TMA-Adaptive FP8 Grouped GEMM: Eliminating Padding Requirements in Low-Precision Training and Inference on Hopper

Zhongling SU [1] **Rong FU** [1] **Weihan CAO** [1] **Jianfei GAO** [1] **Minxi JIN** [1] **Zhilin PEI** [1] **Hui WANG** [1]
[1] **Shanghai Artificial Intelligence Laboratory**, **Shanghai, China**

Abstract

Current FP8 grouped GEMM implementations require padding each group to a fixed alignment (e.g., 128), incurring memory and computational overhead. We propose *TMA-Adaptive FP8 Grouped GEMM*, which eliminates padding by dynamically adapting to variable group dimensions via (1) a TMA descriptor pool with $\log_2(block_M)$ preconfigured descriptors to handle all residual row cases through dynamic runtime selection and dual-phase load-store operations, achieving comprehensive coverage with minimal overhead, and (2) TMA-alignment-aware management to satisfy 16-byte global memory alignment and 128-byte shared memory alignment. Experiments demonstrate 1.7% to 20.4% speed up with up to 23.8% memory reduction compared to padding operation plus state-of-the-art FP8 grouped GEMM, while maintaining full numerical equivalence for valid data. The source code is publicly available at an anonymous repository: `https://github.com/sukoncon/TMA-Adaptive-FP8-Grouped-GEMM`.

1. Introduction and Related Work

To accelerate computation and reduce memory consumption, the exploration of low-precision computation has gained momentum. In 2018, (Micikevicius et al., 2018) pioneered the training of deep neural networks using half-precision floating-point numbers. With subsequent hardware advancements, lower precision techniques continued to evolve, as demonstrated by (Mitchell et al., 2023) which proposed SwitchBack - an int8 quantized linear layer for training. The stability and computational efficiency of 8-bit training/inference for large models were ultimately validated by (DeepSeek-AI, 2025).

Modern GPU architectures Hopper and beyond provide support for FP8 tensor core. In fully utilize GPU capabilities, various optimization techniques have been developed (Zhao et al., 2025; Shah et al., 2024), including warp specialization, Tensor Memory Accelerator (TMA) (NVIDIA, 2022-03-22), and warp group matrix multiply-accumulate, collectively maximizing memory bandwidth utilization and arithmetic intensity for foundation model workloads.

These low-precision techniques find particularly important applications in specialized architectures. Grouped General Matrix Multiplication (Grouped GEMM) operations play a critical role in Mixture-of-Experts (MoE) architectures, where dynamic group sizes arise from variable sequence lengths selected via top-k routing.

In Grouped GEMM computations, padding is essential due to hardware constraints when handling variable dimensions (e.g., dynamically routed sequence lengths in MOE). The necessity arises from two fundamental architectural limitations: First, the static configuration of TMA descriptors during host initialization **prevents dynamic adjustment to varying row dimensions across different groups**. Second, the Hopper architecture's TMA imposes strict alignment requirements - **16-byte for global memory and 128-byte for shared memory** for multi-dimensional bulk tensor asynchronous copies (NVIDIA Corporation, 2025).

In this work, we present TMA-Adaptive FP8 Grouped GEMM, a hardware-compliant optimization framework for FP8 matrix multiplication that eliminates padding overhead while strictly satisfying TMA alignment constraints. Our solution introduces two key innovations:

First, a TMA descriptor pool that predefines descriptors for all possible residual rows (the remaining rows when the group row count is divided by $\texttt{block}_\texttt{M}$), followed by dynamic runtime descriptor selection and dual-phase load-store operations. This approach **achieves coverage of all possible residual row cases with only** $\log_2(block_M)$ **preconfigured descriptors**.

Second, we propose an TMA-alignment-aware memory access scheme that guarantees compliance through two mechanisms: over-fetching to maintain 16-byte global memory alignment, and constraining $\texttt{block}_\texttt{N}$ to multiples of 64 elements to ensure 128-byte shared memory alignment boundaries.

In the following sections, we first present our methodology in Section 2, followed by experimental results and analysis in Section 3. Additional details are provided in the Appendix.

