KABUMI: Communication-Avoiding Batched GEMM on GPUs

HEMENG WANG, SSSLab, Dept. of CST, China University of Petroleum-Beijing, Changping, China
WEIFENG LIU, SSSLab, Dept. of CST, China University of Petroleum-Beijing, Changping, China

Batched general matrix multiplication (GEMM) operations in latency-sensitive inference workloads often expose limited cooperative thread array (CTA)-level parallelism within each GEMM, making efficient GPU utilization challenging. Independent GEMMs can run concurrently, but this creates a tradeoff: each GEMM should expose more CTA-level parallelism when the batch is small and limit its CTA count when the batch already provides enough work to occupy the GPU. This paper presents KABUMI, a communication-avoiding (CA) batched GEMM framework for GPUs. KABUMI treats streaming multiprocessor (SM) participation as a budgeted resource rather than a byproduct of fixed tiling. For each single GEMM, KABUMI selects a CA algorithm according to the reduction length: small-K GEMMs use 1D CA output-space expansion, where output-tile candidates partition the wide column dimension, retain the tiny **A** operand near its owning CTA, stream the corresponding columns of **B**, and directly write disjoint **C** tiles; large-K GEMMs use 3D CA reduction-slice expansion to turn the long reduction dimension into schedulable partial CTAs and combine partial outputs. KABUMI's batch-aware scheduler then maps the current batch size to a per-GEMM CTA-wave budget, exposing more intra-GEMM parallelism for underfilled batches and capping expansion when batch-level concurrency can already occupy the GPU.

We implement KABUMI in CUDA and evaluate it on NVIDIA RTX 5090 and B200 GPUs. Across 1,440 batched-GEMM cases on RTX 5090, KABUMI achieves 1.60×, 2.45×, 2.64×, and 7.00× geomean speedup over cuBLAS batched GEMM, CUTLASS batched GEMM, CUTLASS grouped GEMM, and MAGMA batched GEMM, respectively. Across the same 1,440 batched-GEMM cases on B200, the corresponding geomean speedups are 1.32×, 1.98×, 2.09×, and 5.03×. On model-derived grouped GEMMs, KABUMI reaches 1.30×–1.56× geomean speedup over the evaluated BF16 library paths and 1.11× over FlashInfer in FP8.

CCS Concepts: • **Computer systems organization → Parallel architectures**; • **Computing methodologies → Parallel computing methodologies**; • **Mathematics of computing** → *Mathematical software performance*.

Additional Key Words and Phrases: Communication-Avoiding, GEMM, batched GEMM, GPU, LLM inference, MoE, Tensor Cores

1 Introduction

General matrix multiplication (GEMM) is one of the central kernels in numerical linear algebra and GPU computing [17, 23, 46, 59, 61, 65]. Conventional GPU GEMM implementations are often optimized for large, regular matrix products: they choose tiles at the cooperative thread array (CTA), warp, and instruction levels, stage operands through shared memory and registers, maximize data reuse, and keep Tensor Cores busy [12, 35, 37, 39, 56]. Libraries and kernel frameworks such as cuBLAS, CUTLASS, Triton, and DeepGEMM use these optimization techniques [14, 47, 48, 57]. It is highly effective when a GEMM has high arithmetic intensity and enough output tiles to fill the GPU, but it is less well matched to small and skewed matrix products that appear in real world scenarios.

Just Accepted

New Paper, Not an Extension of a Conference Paper.

Authors' Contact Information: Hemeng Wang, SSSLab, Dept. of CST, China University of Petroleum-Beijing, Changping, Beijing, China; e-mail: hemeng.wang@student.cup.edu.cn; Weifeng Liu, SSSLab, Dept. of CST, China University of Petroleum-Beijing, Changping, Beijing, China; e-mail: weifeng.liu@cup.edu.cn.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 1544-3973/2026/11-ART
https://doi.org/10.1145/3860333

