         FP4 Precision Scale Format
```

This instruction operates on matrices A and B with dimensions 16×64 and 64×8 , respectively, and matrices C and D with dimensions 16×8 . Matrices E_A and E_B , each with dimensions 16×2 and 2×8 , store the shared exponents of matrices A and B for two MX blocks per row and per column, respectively.

When a warp executes a machine instruction for an MMA operation, all 32 threads within the warp collectively perform matrix multiplication for a specific tile shape. To achieve this, each thread in the warp holds a subset of elements, referred to as a *fragment*, of the operand matrices in its registers.

Figure 8 illustrates how the elements of matrix A (MXFP4+ activation), matrix B (MXFP4 weight), and the resulting matrix D (FP32) are distributed across the threads in a warp for the 4-bit `mma.m16n8k64` PTX instruction. For matrix A, each thread holds a fragment in four 32-bit registers, with each register containing eight 4-bit elements, while for matrix B, each thread holds a fragment in two 32-bit registers, each containing eight 4-bit elements. For matrix D, each thread holds four 32-bit elements in four 32-bit registers. Note that each row of matrix A corresponds to two MX+ blocks, while each column of matrix B represents two MX blocks.

In MXFP4+, BM is effectively represented in E2M3, with the private exponent of e_{\max} , whereas the FP4 compute units in the Tensor Core operate on E2M1. Thus, we cannot simply perform an MMA operation with the input matrices A and B. To address this, we decompose the BM value in each MXFP4+ block into the sum

of two values, BM_H and BM_L , as follows:

$$\begin{aligned} BM &= (-1)^s \times 2^{e_{\max}} \times um[3:0] \\ &= BM_H + BM_L, \end{aligned} \quad (3)$$

$$\text{where } BM_H = (-1)^s \times 2^{e_{\max}} \times um[3:2],$$

$$BM_L = (-1)^s \times 2^{e_{\max}-2} \times um[1:0],$$

where $um[3:0]$ denotes a mantissa representation with a leading one explicitly stored in $um[3]$, as in the x86 80-bit extended-precision format [20]. As shown in Equation 3, BM_H and BM_L are effectively E2M1 and can be stored in FP4. Thus, we can process the BM elements of matrix A using the following steps.

- Split BM into BM_H and BM_L .
- Replace BM with BM_L and perform an MMA operation.
- Multiply BM_H with the corresponding elements in matrix B and accumulate the results into matrix D.

Note that additional computation is needed beyond the MMA operation to obtain the correct output of matrix D (i.e., the third step for BM_H). One possible approach to address this is to exploit CUDA cores with FMA operations, while the Tensor Core performs an MMA operation for the second step. However, we observe that this results in more than a 5× slowdown in overall matrix computation with MX+ and MX blocks on the RTX 5090 GPU, compared to matrix multiplication with only MX blocks. This is because each FP4 element must be converted to higher precision (e.g., BF16 or FP32) to perform FMAs in the CUDA core. Furthermore, this also requires each thread to fetch the data from other threads via inter-thread communication (e.g., warp shuffling). For example, in Figure 8, thread 0 (T_0) needs to fetch BM_H s in matrix A from threads 1 and 2 (T_1 and T_2) and the matching operands in matrix B from threads 1, 2, 5, and 6 (T_1 , T_2 , T_5 , and T_6) to accumulate the multiplication result into the first two elements of matrix D.

5.2 Using Underutilized Tensor Cores

To avoid costly conversions and reduce inter-thread communications, we instead perform an additional MMA operation for BM_H while reusing the registers of each thread. We observe that this approach maintains performance in the decode stage where compute units are underutilized, while introducing a modest increase in inference time during the prefill stage (Section 7.3).

In detail, each thread first loads matrix fragments and the BM index, then checks if it holds the BM element using its thread ID. If a thread finds that it holds the BM, it replaces the BM with BM_L . For example, in Figure 8, the first four threads (T_0-T_3) load the same BM index for the first MX+ block of matrix A, which is 8, and compare their thread IDs with $\lfloor \frac{\text{BM index}}{8} \rfloor$. T_1 identifies the match and replaces the BM with BM_L . This process is repeated across all four registers that hold a fragment of matrix A.

To perform an additional MMA operation using BM_H and its matching operands, each thread requires a fragment of matrix A in separate registers containing only BM_H values, with all other elements set to zero. To achieve this, we first assign each thread to a single, exclusive MX+ block, from which it extracts the corresponding BM_H from the BM. For example, in the 16×64 matrix A shown in Figure 8, which contains 32 MX+ blocks, each of the 32 threads in a warp processes a distinct block and retrieves its BM_H value. Threads then prepare their matrix A fragments in registers

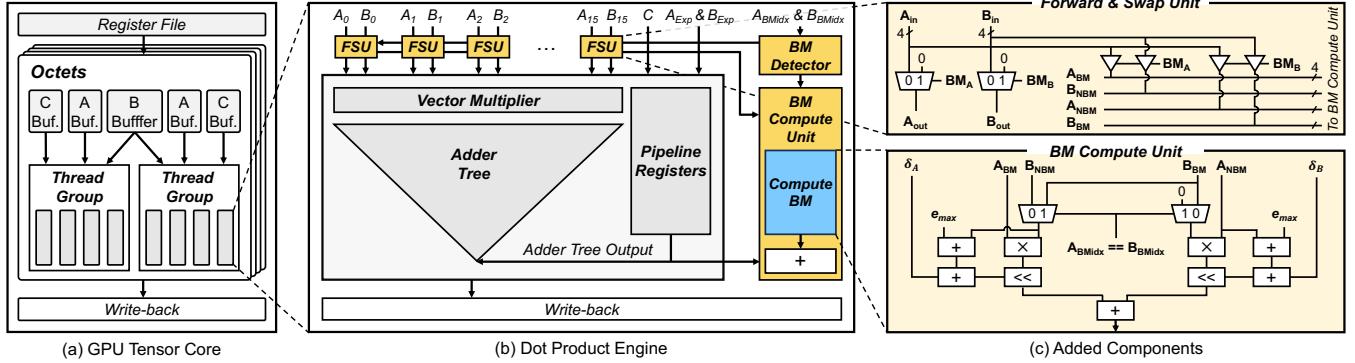


Figure 9: Overall design of hardware integration of MX+ into GPU.

Algorithm 1 Warp executing a sequence of MMA instructions for matrices A (MXFP4+) and B (MXFP4).

```

1  $\triangleright D[mn, nn] += A[mn, 128] \times B[128, nn]$ 
2  $\triangleright$  Input A, B, and BMIdx are in shared memory
3 procedure MMALoop(A, B, BMIdx)
4    $\triangleright$  Load operands to registers
5   a[mn, 128]  $\leftarrow$  LoadFragment(A)
6   b[nn, 128]  $\leftarrow$  LoadFragment(B)
7   bmIdx[mn, 4]  $\leftarrow$  Load(BMIdx)
8    $\triangleright$  Replace BM of A with BML
9   a  $\leftarrow$  ReplaceBM(a, bmIdx)
10   $\triangleright$  Fragments for additional MMAs
11  aBM[mn, 128]  $\leftarrow$  MakeFragment(A, bmIdx)
12  for k in 2 do
13    kk  $\leftarrow$  k · 64
14    for i in [mn / 16] do
15      ii  $\leftarrow$  i · 16
16      for j in [nn / 8] do
17        jj  $\leftarrow$  j · 8
18        mma.m16n8k64 d[ii, jj], a[ii, kk], b[jj, kk], d[ii, jj]
19         $\triangleright$  Perform additional MMA operation for BMH
20        if k == 1 then
21          mma.sp.m16n8k128 d[ii, jj], aBM[ii], b[jj], d[ii, jj]

```

for the additional MMA operation. Those requiring BM_H values retrieve them from the extracting thread and place them at the corresponding BM positions within their fragments.

Algorithm 1 shows the procedure (MMALoop) for performing the multiplication of MXFP4+ and MXFP4 matrices. ReplaceBM (Line 9) identifies the BM in a and replaces it with BM_L . MakeFragment (Line 11) extracts BM_H from the BM and stores it in the BM position of a_{BM} . Note that this process is amortized over multiple for-loop iterations. Finally, a sequence of MMA operations is executed, including an additional MMA ($mma.sp.m16n8k128$) for BM_H (Line 21), while reusing the registers that hold fragments of matrices B and D. We perform a sparse MMA operation, which is twice as fast as a dense MMA, because all elements in matrix A are zero except BM_H .

6 Architectural Support for MX+

The software integration of MX+ presented in Section 5.2 avoids conversion and reduces inter-thread communication overhead. However, it requires an additional MMA operation compared to the MX

format. In this section, we explore the hardware integration of MX+ into GPU systems that support the MX+ format.