2. Methodology

We present an optimization framework (Figure 1) featuring two key innovations. In this section, we first introduce the fundamental concepts of grouped GEMM, then provide detailed descriptions of the two major innovations: dynamic descriptor selection with two-phase load-store in Section 2.2 (corresponding to the yellow blocks in Figure 1) and TMA-Alignment-Aware memory management in Section 2.3 (addressing the green blocks and $block_N$ configuration in Figure 1).

2.1. Notation and Preliminaries for Grouped GEMM

Table 1. Key notation for grouped matrix multiplication

Symbol	Description	Dimensions
g	Group index	Scalar
M^g	Variable row dimension of group g	Scalar
N	Output column dimension	Scalar
K	Hidden dimension	Scalar
$\mathbf{A}^g$	Left matrix for group g	$\mathbb{R}^{M^g \times K}$
$\mathbf{B}^g$	Right matrix for group g	$\mathbb{R}^{K \times N}$
$\mathbf{C}^g$	Output matrix for group g	$\mathbb{R}^{M^g \times N}$
$\mathbf{S}_A^g$	Per-tile scale for $\mathbf{A}^g$	$\mathbb{R}^{M^g \times \lceil K/128 \rceil}$
$\mathbf{S}_B^g$	Per-block scale for $\mathbf{B}^g$	$\mathbb{R}^{\lceil K/128 \rceil \times \lceil N/128 \rceil}$

Table 1 summarizes the key notation used in Grouped GEMM with specialized quantization schemes. Following (DeepSeek-AI, 2025), we employ 1x128 tiled scaling for A and 128x128 blocked scaling for B.

In the context of mixture-of-experts (MoE) architectures for LLMs, the dimension M_g denotes the *dynamically routed sequence length* assigned to expert group g. The static parameters K (hidden dimension) and N (output dimension) remain constant across all groups.

The CUDA kernel configurations $block_M$, $block_N$, and $block_K$ represent fixed tiling sizes for computational optimization.

Among all operands involved in Grouped GEMM, particular attention must be devoted to memory alignment when **loading S_A^g from global memory** and **loading C^g from shared memory**. A detailed analysis of memory alignment is provided in Appendix A.

2.2. Dynamic Descriptor Selection with Two-Phase Load-Store

Our primary technical contribution is a hardware-aware TMA descriptor design that combines static configuration compliance with dynamic runtime adaptation, featuring a novel two-phase load-store mechanism for **residual blocks of output matrix**.

The methodology comprises three stages:

Descriptor Pool Predefinition: During kernel initialization, we construct a descriptor pool (see static configuration of Figure 1) derived from the block dimension $block_M$:

$$\mathcal{D}_{pool} = \left\{ [2^i, block_N] \;\middle|\; i \in \mathbb{N},\ 0 \le i \le \lfloor \log_2(block_M) \rfloor \right\} \tag{1}$$

Runtime Selection: At runtime, for each group g with residual rows $res^g = M^g \mod block_M$, we determine the optimal descriptor:

$$\mathcal{D}_{opt}^g = [2^{\lfloor \log_2(res^g) \rfloor}, block_N] \tag{2}$$

Two-Phase Load-Store: The mechanism employs two coordinated TMA operations: (a) Shared memory rows $\left[0, 2^{\lfloor \log_2(res^g) \rfloor} - 1\right]$ to global memory rows $\left[M^g - res^g, M^g - res^g + 2^{\lfloor \log_2(res^g) \rfloor} - 1\right]$, and (b) Shared memory rows $\left[res^g - 2^{\lfloor \log_2(res^g) \rfloor}, res^g - 1\right]$ to global memory rows $\left[M^g - 2^{\lfloor \log_2(res^g) \rfloor}, M^g - 1\right]$

This design implements a safe overlapping write strategy by intentionally writing only a small portion of identical data in the overlapping memory regions, ensuring that: (1) preservation of all valid results, (2) strict boundary compliance, (3) logarithmic descriptor scalability ($\lfloor \log_2(block_M) \rfloor$ TMAs for $block_M$ residuals), and (4) two TMA operations per residual block regardless of its size. See Appendix B for details.