Large language model (LLM) decoding is a representative source of such shapes [27, 60]. On one GPU after model sharding, a decoder projection can be written as a GEMM $\mathbf{C} = \mathbf{AB}$, where $\mathbf{A} \in \mathbb{R}^{M \times K}$ is the activation matrix, $\mathbf{B} \in \mathbb{R}^{K \times N}$ is the weight shard, and $\mathbf{C} \in \mathbb{R}^{M \times N}$ is the output. In latency-sensitive decoding, the token dimension M can be one or a few tokens, while N and K are determined by model widths and tensor-parallel sharding [45, 54]. These decoder projections therefore expose a shape problem: a single tiny-M GEMM may have too little output work to occupy the GPU, and its reduction dimension can range from short to long, leading to distinct small-K and large-K regimes.

Mixture-of-Experts (MoE) layers [15, 20, 32, 36, 53] turn this shape problem into a batched or grouped GEMM resource problem. After routing, each active expert receives a GEMM over its assigned tokens. These expert GEMMs are independent, often tiny in M, and irregular across experts; systems commonly execute them through grouped or batched GEMM interfaces to reduce launch overhead and expose inter-problem parallelism [21, 25, 30, 38, 52]. The resulting workload is no longer only a small-GEMM problem. It is a collection of small and skewed GEMMs sharing the same streaming multiprocessor (SM) pool.

For these shapes, optimizing data reuse and computational efficiency within each CTA alone is insufficient. The native $M \times N$ output tile space can expose too few CTAs, short reductions can poorly amortize deep pipelines, and maximizing reuse inside large CTAs can leave the memory system underdriven [49, 66, 70]. These limitations motivate a decomposition that exposes more CTA-level parallelism while preserving data reuse and limiting additional communication. Communication-avoiding (CA) algorithms provide a starting point for this design. For example, KAMI [62] applies CA algorithms to improve the performance of small GEMMs on a single GPU. Decoding-oriented batched GEMM introduces two additional considerations. The useful expansion axis depends on the local reduction length, and batched execution adds shared-SM pressure [1, 6, 19, 24]: when the batch is small, the selected CA algorithm should expose more intra-GEMM parallelism; when the batch is large, each GEMM should limit its SM footprint so that more independent GEMMs can execute concurrently. No fixed per-GEMM CA expansion policy fits both regimes.

We address this problem with KABUMI, a CA batched GEMM framework for GPUs. KABUMI treats SM participation as a budgeted resource. It first builds a CA expansion space for each GEMM shape, where each candidate exposes a different amount of CTA-level work and has a different traffic and synchronization cost. For small-K shapes, KABUMI uses a 1D CA output-space expansion: because the reduction is short and M is tiny, candidates partition the wide output dimension, cache or broadcast the tiny $\mathbf{A}$ operand, stream columns of $\mathbf{B}$, and directly write disjoint $\mathbf{C}$ tiles without partial accumulation. For large-K shapes, KABUMI uses 3D CA reduction-slice expansion [16, 29] to turn the long reduction dimension into additional schedulable partial CTAs and then combines partial outputs. KABUMI then applies a batch-aware budgeting policy that maps the current batch size to a per-GEMM SM budget, exposing more intra-GEMM parallelism when the batch underfills the GPU and capping expansion when inter-GEMM concurrency is sufficient. KABUMI therefore jointly selects a single-GEMM CA algorithm and a per-GEMM SM budget under shared-SM pressure.

This paper makes the following contributions:

- We characterize decoding-oriented tiny-M GEMMs as a coupled bandwidth, CTA underuse, and batched SM-sharing problem.
- We introduce KABUMI's single-GEMM CA expansion kernels: 1D CA output-space expansion for short reductions and 3D CA reduction-slice expansion for long reductions.
- We design batch-aware SM budgeting, which gives underfilled batches more intra-GEMM parallelism and caps each GEMM's SM footprint once batch concurrency is sufficient.
- We implement KABUMI in CUDA and evaluate batched GEMM and grouped GEMM on RTX 5090, and batched GEMM on B200, comparing with cuBLAS, CUTLASS batched, CUTLASS grouped, and MAGMA.