6.1 GPU Integration Overview

We present a hardware design that supports the MX+ extension without making intrusive changes to the dot product engine (DPE) in Tensor Cores. During an MMA operation, Tensor Cores gather the input operands for matrix multiplication from different threads in a warp. We capture BMs and their matching operands at the DPE input, then perform BM-related operations using dedicated low-precision scalar compute units. This keeps all modifications confined to areas outside the core dot product pipeline in the DPE. Since only a few BM-related operands need to be computed while the DPE performs a dot product, both latency and area overheads remain negligible. The following section provides a detailed description of this approach.

6.2 Tensor Core Integration

Figure 9 shows the overall Tensor Core design with MX+ hardware integration. Our baseline Tensor Core architecture follows the design in prior work [50], with the difference that each warp executes on a single Tensor Core containing 32 DPEs. Four threads form a *threadgroup* that utilizes four DPEs, and two threadgroups combine to form an *octet*. A warp consists of four octets, with threads collaboratively loading operand matrices into intermediate buffers.

Each Tensor Core completes one FP4 $mma.m16n8k64$ every 16 cycles according to our benchmarking, which can also be inferred from the RTX 5090 specification [45]. Since each thread in a warp computes dot products for eight pairs of MXFP4 blocks to produce four output elements during MMA execution (see Figure 8), we configure each DPE to process one MXFP4 block pair every two cycles; i.e., each DPE processes 16 FP4 input pairs per cycle. Each pair of MXFP6 or MXFP8 blocks is computed every four cycles, since FP8 sustains half the throughput of FP4, and FP6 matches FP8 throughput. The BM indices of the blocks currently being processed (A_{BMIdx} and B_{BMIdx}) are supplied accordingly.

Hardware Extension. The DPE is extended with three main components: 1) BM Detector, 2) Forward and Swap Unit (FSU), and 3) BM Compute Unit (BCU). When the MX+ blocks and BM indices are fed into the DPE, the BM Detector checks the BM indices (A_{BMIdx} and B_{BMIdx}) and activates the corresponding FSUs by sending them

```

// Format: OMMA D, A, B, C, Aexp, Bexp, {Shared Exponent Metadata}
OMMA.SF.16864.F32.E2M1.E2M1.E8 R12, R100.reuse, R104, R12, R0.reuse, R3, URZ
OMMA.SF.16864.F32.E2M1.E2M1.E8 R16, R100.reuse, R106, R16, R0.reuse, R13, URZ
OMMA.SF.16864.F32.E2M1.E2M1.E8 R20, R100.reuse, R108, R20, R0.reuse, R15, URZ
...
(a) Original SASS Instructions

MOV R1, R32      // Move ABMDx (R32) to R1
MOV R4, R48      // Move BBMDx (R48) to R4
OMMA.SF.16864.F32.E2M1.E2M1.E8.R4 R12, R100.reuse, R104, R12, R0.reuse, R3, URZ
MOV R14, R49     // Move BBMDx (R49) to R14
OMMA.SF.16864.F32.E2M1.E2M1.E8.R4 R16, R100.reuse, R106, R16, R0.reuse, R13, URZ
MOV R16, R50     // Move BBMDx (R50) to R16
OMMA.SF.16864.F32.E2M1.E2M1.E8.R4 R20, R100.reuse, R108, R20, R0.reuse, R15, URZ
...
(b) Extended SASS Instructions

```

Figure 10: (a) Original SASS instructions performing matrix multiplication. (b) Extended SASS instructions with BM flag and additional source registers for BM indices.

1-bit B_{MA} and B_{MB} signals. Each FSU consists of several multiplexers and tri-state buffers, and shares a datapath connected to the BCU. When BM (B_{MA} , B_{MB}) signals are set, the FSU directs the BM input and its matching operand to the BCU while forwarding zero to the corresponding DPE input. This ensures that these inputs are excluded from computation in the dot product pipeline. To support FP6 and FP8, we can configure the FSUs such that those at even positions ($2i$, where $i = 0, 1, \dots, 7$) share one datapath, while those at odd positions ($2i+1$) share another. This enables the 4-bit inputs from the adjacent FSU to be routed to the BCU as well. The forwarded inputs are then processed in the BCU, as discussed below.

BM Computation within Tensor Core. As shown in Figure 9(c), the BCU takes as inputs the BMs and their matching operands along with the BM indices. It then performs the following computation:

$$\text{Output} = (A_{BM} \times B_{NBM}) + (B_{BM} \times A_{NBM}),$$

where A_{BM} and B_{BM} are the BMs of matrices A and B, and B_{NBM} and A_{NBM} are their matching operands. The first and second multiplication terms are conditionally left-shifted by δ_A and δ_B in MX++, which are the differences of the shared exponents between MX and MX++ for matrices A and B. These shifts are encoded in the reserved 3 bits of the BM index. Note that these operations complete faster than the DPE, which performs element-wise multiplications and multi-level additions using an adder tree, and do not cause pipeline stalls; i.e., the BM computation does not affect MMA instruction throughput. *Output* is then added to the output of the adder tree before being normalized and converted to FP32.

When the BM indices of A and B are identical, we simply swap B_{NBM} with B_{BM} and set B_{NBM} to zero, effectively computing only one of the two identical terms in the formula. We design the multipliers to support sufficiently high precision to handle scenarios where both the multiplicand and multiplier are BMs.

SASS Instruction Extension. Figure 10(a) shows SASS instructions performing an MXFP4 MMA operation, corresponding to the MXFP4 *mma* PTX instruction described in Section 5.1. The register identifier in the MMA instruction represents multiple consecutive registers [50], with only the lowest register identifier encoded in the instruction. For example, R12 in the first OMMA instruction represents a sequence of four registers: <R12, R13, R14, R15>. Note that R0 and R3, which contain the shared exponents, are single registers rather than register sequences.

Figure 10(b) shows the proposed SASS instruction. The MMA instruction is extended with a BM control flag, which indicates whether the input operands are represented in MX+. We extend the control information by 1 bit for the additional flag; the SASS instruction contains unused bits [22], which we similarly observed for Blackwell instructions through *nvdiasasm* [43]. We also extend the MMA instruction to take two additional source registers, each containing two 8-bit BM indices for matrices A and B, following the register layout of the shared exponent.

For instruction encoding of BM indices, we follow the scheme used by sparse MMA instructions for the MX format. These instructions implicitly encode the register that holds ordered metadata together with the register for the shared exponent of matrix A. The two registers are paired to form a sequence, and a single register identifier is encoded in the instruction. We adopt the same approach by pairing the BM index register with the shared exponent register. As illustrated in Figure 10(b), R32 and R48, each containing BM indices for matrices A and B, are copied to registers R1 and R4, respectively. In the subsequent OMMA instruction, R0 and R3 implicitly represent the register pairs <R0, R1> and <R3, R4>.

7 Evaluation

7.1 Experimental Methodology

Algorithmic Implementation. We implement MX+ on top of the CUDA extension of the MX PyTorch emulation library [39] and evaluate model performance using pre-trained models from Hugging Face [62]. Following prior work on applying MX formats to DNNs [52], we apply MX and MX+ formats to all tensors involved in any dot product operation during LLM inference, including those in the language modeling head and KV cache. We use BF16 for vector operations such as normalization.

Model and Workload. We evaluate MX+ across a diverse set of large language models: OPT-66B [70], Llama-3.1 (8B and 70B) [13], Mistral-7B-v0.3 [23], Phi-4-14B [1], and Qwen-2.5-14B-Instruct [64]. We measure task-specific accuracy (%) on the same lm-evaluation-harness tasks used in prior work [52] and assess language modeling performance using perplexity on the WikiText-2 [38] and C4 [10] datasets. Additionally, we use Llama-2 models [58] of varying sizes to measure inference time under MX+ integration.

Quantization Baselines. For model quantization, two primary approaches exist: post-training quantization (PTQ) and quantization-aware training (QAT). Our evaluation on LLMs uses quantized inference based on PTQ, which does not involve any re-training or fine-tuning. We follow the drop-in replacement PTQ scenario described in prior work [7], where a pre-trained BF16 model is directly cast into MX or MX+ formats for evaluation—referred to as *direct-cast* inference. To demonstrate that MX+ can also help in other contexts, we additionally present the results with quantization-aware fine-tuning on vision and CNN models in Section 8.2.

We assess model performance when using MX+ for both activations and weights, and compare each against its counterpart (i.e., MXFP4, MXFP6, and MXFP8). We use E2M3 and E4M3 for six-bit and eight-bit formats, as prior work shows better accuracy with higher mantissa bit configurations among the two variants in each format [52]. We also evaluate MXFP4++ (applied to both weights and activations) and A-MXFP4+ (MXFP4+ only for activations).