2.3. TMA-Alignment-Aware Memory Management

Section 2.2 demonstrates that the second TMA load operation starts at row $res^g - 2^{\lfloor \log_2(res^g) \rfloor}$, potentially causing misalignment issues. To guarantee 128-byte alignment regardless of the starting row position, we enforce $block_N$ to be a multiple of 64.

The per-row offset of $4\lceil K/128 \rceil$ bytes in global memory for S_A^g may cause misalignment issues. We address this misalignment through boundary-aligned prefetching, where the starting address is adjusted to meet 16-byte alignment requirements. The number of prefetched rows from previous data row_{prev}^g is determined by:

$$\min \left\{ \left[Addr_{SA}^g - 4 row_{\text{prev}}^g \left\lceil \frac{K}{128} \right\rceil \right] \bmod 16 = 0 \right\} \tag{3}$$

where $Addr_{SA}^g$ denotes the base address of vector group S_A^g and the coefficient 4 reflects `float` datatype storage. Subsequent rows $row_{\text{next}}^g = 16 + block_M - row_{\text{prev}}^g$ complete the prefetching window, with TMA descriptors configured

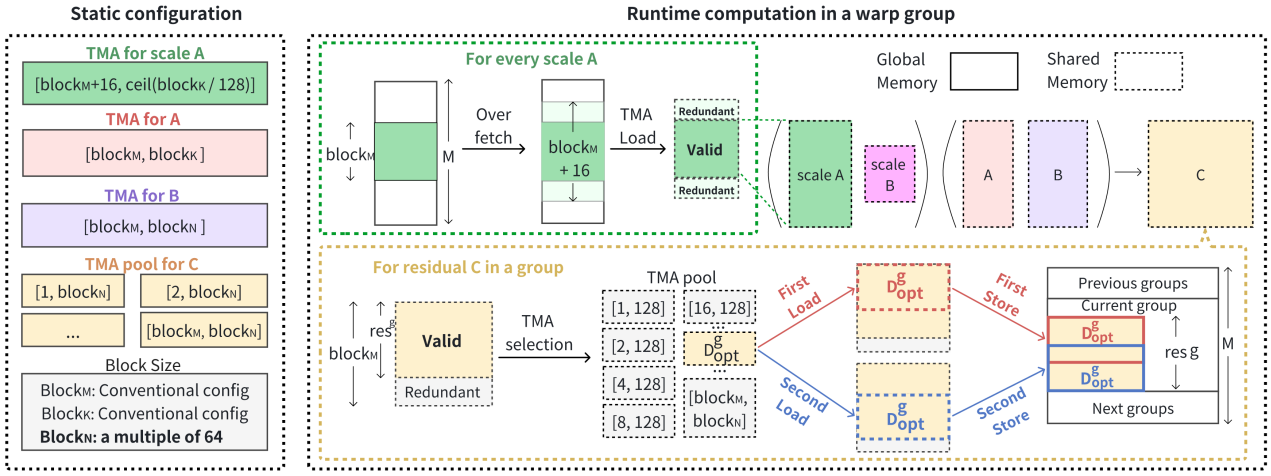

Figure 1. The framework of TMA-Adaptive FP8 Grouped GEMM. The left panel shows the static configuration featuring our proposed TMA pool for C and block size constraint for N. The right panel illustrates the runtime computation within a warp group. Key innovations are highlighted: (1) global memory prefetch for scaled blocks of matrix A (green), and (2) Dynamic Descriptor Selection with Two-Phase Load-Store for residual elements in matrix C (yellow).

for $[16 + block_M, \lceil K/128 \rceil]$ dimensions. Only the central $block_M$ rows (valid green block in Figure 1) participate in computation.

3. Experiment

The experimental evaluation was performed on an NVIDIA H800 GPU accelerated computing platform using PyTorch 2.6.0 and CUDA 12.6. Our baseline implementation integrates explicit input padding with DeepGEMM, which currently represents the state-of-the-art high-performance FP8 GEMM implementation, against which we compare our optimized approach across three critical metrics: computational acceleration, numerical equivalence, and memory efficiency.