Just Accepted

2 Background

2.1 GPU Architecture

Modern GPUs achieve high throughput by distributing CTAs across many SMs. Each SM contains warp schedulers, a register file, compute units such as CUDA Cores and Tensor Cores, load/store units, and software-managed shared memory [12, 43]. These resources form a hierarchical machine: work is issued at CTA and warp granularity, arithmetic is performed inside SMs, and data moves through registers, shared memory, caches, and HBM. Figure 1 summarizes this view.

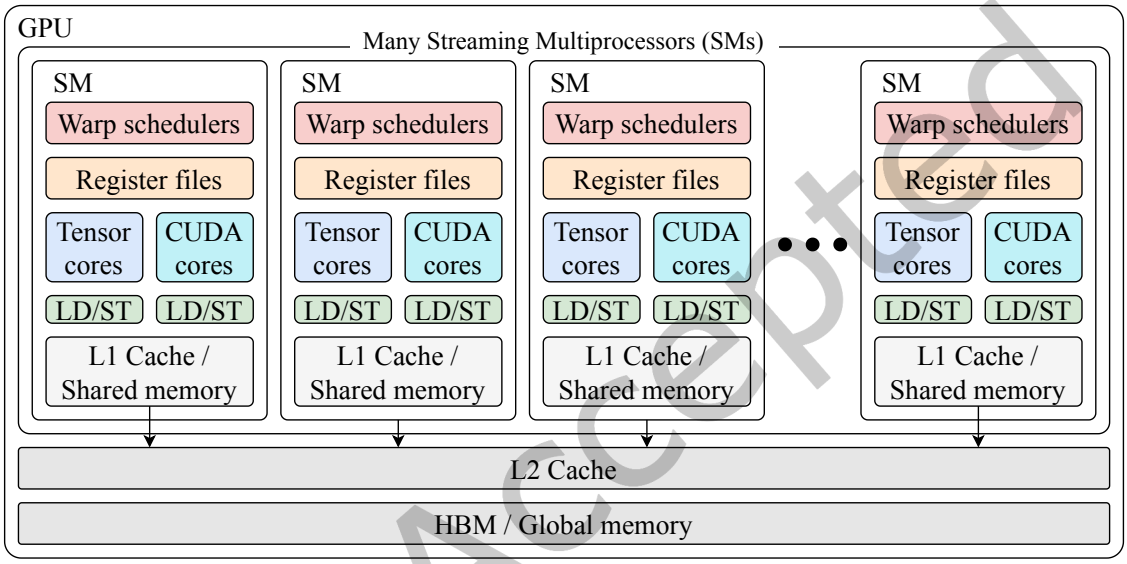


Fig. 1. A simplified GPU architecture view. Efficient kernels must coordinate the compute path inside each SM with the memory path that feeds data through caches and global memory.

This organization creates two relevant limits. Compute-bound kernels are limited by arithmetic issue rate, especially specialized matrix instructions. Memory-bound kernels are limited by load/store issue rate and by the bandwidth of the L2/HBM subsystem. In both cases, high performance requires enough useful CTAs to involve the SM pool, enough independent memory requests to drive the memory system, and efficient use of registers, shared memory, and caches. These constraints make performance sensitive to how much parallel work a kernel exposes and how that work is budgeted across SMs.

2.2 GPU GEMM

GEMM computes $\mathbf{C} = \mathbf{AB}$, where an $M \times K$ matrix $\mathbf{A}$ is multiplied by a $K \times N$ matrix $\mathbf{B}$ to produce an $M \times N$ matrix $\mathbf{C}$. GPU implementations usually map tiles of $\mathbf{C}$ to CTAs and then decompose each CTA tile into warp- and instruction-level tiles. The corresponding panels of $\mathbf{A}$ and $\mathbf{B}$ are staged through cache or shared memory, partial sums are accumulated in registers, and an epilogue writes the result.

We use arithmetic intensity in the roofline sense: floating-point operations per byte moved from the relevant memory level [66]. For a GEMM with element size e bytes, we estimate arithmetic intensity from the input and

Just Accepted

output matrix sizes as $I(M, N, K) = 2MNK/[e(MK + KN + MN)]$. This estimate captures how matrix shape affects arithmetic intensity but does not model cache behavior.