Table 2: Direct-cast inference results of zero-shot tasks. Higher is better for all tasks. All results are zero-shot except for \dagger -marked tasks in OPT-66B, which are 5-shot due to poor BF16 performance.

Model	Format	ARC easy	ARC challenge	Lambada	College CS \dagger	Int. law \dagger	Juris-prudence \dagger
OPT 66B	BF16	67.26	39.76	73.63	39.00	29.75	25.00
	MXFP8+	66.88	39.51	73.37	37.00	31.40	31.48
	MXFP8	65.99	37.88	73.18	28.00	29.75	27.78
	MXFP6+	66.58	40.10	73.55	36.00	28.10	24.07
	MXFP6	66.08	38.82	72.66	34.00	31.40	26.85
	MXFP4++	62.54	35.92	70.70	29.00	33.06	27.78
	MXFP4+	62.08	36.86	69.26	25.00	30.58	23.15
	A-MXFP4+	60.14	33.96	67.82	26.00	32.23	30.56
	MXFP4	35.23	24.83	02.97	24.00	19.83	25.93
Llama-3.1 8B	BF16	81.19	53.33	75.39	54.00	82.64	73.15
	MXFP8+	81.02	54.01	75.14	53.00	80.99	71.30
	MXFP8	79.88	53.16	75.22	43.00	79.34	75.00
	MXFP6+	80.72	53.16	75.26	51.00	79.34	74.07
	MXFP6	80.22	52.90	75.06	47.00	79.34	75.00
	MXFP4++	70.71	46.16	68.10	34.00	70.25	59.26
	MXFP4+	70.29	45.65	66.83	38.00	66.12	54.63
	A-MXFP4+	68.56	41.47	66.83	41.00	56.20	50.00
	MXFP4	49.41	29.69	40.17	26.00	23.97	23.15
Llama-3.1 70B	BF16	86.49	64.85	78.91	64.00	89.26	85.19
	MXFP8+	86.78	64.25	79.22	63.00	89.26	86.11
	MXFP8	87.04	64.59	78.71	65.00	89.26	86.11
	MXFP6+	85.40	63.91	78.73	63.00	86.78	85.19
	MXFP6	85.27	63.14	78.42	64.00	88.43	85.19
	MXFP4++	81.65	58.11	72.81	58.00	84.30	84.26
	MXFP4+	79.17	54.86	72.70	57.00	87.60	82.41
	A-MXFP4+	78.11	53.16	70.68	52.00	84.30	80.56
	MXFP4	68.18	44.88	61.91	45.00	75.21	63.89
Mistral 7B	BF16	78.32	52.22	75.26	53.00	76.03	70.37
	MXFP8+	77.90	51.19	75.04	52.00	73.55	72.22
	MXFP8	78.54	52.22	74.99	50.00	74.38	71.30
	MXFP6+	78.45	51.88	74.85	51.00	76.03	70.37
	MXFP6	78.32	52.73	74.93	52.00	76.03	68.52
	MXFP4++	75.67	49.06	71.43	42.00	68.60	57.41
	MXFP4+	74.20	47.78	70.79	45.00	67.77	65.74
	A-MXFP4+	74.28	47.78	72.40	40.00	60.33	54.63
	MXFP4	69.57	43.26	65.17	31.00	47.93	43.52
Phi-4 14B	BF16	72.90	55.97	72.50	65.00	90.91	83.33
	MXFP8+	72.94	56.23	72.13	67.00	90.08	81.48
	MXFP8	73.36	56.74	72.13	66.00	90.08	84.26
	MXFP6+	71.63	55.46	71.82	67.00	89.26	82.41
	MXFP6	72.26	55.46	71.78	68.00	90.08	83.33
	MXFP4++	71.63	55.46	69.88	63.00	90.08	78.70
	MXFP4+	72.47	54.95	67.94	64.00	90.08	82.41
	A-MXFP4+	72.31	54.95	68.87	65.00	88.43	83.33
	MXFP4	72.35	53.24	64.43	58.00	86.78	84.26
Qwen-2.5 14B	BF16	81.52	62.46	72.87	71.00	87.60	87.04
	MXFP8+	81.27	61.77	72.85	72.00	88.43	87.04
	MXFP8	80.81	61.69	72.48	72.00	88.43	87.04
	MXFP6+	81.06	60.58	72.02	71.00	89.26	86.11
	MXFP6	80.22	60.75	72.23	70.00	89.26	85.19
	MXFP4++	78.91	57.68	67.88	67.00	88.43	85.19
	MXFP4+	77.15	54.61	66.12	66.00	87.60	82.41
	A-MXFP4+	75.72	52.90	65.42	69.00	81.82	82.41
	MXFP4	69.57	48.89	51.89	48.00	71.07	68.52

MX+ Software Integration. We consider two use-case scenarios for MX+ software integration, based on whether the system natively supports MX formats: 1) data stored in MX formats are converted to BF16 before computation, and 2) MX formats are computed directly within hardware. For the first scenario, we extend the Triton compiler [57]—which already supports MX formats for matrix multiplication via BF16 conversion—by implementing conversion code for Block Max (BM) (i.e., Equation 2) within the matrix multiplication kernel. We evaluate this using BF16 for activations and MXFP4 variants for weights on an NVIDIA RTX A6000 GPU [40], which

Table 3: Perplexity of different models via direct-cast inference. The sequence lengths are 1024 (Top) and 2048 (Bottom). Lower is better.

Model	OPT 66B		Llama-3.1 8B		Mistral 7B		Phi-4 14B		Qwen-2.5 14B	
	Wiki2	C4	Wiki2	C4	Wiki2	C4	Wiki2	C4	Wiki2	C4
BF16	10.66	10.90	6.97	9.40	3.17	7.22	5.93	8.26	7.49	14.82
MXFP8+	10.70	10.92	7.06	9.51	3.23	7.26	5.95	8.28	7.55	14.89
MXFP8	11.20	11.11	7.13	9.63	3.28	7.30	6.00	8.32	7.57	14.91
MXFP6+	10.75	10.94	7.10	9.58	3.36	7.34	5.98	8.30	7.56	14.92
MXFP6	11.24	11.19	7.19	9.70	3.43	7.39	6.01	8.33	7.58	14.94
MXFP4++	12.53	11.85	9.87	13.08	5.34	8.94	6.57	8.97	8.28	15.93
MXFP4+	12.74	12.01	10.14	13.37	5.79	9.36	6.67	9.08	8.43	16.19
A-MXFP4+	13.92	13.64	11.03	14.64	6.03	9.65	6.93	9.45	8.52	16.36
MXFP4	167.61	276.80	27.69	33.80	9.15	13.72	10.06	13.18	9.47	17.45
BF16	9.35	10.15	6.27	8.62	2.81	6.44	5.32	7.81	6.67	13.45
MXFP8+	9.39	10.17	6.35	8.73	2.86	6.48	5.34	7.83	6.72	13.51
MXFP8	9.82	10.38	6.42	8.84	2.91	6.51	5.36	7.86	6.74	13.53
MXFP6+	9.43	10.19	6.38	8.79	2.98	6.56	5.35	7.85	6.74	13.54
MXFP6	9.94	10.44	6.46	8.90	3.04	6.60	5.38	7.88	6.75	13.56
MXFP4++	11.17	11.18	9.22	13.10	4.81	8.34	5.90	8.52	7.36	14.49
MXFP4+	11.35	11.31	9.54	13.41	5.25	8.58	5.97	8.62	7.48	14.73
A-MXFP4+	12.63	13.63	10.46	15.00	5.45	8.96	6.27	9.04	7.58	14.87
MXFP4	209.83	306.33	27.38	36.41	8.43	13.80	9.96	14.40	8.45	15.99

lacks native MX support. For the second scenario, we implement our algorithm in Section 5.2 using the CUTLASS library [56] and integrate our matrix multiplication kernel into vLLM [31]. We then evaluate its performance on an NVIDIA RTX 5090 GPU [45], which provides native hardware support for MX formats.

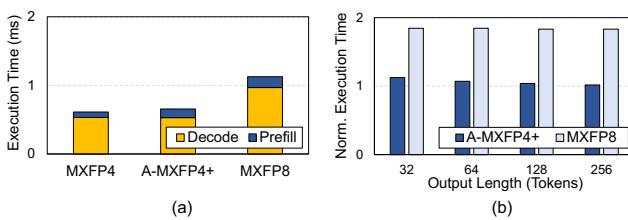
MX+ Hardware Integration. We implement the components added for the MX+ GPU integration in RTL and synthesize them using Synopsys Design Compiler with a commercial 28nm technology node. To evaluate performance with MX+ integration, we extend AccelSim [27] with the configurations similar to the NVIDIA RTX 5090 GPU. Each instruction for an MMA operation is modified to include additional access to the register file. We also model the latency of adding the BCU output to the adder tree result for the MMA instruction. The matrix multiplication traces are generated using the CUTLASS library.