3.1. Experimental Setup

The parameter space was designed to reflect practical configurations in modern large language model architectures, with tensor dimensions spanning $N, K \in \{3072, 4096, 5120, 6144, 7168, 8192\}$, group counts in $\{4, 8, 16, 32\}$, and sequence lengths after top-k routing $M \in \{8192, 16384, 32768, 65536\}$ with each group dimension M^g being randomly generated (see Appendix C.1).

We implemented the padding operation for matrices A and S_A using a custom high-performance kernel written in Triton. This kernel achieves approximately 2000 GB/s DRAM bandwidth (2173 GB/s maximum), ensuring the validity of our comparative results against the baseline.

3.2. Performance Analysis

Figure 2 demonstrates computational acceleration ratios and memory savings across configurations.

The acceleration (Figure 2a) ranges from 1.7% to 20.4%, exhibiting a weak positive correlation with sequence length $M (r = 0.09)$ and group count $(r = 0.096)$, while showing a strong negative correlation with parameter $N (r = -0.899)$. The observed acceleration stems from eliminating padding-induced memory transactions, while the dominant anti-correlation with N reflects linear scaling of residual write-back costs.

Memory savings ((b) in Figure 2) demonstrate strong inverse correlation with sequence length M accompanied by near-linear scaling with group count, achieving maximum observed savings of 23.8% when operating at $M = 8192$ with 32 groups. This fundamental relationship stems from geometric padding characteristics in grouped tensor operations - smaller M values combined with higher group counts yeild more padding data.

Numerical equivalence was verified through comparison of these test cases after removing zero-padded elements from baseline outputs, demonstrating bitwise identical results between our method and baseline implementations. This zero-error guarantee stems from our two-phase load and store: the final write-back operation strictly preserves only the valid data region.

4. Conclusion

This paper presents TMA-Adaptive FP8 Grouped GEMM, a hardware-aligned solution that eliminates the requirement

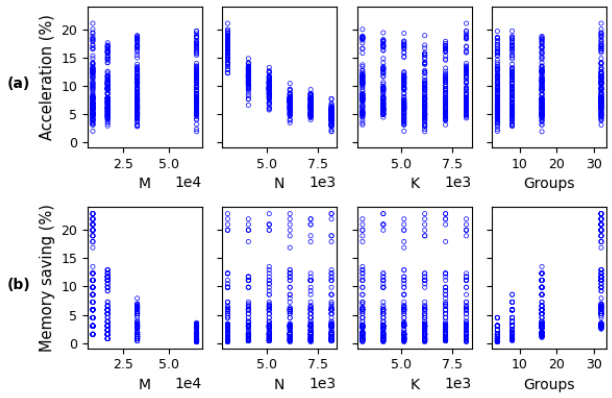

Figure 2. Performance comparison between optimized and baseline implementations showing (a) computational acceleration ratios and (b) relative memory savings across varying M, N, K and group counts.

for padding groups to fixed alignment multiples (e.g., 128 elements) in low-precision Grouped GEMM.

By resolving key hardware-alignment conflicts in variable-length scenarios—specifically, static TMA descriptor limitations and memory boundary misalignments—our approach simultaneously enhances memory efficiency and computational throughput without degrading numerical accuracy.

Although our method constrains $block_N$ to multiples of 64 (e.g., 64, 128, 192), these values represent optimal or near-optimal configurations for the N dimension in practice. Experimental validation demonstrates significant improvements, including up to 23.8% reduction in memory allocation overhead and 1.7–20.4% end-to-end speedup compared to state-of-the-art padding implementations, while numerical verification confirms bitwise equivalence, preserving accuracy.

By fundamentally eliminating padding operations, our solution establishes an innovative paradigm for Grouped GEMM, delivering superior computational efficiency with reduced memory footprint and seamless plug-and-play compatibility for dynamic routing without kernel modifications beyond Grouped GEMM.

Crucially, this work directly enhances Mixture of Experts (MoE) Large Language Models: it accelerates inference and training by minimizing latency and memory overhead during dynamic expert routing, thereby facilitating scalable high-performance MoE LLM deployments.