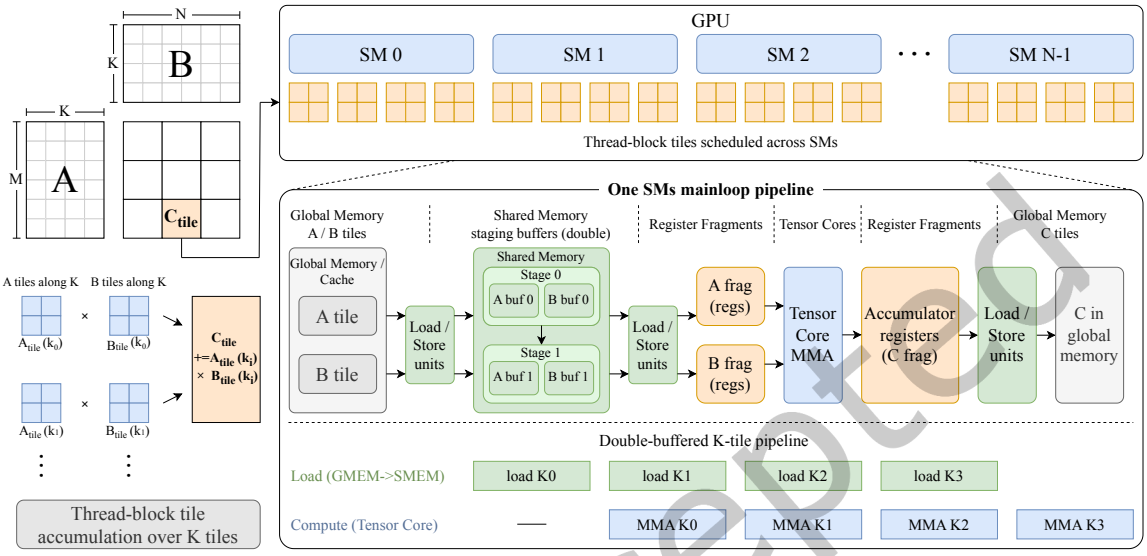


Fig. 2. CUTLASS-style hierarchical GEMM decomposition. Large GEMMs are mapped through CTA, warp, and instruction-level tiles so that data loaded from global memory is reused through shared memory and registers before the result is stored.

This hierarchical decomposition is the dominant optimization strategy in production GPU GEMM libraries and template frameworks. For large, regular GEMMs with sufficient data reuse and CTA-level parallelism, performance is typically determined by Tensor Core utilization and pipeline efficiency. The assumptions weaken for tiny-M or irregular shapes: small M or N can leave the output tile space smaller than the SM pool, short K gives each output element little work to amortize memory traffic, and highly uneven dimensions can create load imbalance or wasted work [10, 70].

At the algorithmic level, CA algorithms classify GEMM by which dimensions of the (M, N, K) iteration space are partitioned. A 1D CA algorithm slices one dimension, so processors own computation planes and communicate mainly one operand. A 2D CA algorithm, such as SUMMA [58], slices two dimensions into pencils and broadcasts operands along processor-grid rows and columns. A 3D CA algorithm [16] slices all three dimensions: replicated inputs feed layer-local 2D computations, followed by reduction of partial $\mathbf{C}$ outputs. Fractional variants such as 1.5D and 2.5D interpolate between these endpoints by partially replicating operands to reduce communication at extra storage or reduction cost [55].

2.3 Batched GEMM

Batched GEMM evaluates independent matrix multiplications $\mathbf{C}_p = \mathbf{A}_p\mathbf{B}_p$ for $p = 0, \ldots, P-1$, where P is the batch size [19]. Strided batched GEMM uses a regular layout for same-size instances; pointer-array and grouped GEMM support arbitrary addresses or varying dimensions.