7.2 Language Model Performance

Table 2 shows the accuracy of the baseline BF16, MX, and MX+ for the lm-evaluation-harness tasks used in [52]. Overall, MX+ improves accuracy over its MX counterparts, with MXFP8+ and MXFP6+ achieving improvements of up to 10.00 and 4.00 percentage points. The accuracy difference between MXFP4 and MXFP4+ is particularly substantial (up to +42.15%, excluding OPT-66B where MXFP4 does not work), and even the case where MXFP4+ is used only for activations (A-MXFP4+), while MXFP4 is still used for weights, still significantly outperforms MXFP4. This indicates that representing activation outliers becomes challenging in low-precision formats, and MX+ effectively addresses this problem by representing BMs with higher precision. Importantly, MXFP4+ achieves this while maintaining hardware and memory efficiency by representing all elements uniformly in four bits like MXFP4. Building on MXFP4+, MXFP4++ further improves accuracy by also representing NBMs more accurately, achieving up to +4.63% higher accuracy compared to MXFP4+. Consistent with the accuracy results, Table 3 shows that both MX and MX+ always achieve lower perplexity than the original MX formats across sequence lengths and datasets.

Table 4: Matrix multiplication time with BF16 activation and MXFP4+ or MXFP4++ weight, normalized to the MXFP4 weight case.

Normalized Time N=4096, K=4096	Small Activations			Large Activations		
	M=8	M=16	M=32	M=1024	M=2048	M=4096
MXFP4+	1.08	1.07	1.08	1.04	1.01	1.01
MXFP4++	1.08	1.09	1.10	1.04	1.05	1.04

**Figure 11: (a) Execution time breakdown with 64 output tokens. (b) Normalized execution time across output tokens.**

7.3 Performance of MX+ Software Integration

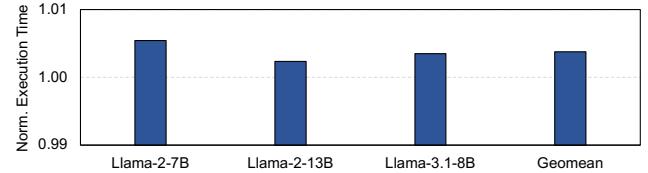
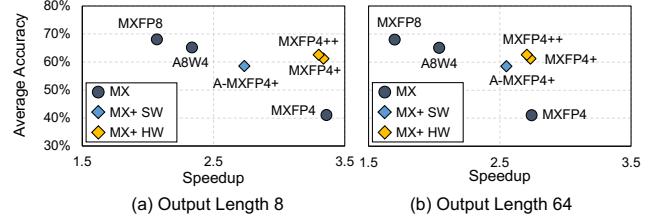
In this section, we evaluate two use-case scenarios of MX+ software integration, as discussed in Section 7.1.

Conversion Before Computation. Table 4 presents the execution time for matrix multiplication using BF16 activations and MXFP4+ (or MXFP4++) weights, normalized to the MXFP4 weight case. The execution times reported in Table 4 include both BF16 conversion overhead and BF16 MMA operations. Note that no additional MMA operation is required for MXFP4+ (or MXFP4++) in this case. Matrix multiplication kernels are generated from Triton [57] for the dimensions spanning from low (small activations) to high (large activations) data reuse scenarios. The results show that the BM handling overhead during BF16 conversion becomes more pronounced with smaller activations than larger ones. In high reuse cases, BF16 MMAs dominate the overall matrix multiplication time, while amortizing the conversion overhead. Note that, in both cases, the additional BM handling required during conversion introduces only a small performance overhead over the MX.

Direct Computation. In the following sections, we refer to the aggregated matrix multiplication time during LLM inference in vLLM for a given number of concurrent requests as *execution time*. Figure 11(a) shows the execution time during the prefetch and decode stages for Llama-2-13B, with four requests of 1024 input tokens and 64 output tokens. The results show that A-MXFP4+ performs close to MXFP4, while MXFP8 leads to a large slowdown. Since the decode stage that dominates the execution time is memory-bounded, an additional MMA operation in A-MXFP4+ incurs a negligible performance overhead (6.7%). A-MXFP4+ shows a moderate slowdown in the prefetch stage (1.54x), which comprises 18.78% of the execution time. Figure 11(b) shows the execution time across different output tokens, normalized to MXFP4. A-MXFP4+ shows up to a 1.13x slowdown, while MXFP8 shows up to a 1.85x slowdown compared to MXFP4. As the number of output tokens increases in Figure 11(b), the decode stage accounts for a larger portion of the execution time, reducing the gap between MXFP4 and A-MXFP4+.

7.4 MX+ Hardware Integration

Performance. Figure 12 shows the execution time of MXFP4+ with hardware integration during the prefetch stage for a request with 2048

**Figure 12: Normalized execution time of MXFP4+.****Figure 13: End-to-end inference speedup over BF16 and average accuracy of lm-eval-harness tasks on Llama-2-13B. A8W4 uses MXFP8 for activations and MXFP4 for weights.**

input tokens, normalized to MXFP4. Overall, MXFP4+ shows a 0.38% slowdown on average compared to MXFP4. This is because the BCU computation does not affect the throughput of the instructions for MMA operations. Extra register file access and increased instruction latency have a negligible impact on performance.

Area and Power. Table 5 shows the area and power of the additional components for MX+ per Tensor Core. We add 16 FSUs, a BM Detector, and a Compute Unit for each 32 DPEs in the Tensor Core. Our design has an area of 0.020mm^2 and a power consumption of 12.11mW . Note that directly comparing the area with the RTX 5090 is not feasible since we use a 28nm technology node while the GPU is fabricated using a more advanced node (4nm). However, we believe the area overhead would be even smaller if fabricated using more advanced node. The area overhead of MX+ is much smaller than recent quantization work that integrates hardware components into Tensor Cores, such as RM-STC [19] and OliVe [15].

End-to-End Speedup. Figure 13 shows the speedup over BF16 in vLLM and the average accuracy for lm-eval-harness tasks on Llama-2-13B. We evaluate performance using four requests of 1024 input tokens with either 8 or 64 output tokens, representing scenarios where the prefetch or decode stage respectively dominates inference time. We compare MX formats with MXFP4+ and MXFP4++ under hardware support, as well as A-MXFP4+ under software support. For the long output length, A-MXFP4+ achieves a speedup close to MXFP4 with 17.46% higher accuracy. We modify the CUTLASS library to reduce unnecessary data loading and computation when an output matrix dimension is smaller than the tile size of a thread block. The library supports a single tile shape for A8W4, where each thread block computes an $M=128$ and $N=128$ output tile [44],

Table 5: Area and power for MX+ support per Tensor Core.

Component	Configuration	Area [mm^2]	Power [mW]
Forward and Swap Unit	$32 \times (16 \text{ units})$	0.004	0.59
BM Detector	32	0.004	2.86
BM Compute Unit	32	0.012	8.66
Total		0.020	12.11

Table 6: Total quantization time normalized to MXFP4.

Input Tokens	32	128	512	1024	2048
MXFP4+	1.00	1.00	1.03	1.05	1.05
MXFP4++	1.05	1.04	1.13	1.15	1.15

Table 7: Perplexity on WikiText-2 via direct-cast inference.

Scheme	OPT 66B	Llama-2			Llama-3.1		Mistral	Phi-4	Qwen-2.5
		7B	13B	70B	8B	7B	14B	14B	
BF16	9.35	5.47	4.89	3.32	6.27	2.81	5.32	6.67	5.70
SMQ (INT4)	5E+4	3E+3	5E+3	1E+3	8E+3	2E+4	1E+3	32.50	3E+3
SMQ (MXFP4)	33.96	10.44	9.94	6.03	21.00	7.38	9.72	8.13	10.13
QuaRot (INT4)	138.88	8.11	6.07	4.12	9.85	35.76	6.32	7.55	9.71
QuaRot (MXFP4)	13.49	13.41	7.12	4.17	9.60	19.47	6.63	7.63	8.17
Atom (INT4+INT8)	9.42	5.94	5.21	3.66	7.43	4.22	5.70	7.16	6.77
ANT	1E+4	180.81	146.11	32.83	927.90	1E+5	4E+2	22.06	1E+2
OliVe	6E+3	86.27	3E+3	97.88	3E+2	4E+4	43.40	14.66	13.59
Tender	12.38	36.47	55.08	13.43	70.74	3E+4	38.88	19.40	2E+2
MX-ANT	10.35	6.10	5.33	3.79	8.11	4.37	5.88	7.17	6.81
MX-OliVe	20.89	6.33	5.50	3.91	8.31	4.71	6.01	7.31	7.24
MX-Tender	9.58	6.32	5.47	3.84	9.42	11.58	6.22	7.25	7.63
MXFP4+	10.54	5.87	5.17	3.58	7.22	3.85	5.65	6.97	6.66
MXFP4++	10.46	5.84	5.16	3.56	7.14	3.78	5.58	6.95	6.54

whereas M of the output matrix is 4 in the decode stage. Even with this optimization, A8W4 performance remains close to MXFP8.