Acknowledgements

Project supported by Shanghai Municipal Science and Technology Major Project.

References

DeepSeek-AI. Deepseek-v3 technical report, 2025. URL https://arxiv.org/abs/2412.19437.

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., and Wu, H. Mixed precision training, 2018. URL https://arxiv.org/abs/1710.03740.

Mitchell, W., Tim, D., Luke, Z., Ari, M., Ali, F., and Ludwig, S. Stable and low-precision training for large-scale vision-language models. In Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., and Levine, S. (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 10271–10298. Curran Associates, Inc., 2023.

NVIDIA. Nvidia hopper architecture in-depth. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/, 2022-03-22. Accessed: 2025-06-18.

NVIDIA Corporation. Cuda c++ programming guide, 2025. URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#table-alignment-multi-dim-tma.

Shah, J., Bikshandi, G., Zhang, Y., Thakkar, V., Ramani, P., and Dao, T. Flashattention-3: Fast and accurate attention with asynchrony and low-precision, 2024. URL https://arxiv.org/abs/2407.08608.

Zhao, C., Zhao, L., Li, J., and Xu, Z. Deepgemm: clean and efficient fp8 gemm kernels with fine-grained scaling. https://github.com/deepseek-ai/DeepGEMM, 2025.

A. Alignment Analysis for Grouped GEMM Operands

Alignment analysis for A^g: The tensor A^g participates in Tensor Memory Access (TMA) operations for both global memory loads and shared memory stores. In global memory, the per-row offset of A^g is K bytes, where K denotes the hidden dimension size in Mixture-of-Experts (MoE) architectures. The common practice of setting $Kmod16 = 0$ in modern LLM configurations (Deepseek, Mixtral, Llama) inherently satisfies the 16-byte global memory alignment requirement.

For shared memory access, we enforce 128-byte alignment during kernel initialization by allocating memory starting from 128-byte aligned addresses. This design makes the alignment condition independent of either the residual row size or $block_K$ parameter.

Alignment analysis for B^g: The alignment properties of B^g follow similar principles as A^g (detailed analysis omitted for brevity).

Alignment analysis for S_A^g: The scale tensor S_A^g participates in TMA operations involving both global memory loading and shared memory storing. In global memory, the per-row offset is $4\lceil K/128 \rceil$ bytes, which may potentially violate alignment requirements. To address this challenge, we introduce an over-fetching technique described in Section 2.3. To ensure proper alignment in shared memory, we predefine a 128-byte aligned starting address during kernel configuration.

Alignment analysis for S_B^g: The tensor S_B^g does not require TMA load operations since each warp group matrix multiply operation typically needs only one or two elements for scaling.

Alignment analysis for C^g: The output tensor C^g participates in TMA operations for both shared memory loads and global memory stores. In shared memory, the per-row offset is $2 \cdot block_N$ bytes (where the coefficient 2 accounts for half-precision data format). As shown in Section 2.2, the second phase of TMA begins at row $res^g - 2^{\lfloor \log_2(res^g) \rfloor}$, which may introduce misalignment. To address this issue, we constrain $block_N$ to be a multiple of 64, ensuring 128-byte alignment regardless of the starting row. For global memory access, the per-row offset is $2N$ bytes. The common practice of setting $2Nmod16 = 0$ in modern LLM architectures (Deepseek, Mixtral, Llama) inherently satisfies the 16-byte alignment requirement.