Common batched GEMM optimizations amortize launch overhead and improve performance on small matrices through compact tiling, register blocking, layout-aware access, code generation, or autotuning [1, 6, 24, 26, 39].

Just Accepted

They improve batched execution, but usually do not make the per-GEMM SM footprint an explicit batch-dependent decision.

Decoding workloads can produce small GEMMs that share GPU resources during batched execution [11]. A token denotes one element of the input or generated sequence, and each autoregressive decoding step produces the next token for every active sequence. Latency-sensitive decoding prioritizes next-token latency. Dense projections have tiny-M because M is the active-token count, the number of token rows processed by one local projection, often one or a few in latency-sensitive decoding. Model sharding partitions model parameters across GPUs; tensor-parallel sharding determines the local N and K of each projection. MoE routing turns this into many expert GEMMs over assigned tokens; an expert is active when it is assigned at least one token. Thus batch size P, single-GEMM parallelism, and per-GEMM SM footprint jointly determine SM allocation; Section 3 develops this tradeoff for decoding-oriented tiny-M batched GEMMs.

3 Motivation

3.1 Shape Skew and Arithmetic Intensity

LLM decoding often produces GEMMs with highly skewed dimensions. In dense decoder projections, M is the number of active tokens in a projection call, often one or a few under latency-sensitive decoding and variable under continuous batching. MoE routing makes the shape issue batched and irregular: each active expert receives a GEMM over its assigned tokens, so expert GEMMs can have very small and uneven M. These GEMMs combine small M with N and K inherited from model width and sharding, and can have low arithmetic intensity even when the full model is large.

For a fixed element size, the estimated data transfer volume is proportional to $MK + KN + MN$. For fixed MNK (equivalently, fixed GEMM FLOPs up to a constant factor), the arithmetic mean–geometric mean inequality gives

$$MK + KN + MN \geq 3(MNK)^{2/3},$$

with equality only when $M = N = K$. Thus, among shapes with the same FLOPs, square GEMMs have the highest ideal arithmetic intensity, while highly skewed GEMMs require more global memory traffic. When the total computation is not large enough to amortize this traffic, the kernel can become memory-bound even though GEMM is usually treated as compute intensive.

Figure 3(a) isolates this shape effect by keeping $MNK = 2^{30}$ while moving from a square GEMM to tiny-output decoding-like GEMMs. We plot both ideal arithmetic intensity and measured cuBLAS throughput, normalized to s_1. As the output space shrinks and the shape becomes more skewed, arithmetic intensity drops monotonically, and cuBLAS throughput eventually falls by orders of magnitude. The sweep is synthetic, but the stressed shapes arise when sparse activation and model sharding produce tiny-M GEMMs in both small-K and large-K regimes.

3.2 Tiny Output Spaces and CTA Underutilization

Low arithmetic intensity means that relatively few arithmetic operations are performed per byte transferred. A small output space also limits the number of CTAs that can be launched. The hierarchical mapping in Figure 2 is designed to maximize reuse inside large CTA, warp, and instruction tiles, but tiny-M and skewed shapes may provide too little native $M \times N$ output work to use the SM pool effectively. With too few CTAs, a kernel cannot involve enough SMs or issue enough concurrent memory requests to approach the HBM bandwidth assumed by the intensity argument. Figure 3(b) reports the real CTA counts launched by cuBLAS. We use the measured CTA counts to characterize the parallelism exposed by cuBLAS. The most skewed shapes still suffer the throughput collapse in Figure 3(a), showing that the bottleneck is also the mismatch between tiny GEMM work and useful CTA-level parallelism, data reuse, and memory-system utilization.