With hardware support, MXFP4+ delivers speedups comparable to MXFP4, achieving 3.34× and 2.73× improvements over BF16 in prefill and decode-dominant scenarios, respectively, while providing 20.17% higher accuracy. Despite requiring additional computation to find the second maximum magnitude during conversion, MXFP4++ maintains competitive performance, running only 1.04% and 1.00% slower than MXFP4. Table 6 shows the quantization time across the input token lengths. MXFP4 exhibits quantization times similar to MXFP4, while MXFP4++ shows a small increase. Since quantization accounts for only a small portion of inference time, this overhead has a negligible impact on overall performance.

8 Analysis and Discussion

8.1 Comparison with Other Schemes

In this section, we compare model performance between MX+ and other algorithm-only or algorithm-hardware quantization schemes. For a fair comparison, we quantize matrix multiplication between weights and activations, excluding language modeling head—the intersection of quantized operations across the schemes.

SmoothQuant [63] rescales activation channels, while QuaRot [3] rotates activation using orthogonal matrices to reduce the overall magnitude. Atom [73] reorders channels and quantizes the channels with outliers using higher precision (INT8). Table 7 shows that SmoothQuant (SMQ) falls short in 4-bit precision, as discussed in multiple studies [3, 32, 35]. We observe that QuaRot does not completely remove outliers and performs worse than MXFP4+; the magnitude of outlier values is not reduced after rotation (e.g., the down projection layer in Llama-3.1). MX+ focuses on precisely representing important outlier values, unlike QuaRot, as the fine-grained grouping of MX already limits the impact of outliers to other values. Atom shows comparable model performance as it represents outliers in 8-bit but still performs worse than MX+.

ANT [16] and OliVe [15] use custom formats, while Tender [32] groups channels of similar range and uses standard INT4. As shown in the table, they suffer at 4-bit due to coarse channel-group or

Table 8: Direct-cast perplexity of different weight formats under BF16 activation with AWQ (A16W4) and MXFP8 activation (A8W4).

Activation Weight	BF16 & AWQ			MXFP8	
	INT4	MXFP4	MXFP4+	MXFP4	MXFP4+
Llama-3.1-8B	6.67	6.99	6.64	7.40	6.97
Mistral-7B	5.44	5.56	5.42	5.67	5.52

Table 9: Top-1 accuracy (%) on ImageNet.

Family	Model	FP32	Direct-cast		QA fine-tuning	
			MXFP4	MXFP4+	MXFP4	MXFP4+
Vision	DeiT-Tiny	71.64	61.79	66.60	68.96	69.96
Transformer	DeiT-Small	79.82	73.82	77.29	76.96	77.44
CNN	ResNet-18	69.18	49.28	62.66	65.76	66.72
	ResNet-34	74.55	52.67	64.35	69.97	71.40

tensor-wise grouping. We extend the schemes to support finer-grained grouping solely for accuracy comparison purposes, referred to as MX-ANT, MX-OliVe, and MX-Tender, though this would noticeably increase their runtime overhead. MX-Tender groups channels of each two rows at runtime. MX-ANT and MX-OliVe support group-wise quantization of size 32. Both adaptively select per-group data types for weights and per-tensor data types for activations. All schemes use floating-point scaling factors calculated per group at runtime. Nevertheless, MX+ still shows better performance, demonstrating the effectiveness of representing BMs in high precision.

8.2 Broader Applicability of MX+

Weight-Only Quantization. While MX+ primarily targets the scenario where we want to use low-bit precision for both weight and activation tensors, it also provides the benefit for weight-focused quantization scenarios. Table 8 shows the perplexity when employing 4-bit data formats for weights, with activations in either BF16 using AWQ [34] or MXFP8. AWQ is a weight-only quantization method, which scales important weight channels to larger magnitudes to protect them under low-bit quantization. Although directly using MXFP4 with AWQ degrades model performance, MX+ can synergistically operate with AWQ. This is because scaling up the important channels allows more important weight elements to be identified as BM. As a result, the model performance is further enhanced compared to the original AWQ (Weight INT4). When using MXFP8 activations with MXFP4 weights, precise representation of weights can be more critical than activations, as it uses half the number of bits. Using MXFP4+ under this setting noticeably enhances the model performance, as shown in the table.

Other DNN Workloads. We also evaluate MX+ on Vision Transformer [59] and CNN models [18] for image classification tasks on the ImageNet dataset [53]. Table 9 shows that the accuracy of MXFP4+ is higher compared to MXFP4, with improvements of up to +4.81% and +13.38% for the DeiT and ResNet models, under direct-cast inference. As discussed in prior work [11, 35, 54, 72], we observe that activation outliers are also present in these models and are typically scattered across MX blocks. MX+ represents these outliers more precisely, thereby improving accuracy.

We also perform quantization-aware (QA) fine-tuning and evaluate the effectiveness of MXFP4+ on the fine-tuned models. Compared to direct-cast inference, the accuracy gap between MXFP4

Table 10: Perplexity on WikiText-2 across non-FP microscaling formats in a direct-cast setting.

Model	MXINT8+	MXINT8	MXINT4+	MXINT4
Llama-3.1-8B	6.286	6.287	12.850	14.339
Mistral-7B	5.321	5.321	6.841	7.156

Table 11: Direct-cast inference accuracy for NVFP4 and NVFP4+ (NVFP4 with extra precision for BM) on lm-eval-harness tasks.

Model	Scheme	ARC easy	ARC challenge	Lam- bada	College CS	Int. law	Juris- prudence
Llama-3.1-8B	NVFP4	68.81	45.73	56.51	30.00	63.64	53.70
	NVFP4+	72.31	46.25	69.36	35.00	71.90	56.48
Mistral-7B	NVFP4	74.03	48.46	70.25	41.00	69.42	60.19
	NVFP4+	74.75	48.63	70.81	43.00	74.38	62.96

and MXFP4+ becomes narrower, as the fine-tuned models for image classification tasks can achieve accuracy close to the FP32 baseline even when using MXFP4. However, for more complex models and challenging tasks where fine-tuned MXFP4 models cannot reach FP32-level accuracy, the difference in accuracy between MXFP4 and MXFP4+ is likely to be more pronounced.

Applicability of MX+ to Non-FP Microscaling Formats. Beyond the three MXFP variants, the OCP MX specification defines one additional MX-compliant format with integer element data type: MXINT8. Although MXINT8 lacks an exponent field in its element data type, the approach of adding extra precision to the BM element could be similarly applied to MXINT8. For instance, the INT8 encoding in MXINT8 uses one sign bit, one integer bit, and six fractional bits. In this configuration, e_{\max} equals zero in Equation 1 since element values are always smaller than 2. The shared exponent becomes simply the exponent of the BM value, while the BM element is represented in the $\pm 1.xxxxxx$ format. This allows us to potentially make the integer bit implicit and use it as an extra fractional bit for the BM element. Table 10 shows perplexity results for this method applied to MXINT8 and a *hypothetical* MXINT4 format (one sign bit, one integer bit, two fractional bits). For MXINT8, increasing fraction bits from six to seven barely helps. In contrast, MXINT4 benefits from additional fraction bits, similar to MXFP4+ or MXFP6+. If MXINT4 becomes part of the concrete MX-compliant formats, this direction might be worth exploring as well.

Applicability of MX+ to NVFP4. NVIDIA recently introduced NVFP4, a 4-bit floating-point format that bears similarity to MXFP4. Both formats use FP4 (E2M1) elements with a shared scale per block. However, NVFP4 differs by using a smaller block size of 16 elements and an E4M3 scale factor. Table 11 presents the direct-cast accuracy results for NVFP4. When compared to the results in Table 2, MXFP4+ and MXFP4++ perform better than or comparably to NVFP4. This is because outliers are typically represented more accurately in MX+ due to the extra precision for BMs.