B. Example of TMA Runtime Selection and Two-Phase Load-Store

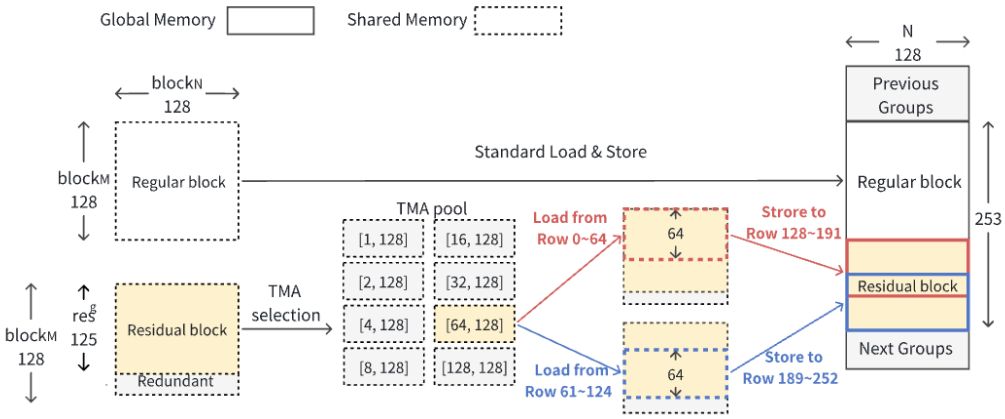

Figure 3. An illustration of TMA runtime selection and the two-phase load-store operation. The configuration uses $M^g = 381$ and $N = 128$ for demonstration purposes.

Figure 3 demonstrates our method using group g with output matrix dimensions $[M^g, N] = [253, 128]$. With $block\_M = 128$, the first 128 rows (white region) use standard TMA operations with descriptor $[128, 128]$, while the residual $res^g = 125$ rows (yellow) require our two-phase approach.

Following Section 2.2, we select the optimal descriptor $\mathcal{D}_{opt}^g = [64, 128]$ for the residual portion. The two-phase operation proceeds as:

1) **First Load-Store**: Transfers rows 0-63 from shared memory to global rows 128-191 (red blocks).

2) **Second Load-Store**: Loads rows 61-124 from shared memory to global rows 189-252 (blue blocks). The 3-row overlap

ensures complete coverage while preventing out-of-bounds writes.

This strategy guarantees that:

- All 125 residual rows are properly stored
- No corruption occurs in adjacent memory regions
- The hardware's static descriptor requirements are maintained

C. Details for experiment

C.1. M^g Generation

To generate the group dimension M^g, we employ a randomized algorithm that produces a list of groups G with elements summing to M. The generation process consists of the following steps:

1. Initialize a zero vector $\mathbf{v}$ of length G
2. For each element v_i in $\mathbf{v}$, assign a random integer value uniformly distributed between 0 and $2\lfloor M/G \rfloor$
3. Compute a scaling factor $\alpha = M/\sum_{i=1}^{G} v_i$ and apply it to each element: $v_i \leftarrow \lfloor \alpha v_i \rfloor$
4. Adjust the last element v_G to compensate for any residual difference: $v_G \leftarrow v_G + (M - \sum_{i=1}^{G} v_i)$

This approach ensures that the generated group dimensions maintain the desired total sum M while introducing controlled randomness in the distribution across groups. The scaling operation preserves the relative proportions of the initial random assignments, and the final adjustment guarantees exact sum preservation.

C.2. Coefficient Matrices

Table 2. Coefficient matrix of M, N, K, groups and acceleration

	M	N	K	groups	acceleration (%)
M	1.00	4.44×10^{-16}	7.75×10^{-17}	-3.43×10^{-18}	0.0959
N	4.44×10^{-16}	1.00	-2.50×10^{-16}	-4.85×10^{-17}	-0.8991
K	7.75×10^{-17}	-2.50×10^{-16}	1.00	-2.90×10^{-17}	-0.0274
groups	-3.43×10^{-18}	-4.85×10^{-17}	-2.90×10^{-17}	1.00	0.1036
acceleration (%)	0.0959	-0.8991	-0.0274	0.1036	1.00

Table 3. Coefficient matrix of M, N, K, groups and memory saving

	M	N	K	groups	memory saving (%)
M	1.00	4.44×10^{-16}	7.75×10^{-17}	-3.43×10^{-18}	-0.5460
N	4.44×10^{-16}	1.00	-2.50×10^{-16}	-4.85×10^{-17}	0.0051
K	7.75×10^{-17}	-2.50×10^{-16}	1.00	-2.90×10^{-17}	0.0036
groups	-3.43×10^{-18}	-4.85×10^{-17}	-2.90×10^{-17}	1.00	0.6356
memory saving (%)	-0.5460	0.0051	0.0036	0.6356	1.00