The MX+ extension can be similarly applied to NVFP4 since both MXFP4 and NVFP4 map the BM as closely as possible to the maximum representable magnitude in FP4 when computing scale factors [42]. Similar to MX+, we extend the mantissa bits of the BM element in NVFP4, except when the BM magnitude is extremely small that the exponent in the element data type is not set to maximum (i.e., when the shared scale becomes $X_{E4M3} \leq$

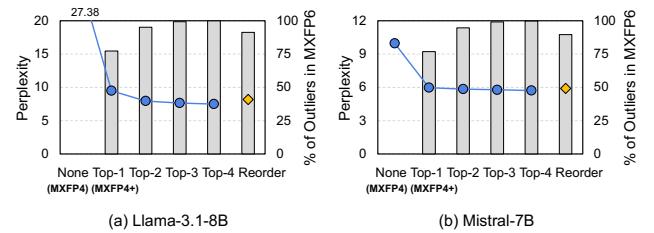


Figure 14: Perplexity when representing *top-k* magnitude elements of each block in MXFP6 while others in MXFP4 in a direct-cast setting. Bar plot represents the percentage of outliers in MXFP6.

00000010_2). In such cases, we use the original NVFP4 representation for the block. Note that the frequency of such cases can be reduced through an extra per-tensor software scaling step, which shifts values to larger magnitudes for per-block scaling. The extended NVFP4, termed NVFP4+, has an additional 4 bits per block to store the BM index, which can be packed with BM indices from other blocks for byte alignment. As shown in Table 11, NVFP4+ achieves higher accuracy than NVFP4.

Support for MX+ in Systolic Array Variants. Instead of performing matrix multiplications on GPUs, one can also consider using fixed-function matrix pipelines such as TPU [25]. These pipelines typically implement weight-stationary or output-stationary systolic array designs [25, 26], where each processing element (PE) performs one MAC operation per cycle. Supporting MX+ in these pipelines can be done similarly to the GPU integration by adding FSUs and BCUs to the datapath. For example, in a representative 32×32 MX-compliant systolic array, an FSU is attached to each PE, with a single BCU shared among the PEs in each column. In the weight-stationary dataflow, the PEs in a column collectively perform a dot product of an MX block pair. The BCU is located below the systolic array and receives BM values and their matching operands forwarded by FSUs, along with a partial sum. It then computes the BM-related operands, adds the result to the partial sum, and forwards the value to the accumulator. The process is similar for the output-stationary dataflow. Each PE performs a dot product of an MX block pair over 32 cycles, with FSUs collecting BM-related operands. After these cycles, the operands and partial sum are forwarded to the BCU. The updated partial sum is then routed back to each PE where the accumulator resides.

8.3 Addressing Multiple Outliers in a Block

Although we keep the proposed algorithm simple to minimize end-user overhead and enable seamless integration with various frameworks, model performance can be further improved when MX+ is optimized to capture outliers co-locating in the same blocks.

Outlier Analysis. We represent the *top-k* magnitude elements in MXFP6 for each block while keeping others in MXFP4. Figure 14 shows perplexity and the percentage of outliers represented in MXFP6 within activation tensors. We identify outliers using the 3σ rule following [15] and focus on activations, as weights generally have a lower impact on model performance. The results show that extending MX+ to store additional BM index to track up to two outliers provides some gain, while additional outlier representation shows diminishing returns. This is because most activation outliers are represented in higher precision at *top-2*. To balance complexity

Table 12: Direct-cast inference accuracy on lm-eval-harness tasks. *Reorder* denotes MXFP4+ with channel reordering applied to query and key matrices.

Model	Scheme	ARC easy	ARC challenge	Lam- bada	College CS	Int. law	Juris- prudence
Llama-3.1-8B	MXFP4+	70.29	45.65	66.83	38.00	66.12	54.63
	Reorder	72.77	46.08	68.58	42.00	77.69	62.04
Mistral-7B	MXFP4+	74.20	47.78	70.79	45.00	67.77	65.74
	Reorder	75.42	48.55	71.61	49.00	69.42	66.67

and model performance, we choose channel reordering as an optional optimization on top of MX+ to *explicitly* separate outliers in the same block. When applying channel reordering with MX+, the perplexity and percentage closely follow that of *top-2* as shown in Figure 14, which we discuss in detail in the remaining section.

Scattering Outliers with Channel Reordering. As shown in Figure 4, activation outliers are typically concentrated at the channel granularity. To allow more outliers to be identified as BM, we can also *explicitly* scatter outliers across blocks via channel-wise reordering. For instance, we first sort the channels of each activation based on the number of outliers. Channels with the most outliers are then placed one in every 32 (i.e., block size) channels. The remaining sorted channels are split in half, and we arrange the lower half channels in the remaining places in descending order, followed by the upper half channels in the same manner.

Table 12 shows the accuracy results for channel-wise reordering, denoted as *Reorder*. The improvement stems from more precise outlier representations. For example, the percentage of blocks with multiple outliers among outlier-containing blocks decreases from 22.52% to 4.58% in a sampled query matrix after reordering. For each task, we predetermine the channel ordering of query and key matrices by averaging outlier counts per channel between the two matrices using 10% of samples. Both matrices use identical channel ordering to maintain mathematical correctness. Reordering is fused with quantization by storing each quantized output at its reordered channel address, making the reordering overhead negligible.

9 Related Work

Block-Based Data Formats for DNNs. Block floating point (BFP) has been adopted for efficient inference and training of DNNs in academia [5, 8, 12, 29, 36, 37, 65, 68, 71]. FAST [71] uses different BFP precisions for different layers and training iterations during DNN training. Bucket Getter [37] implements intermediate accumulators in BFP PEs, each of which accumulates values within a similar exponent range. In recent years, industry has also increasingly explored BFP variants for efficient DNN processing [6, 7, 14, 30]. However, defining different data formats across organizations increases end-user overhead and software fragmentation. To address this, multiple companies collaborated to standardize data formats and introduced Microscaling formats [47], which are increasingly integrated into computing systems via software and hardware support [2, 21, 41]. MX+ builds on this standard with a non-intrusive extension, making it easily deployable across a wide range of platforms.

Outlier-Aware Quantization. Efficient execution of DNN workloads has been extensively studied in various contexts [4, 9, 24, 31, 33, 49, 51, 60, 69]. Quantization is one of the most widely used

Table 13: Comparison of different quantization schemes.

Scheme	AWQ	SqueezeLLM	SMQ	QuaRot	OliVe	Tender	LLM-FP4	MX+
Compute Efficiency?	✗	✗	✓	✓	✓	✓	✓	✓
Standard & General?	✓	✓	✓	✓	✗	✓	✗	✓
High Accuracy?	✓	✓	✗	✗	✗	✗	✗	✓

strategies for achieving efficient DNN execution [17, 55, 66, 67, 74], and several previous studies focus on preserving the precision of outliers, which is critical for maintaining model accuracy. In the context of accelerating convolutional neural networks, OLAcel [48] implements both 16-bit and 4-bit MAC units, with the 16-bit units handling outlier computations. DRQ [54] employs an algorithm to identify accuracy-sensitive regions in tensors and performs high-precision computation on these regions. While effective, these approaches rely on mixed-precision computation, which leads to increased hardware complexity and unaligned memory access.

Table 13 compares MX+ with existing outlier-aware work for LLMs. AWQ [34] and SqueezeLLM [28] are weight quantization methods that perform computation in high precision after dequantization. SmoothQuant (SMQ) [63] and QuaRot [3] reduce the magnitude of activation outliers through rescaling or rotation but show subpar accuracy at low bit widths because outliers are not completely addressed. OliVe [15] employs custom data formats, whereas MX+ is based on standard formats while providing higher accuracy. Tender [32] groups activation channels with similar dynamic ranges and achieves comparable accuracy to MXFP4, as discussed in the paper. LLM-FP4 [35] uses a custom floating-point format with channel-wise scales encoded into the exponent bias. We observe that LLM-FP4 performs worse than MXFP4 in our experiments.

10 Conclusion

Serving LLMs requires substantial compute and memory resources, and the MX data formats developed by leading industry companies are increasingly being adopted to mitigate these challenges. In this work, we investigate the implications of employing MX formats for LLM inference and identify that model performance significantly degrades under ultra-low precision due to quantization of activation tensors containing outliers. Building on the insight that the block absolute maximum element does not need to store its exponent in the element data type, we propose MX+, a non-intrusive extension to MX that repurposes the exponent field as an extended mantissa. Without adding complexity, MX+ substantially improves model performance across various precisions, with greater gains at lower bit widths. It also enables straightforward deployment in software integration scenarios with marginal overhead during inference, which can be virtually eliminated with hardware support.

Acknowledgments

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grants funded by the Korean government (MSIT) (IITP-2025-RS-2022-00156295, IITP-2025-RS-2023-00256081) and a research grant from Samsung Advanced Institute of Technology (SAIT). The Institute of Engineering Research at Seoul National University provided research facilities for this work. Jaewoong Sim is the corresponding author.

References

- [1] Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J Hewett, Mojtaba Javaheripour, Piero Kauffmann, et al. 2024. Phi-4 Technical Report. *arXiv preprint arXiv:2412.08905* (2024).
- [2] AMD. 2025. AMD Instinct MI355X GPUs. <https://www.amd.com/en/products/accelerators/instinct/mi350/mi355x.html>
- [3] Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian L. Croci, Bo Li, Pashmina Cameron, Martin Jaggi, Dan Alistarh, Torsten Hoefer, and James Hensman. 2024. QuaRot: Outlier-Free 4-Bit Inference in Rotated LLMs. In *Advances in Neural Information Processing Systems (NeurIPS)*.
- [4] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*.
- [5] Steve Dai, Ranga Venkatesan, Mark Ren, Brian Zimmer, William Dally, and Bruce Khailany. 2021. VS-Quant: Per-vector Scaled Quantization for Accurate Low-Precision Neural Network Inference. In *Proceedings of Machine Learning and Systems (MLSys)*.
- [6] Bita Darvish Rouhani, Daniel Lo, Ritchie Zhao, Ming Liu, Jeremy Fowers, Kalin Ovtcharov, Anna Vinogradsky, Sarah Massengill, Lita Yang, Ray Bittner, Alessandro Forin, Haishan Zhu, Tae-sik Na, Prerak Patel, Shuai Che, Lok Chand Koppaka, Xia Song, Subhjoy Som, Kaustav Das, Saurabh Tiwary, Steve Reinhardt, Sitaram Lanka, Eric Chung, and Doug Burger. 2020. Pushing the Limits of Narrow Precision Inferencing at Cloud Scale with Microsoft Floating Point. In *Advances in Neural Information Processing Systems (NeurIPS)*.
- [7] Bita Darvish Rouhani, Ritchie Zhao, Venmugil Elango, Rasoul Shafipour, Mathew Hall, Maral Mesmakhosroshahi, Ankit More, Levi Melnick, Maximilian Golub, Girish Varatkar, Lai Shao, Gaurav Kolhe, Dmitry Melts, Jasmine Klar, Renee L'Heureux, Matt Perry, Doug Burger, Eric Chung, Zhaoxia (Summer) Deng, Sam Naghshineh, Jongsoo Park, and Maxim Naumov. 2023. With Shared Microexponents, A Little Shifting Goes a Long Way. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*.
- [8] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. QLoRA: Efficient Finetuning of Quantized LLMs. In *Advances in Neural Information Processing Systems (NeurIPS)*.
- [9] Caiwen Ding, Siyu Liao, Yanzhi Wang, Zhe Li, Ning Liu, Youwei Zhuo, Chao Wang, Xuehai Qian, Yu Bai, Geng Yuan, Xiaolong Ma, Yipeng Zhang, Jian Tang, Qinru Qi, Xue Lin, and Ba Yuan. 2017. CirCNN: Accelerating and Compressing Deep Neural Networks Using Block-Circulant Weight Matrices. In *Proceedings of the 50th Annual International Symposium on Microarchitecture (MICRO)*.
- [10] Jesse Dodge, Maarten Sap, Ana Marasovic, William Agnew, Gabriel Ilharco, Dirk Groeneveld, Margaret Mitchell, and Matt Gardner. 2021. Documenting Large Webtext Corpora: A Case Study on the Colossal Clean Crawled Corpus. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.
- [11] Peiyang Dong, Lei Lu, Chao Wu, Cheng Lyu, Geng Yuan, Hao Tang, and Yanzhi Wang. 2023. Packqvit: Faster Sub-8-bit Vision Transformers via Full and Packed Quantization on the Mobile. In *Advances in Neural Information Processing Systems (NeurIPS)*.
- [12] Mario Drumond, Tao Lin, Martin Jaggi, and Babak Falsafi. 2018. Training DNNs with Hybrid Block Floating Point. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS)*.
- [13] Abhimanyu Dubey et al. 2024. The Llama 3 Herd of Models. *arXiv preprint arXiv:2407.21783* (2024).
- [14] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A Configurable Cloud-scale DNN processor for Real-time AI. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*.
- [15] Cong Guo, Jiaming Tang, Weiming Hu, Jingwen Leng, Chen Zhang, Fan Yang, Yunxin Liu, Minyi Guo, and Yuhao Zhu. 2023. OliVe: Accelerating Large Language Models via Hardware-friendly Outlier-Victim Pair Quantization. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*.
- [16] Cong Guo, Chen Zhang, Jingwen Leng, Zihan Liu, Fan Yang, Yunxin Liu, Minyi Guo, and Yuhao Zhu. 2022. ANT: Exploiting Adaptive Numerical Data Type for Low-bit Deep Neural Network Quantization. In *Proceedings of the 55th Annual International Symposium on Microarchitecture (MICRO)*.
- [17] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*.
- [18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.
- [19] Guyue Huang, Zhengyang Wang, Po-An Tsai, Chen Zhang, Yufei Ding, and Yuan Xie. 2023. RM-STC: Row-Merge Dataflow Inspired GPU Sparse Tensor Core for Energy-Efficient Sparse Acceleration. In *Proceedings of the 56th Annual International Symposium on Microarchitecture (MICRO)*.
- [20] Intel. 2023. Floating-Point Reference Sheet for Intel® Architecture. <https://www.intel.com/content/www/us/en/content-details/786447/floating-point-reference-sheet-for-intel-architecture.html>
- [21] Intel. 2024. Intel Unleashes Enterprise AI with Gaudi 3, AI Open Systems Strategy and New Customer Wins. <https://www.intel.com/news-events/press-releases/detail/1689/intel-unleashes-enterprise-ai-with-gaudi-3-ai-open-systems>
- [22] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. 2019. Dissecting the NVIDIA Turing T4 GPU via Microbenchmarking. *arXiv preprint arXiv:1903.07486* (2019).
- [23] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lamplé, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7B. *arXiv preprint arXiv:2310.06825* (2023).
- [24] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. 2023. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*.
- [25] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaeimmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penkunda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*.
- [26] Roman Kaplan. 2024. Intel Gaudi 3 AI Accelerator: Architected for Gen AI Training and Inference. In *2024 IEEE Hot Chips 36 Symposium (HCS)*.
- [27] Mahmoud Khairy, Zhenheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA)*.
- [28] Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiyu Li, Sheng Shen, Michael W. Mahoney, and Kurt Keutzer. 2024. SqueezeLLM: Dense-and-Sparse Quantization. In *Proceedings of the International Conference on Machine Learning (ICML)*.
- [29] Yoonsung Kim, Changhun Oh, Jinwoo Hwang, Wonung Kim, Seongryong Oh, Yubin Lee, Hardik Sharma, Amir Yazdanbakhsh, and Jongse Park. 2024. DA-CAPO: Accelerating Continuous Learning in Autonomous Systems for Video Analytics. In *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA)*.
- [30] Urs Köster, Tristan J. Webb, Xin Wang, Marcel Nassar, Arjun K. Bansal, William H. Constable, Oğuz H. Elibol, Scott Gray, Stewart Hall, Luke Hornof, Amir Khosrowshahi, Carey Kloss, Ruby J. Pai, and Naveen Rao. 2017. Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS)*.
- [31] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*.
- [32] Jungi Lee, Wonbeom Lee, and Jaewoong Sim. 2024. Tender: Accelerating Large Language Models via Tensor Decomposition and Runtime Requantization. In *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA)*.
- [33] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
- [34] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, and Song Han. 2023. AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration. In *Proceedings of Machine Learning and Systems (MLSys)*.
- [35] Shih-yang Liu, Zechun Liu, Xijie Huang, Pingcheng Dong, and Kwang-Ting Cheng. 2023. LLM-FP4: 4-Bit Floating-Point Quantized Transformers. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

- [36] Yun-Chen Lo, Tse-Kuang Lee, and Ren-Shuo Liu. 2023. Block and Subword-Scaling Floating-Point (BSFP): An Efficient Non-Uniform Quantization For Low Precision Inference. In *Proceedings of the International Conference on Learning Representations (ICLR)*.
- [37] Yun-Chen Lo and Ren-Shuo Liu. 2023. Bucket Getter: A Bucket-based Processing Engine for Low-bit Block Floating Point (BFP) DNNs. In *Proceedings of the 56th Annual International Symposium on Microarchitecture (MICRO)*.
- [38] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2017. Pointer Sentinel Mixture Models. In *Proceedings of the International Conference on Learning Representations (ICLR)*.
- [39] Microsoft. 2024. MX PyTorch Emulation Library. <https://github.com/microsoft/microxaling>
- [40] NVIDIA. 2022. NVIDIA RTX A6000 Graphics Card. <https://www.nvidia.com/en-us/design-visualization/rtx-a6000/>
- [41] NVIDIA. 2024. NVIDIA Blackwell Architecture Technical Brief. <https://resources.nvidia.com/en-us/blackwell-architecture>
- [42] NVIDIA. 2025. Block Scaling. <https://docs.nvidia.com/deeplearning/cudnn/frontend/latest/operations/BlockScaling.html>
- [43] NVIDIA. 2025. CUDA Binary Utilities. <https://docs.nvidia.com/cuda/cuda-binary-utilities/#nvdisasm>
- [44] NVIDIA. 2025. NVIDIA CUTLASS Documentation. https://docs.nvidia.com/cutlass/media/docs/cpp/blackwell_functionality.html#tile-size
- [45] NVIDIA. 2025. NVIDIA RTX Blackwell GPU Architecture. <https://images.nvidia.com/aem-dam/Solutions/geforce/blackwell/nvidia-rtx-blackwell-gpu-architecture.pdf>
- [46] NVIDIA. 2025. Parallel Thread Execution ISA Version 8.7. <https://docs.nvidia.com/cuda/parallel-thread-execution/#warp-level-matrix-fragment-mma-16864>
- [47] Open Compute Project 2023. OCP Microscaling Formats (MX) Specification Version 1.0. Open Compute Project.
- [48] Eunhyeok Park, Dongyoung Kim, and Sungjoo Yoo. 2018. Energy-Efficient Neural Network Accelerator Based on Outlier-Aware Low-Precision Computation. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*.
- [49] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.
- [50] Md Aamir Raihan, Negar Goli, and Tor M. Aamodt. 2019. Modeling Deep Learning Accelerator Enabled GPUs. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.
- [51] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. 2016. Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*.
- [52] Bita Darvish Rouhani, Ritchie Zhao, Ankit More, Mathew Hall, Alireza Khodamoradi, Sumner Deng, Dhruv Choudhary, Marius Cornea, Eric Dellinger, Kristof Denolf, Stosic Dusan, Venmugil Elango, Maximilian Golub, Alexander Heinecke, Phil James-Roxby, Dharmesh Jani, Gaurav Kolhe, Martin Langhammer, Ada Li, Levi Melnick, Maral Mesmakhosroshahi, Andres Rodriguez, Michael Schulte, Rasoul Shafipour, Lei Shao, Michael Siu, Pradeep Dubey, Paulius Mikkevicius, Maxim Naumov, Colin Verrilli, Ralph Wittig, Doug Burger, and Eric Chung. 2023. Microscaling Data Formats for Deep Learning. *arXiv preprint arXiv:2310.10537* (2023).
- [53] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. 2015. Imagenet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision* (2015).
- [54] Zhiqiang Song, Bangqi Fu, Feiyang Wu, Zhaoming Jiang, Li Jiang, Naifeng Jing, and Xiaoyao Liang. 2020. DRQ: Dynamic Region-based Quantization for Deep Neural Network Acceleration. In *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA)*.
- [55] Thierry Tambe, Coleman Hooper, Lillian Pentecost, Tianyu Jia, En-Yu Yang, Marco Donato, Victor Sanh, Paul Whatmough, Alexander M. Rush, David Brooks, and Gu-Yeon Wei. 2021. EdgeBERT: Sentence-Level Energy Optimizations for Latency-Aware Multi-Task NLP Inference. In *Proceedings of the 54th Annual International Symposium on Microarchitecture (MICRO)*.
- [56] Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. 2025. CUTLASS. <https://github.com/NVIDIA/cutlass>
- [57] Philipp Tillett, H. T. Kung, and David Cox. 2019. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*.
- [58] Hugo Touvron et al. 2023. Llama 2: Open Foundation and Fine-tuned Chat Models. *arXiv preprint arXiv:2307.09288* (2023).
- [59] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. 2021. Training data-efficient image transformers & distillation through attention. In *Proceedings of the International Conference on Machine Learning (ICML)*.
- [60] Swagath Venkataramani, Vijayalakshmi Srinivasan, Wei Wang, Sanchari Sen, Jintao Zhang, Ankur Agrawal, Monodeep Kar, Shubham Jain, Alberto Mannari, Hoang Tran, Yulong Li, Eri Ogawa, Kazuaki Ishizaki, Hiroshi Inoue, Marcel Schaaf, Mauricio Serrano, Jungwook Choi, Xiao Sun, Naigang Wang, Chia-Yu Chen, Allison Allain, James Bonomo, Nianzheng Cao, Robert Casatuta, Matthew Cohen, Bruce Fleischer, Michael Guillorn, Howard Haynie, Jinwook Jung, Ming Kang, Kyu-hyou Kim, Siyu Koswatta, Saekyu Lee, Martin Lutz, Silvia Mueller, Jinwook Oh, Ashish Ranjan, Zhibin Ren, Scot Rider, Kerstin Schelten, Michael Scheuermann, Joel Silberman, Jie Yang, Vidhi Zalani, Xin Zhang, Ching Zhou, Matt Ziegler, Vinay Shah, Moriyoshi Ohara, Pong-Fei Lu, Brian Curran, Sunil Shukla, Leland Chang, and Kailash Gopalakrishnan. 2021. RaPiD: An Accelerator for Ultra-low Precision Training and Inference. In *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA)*.
- [61] Lei Wang, Lingxiao Ma, Shijie Cao, Quanlu Zhang, Jilong Xue, Yining Shi, Ningxin Zheng, Ziming Miao, Fan Yang, Ting Cao, Yuqing Yang, and Mao Yang. 2024. Ladder: Enabling Efficient Low-Precision Deep Learning Computing through Hardware-aware Tensor Transformation. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
- [62] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierrick Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. HuggingFace's Transformers: State-of-the-art Natural Language Processing. *arXiv preprint arXiv:1910.03771* (2019).
- [63] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models. In *Proceedings of the International Conference on Machine Learning (ICML)*.
- [64] Qwen: An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Kepin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2025. Qwen2.5 Technical Report. *arXiv preprint arXiv:2412.15115* (2025).
- [65] Thomas Yeh, Max Sterner, Zerlina Lai, Brandon Chuang, and Alexander Ihler. 2022. Be Like Water: Adaptive Floating Point for Machine Learning. In *Proceedings of the International Conference on Machine Learning (ICML)*.
- [66] Jeffrey Yu, Kartik Prabhu, Yonatan Urman, Robert M. Radway, Eric Han, and Priyanka Raina. 2024. 8-bit Transformer Inference and Fine-tuning for Edge Accelerators. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
- [67] Ali Hadi Zadeh, Mostafa Mahmoud, Ameer Abdelhadi, and Andreas Moshovos. 2022. Mokey: Enabling Narrow Fixed-Point Inference for Out-of-the-Box Floating-Point Transformer Models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*.
- [68] Cheng Zhang, Jianyi Cheng, Ilia Shumailov, George Constantinides, and Yiren Zhao. 2023. Revisiting Block-based Quantisation: What is Important for Sub-8-bit LLM Inference?. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.
- [69] Haoyang Zhang, Yiru Zhou, Yuqi Xue, Yiqi Liu, and Jian Huang. 2023. G10: Enabling An Efficient Unified GPU Memory and Storage Architecture with Smart Tensor Migrations. In *Proceedings of the 56th Annual International Symposium on Microarchitecture (MICRO)*.
- [70] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuhui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punith Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. OPT: Open Pre-trained Transformer Language Models. *arXiv preprint arXiv:2205.01068* (2022).
- [71] Sai Qian Zhang, Bradley McDanel, and HT Kung. 2022. FAST: DNN Training Under Variable Precision Block Floating Point with Stochastic Rounding. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.
- [72] Ritchie Zhao, Yuwei Hu, Jordan Dotzel, Chris De Sa, and Zhirui Zhang. 2019. Improving Neural Network Quantization Without Retraining Using Outlier Channel Splitting. In *Proceedings of the International Conference on Machine Learning (ICML)*.
- [73] Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. 2024. Atom: Low-Bit Quantization for Efficient and Accurate LLM Serving. In *Proceedings of Machine Learning and Systems (MLSys)*.
- [74] Yongwei Zhao, Chang Liu, Zidong Du, Qi Guo, Xing Hu, Yimin Zhuang, Zhenxing Zhang, Xinkai Song, Wei Li, Xishan Zhang, Ling Li, Zhiwei Xu, and Tianshi Chen. 2021. Cambricon-Q: A Hybrid Architecture for Efficient Training. In *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA)*.