Just Accepted

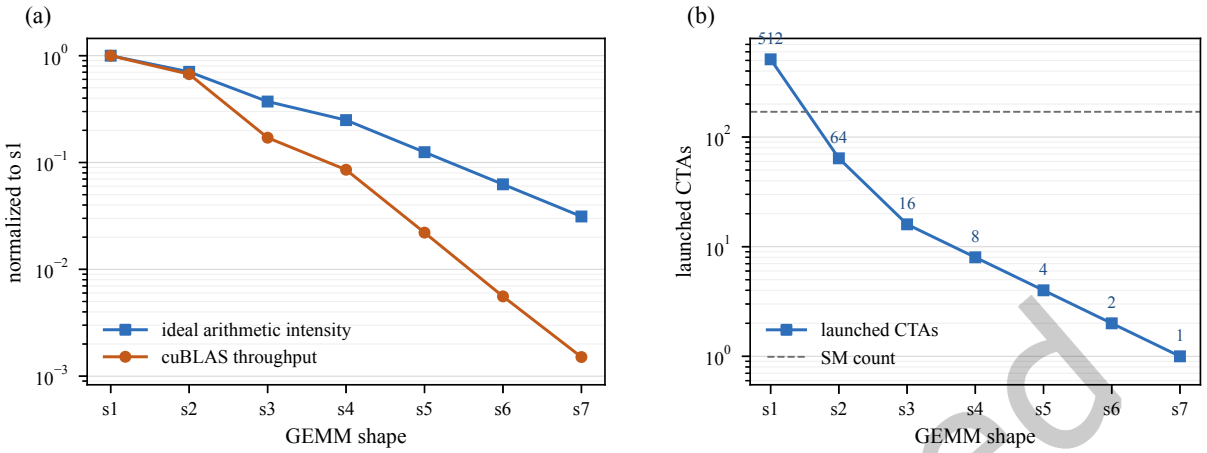


Fig. 3. Fixed-FLOP FP32 GEMM shape sweep on an RTX 5090. Each shape tuple s_i follows (M, N, K) order. Both panels use $MNK = 2^{30}$ with $s_1 = (1024, 1024, 1024)$, $s_2 = (512, 512, 4096)$, $s_3 = (256, 256, 16384)$, $s_4 = (128, 256, 32768)$, $s_5 = (64, 128, 131072)$, $s_6 = (32, 64, 524288)$, and $s_7 = (16, 32, 2097152)$. (a) Normalized ideal arithmetic intensity and cuBLAS throughput. (b) Nsight Compute launched CTA count; the dashed line marks the 170 SMs.

Just Accepted

This changes the optimization target. The goal is to select a CA algorithm that increases CTA-level parallelism while controlling data movement, synchronization, and partial-accumulation costs. For $\mathbf{C} = \mathbf{AB}$, additional CTA work can come from the output space (M, N) or the reduction dimension K. These axes lead to different CA structures. When K is small, reduction-space decomposition creates partial sums with too little computation to amortize the reduction overhead, so KABUMI uses 1D CA output-space expansion: it partitions the wide output dimension, keeps the tiny **A** operand resident within the owning CTA, streams columns of **B**, and writes disjoint **C** tiles. When K is large, 3D CA reduction-slice expansion can expose useful reduction-axis work, but pays extra traffic and accumulation overhead. Thus the expansion axis should follow the local K regime rather than a fixed decomposition, motivating KABUMI's shape-aware single-GEMM CA expansion.

3.3 Batched GEMM and Shared-SM Pressure

Shape-dependent CA expansion is only the first layer of the problem. In batched GEMM, the P independent instances share dimensions, datatype, layout, and kernel mode, so the resource decision can be summarized by a per-shape SM footprint and a batch size. This makes SM allocation a kernel-level scheduling problem: the number of CTAs generated by each GEMM affects how many GEMMs can execute concurrently. This view is consistent with GPU multitasking work on spatial multitasking [5], elastic kernels [50, 72], simultaneous-kernel execution, and model-based management [40, 64, 67], which shows the importance of accounting for shared SM, cache, and memory-bandwidth resources when scheduling concurrent work.

For one GEMM, exposing more CTA parallelism is desirable until additional SM participation stops reducing latency. For a batch, however, the same choice also changes how many other instances can run at the same time. Expanding every GEMM can waste SM slots when the batch already contains enough independent work, while assigning a small fixed footprint under-provisions small batches. The right granularity depends jointly on the shape, which determines useful single-GEMM expansion, and on P, which determines batch-level pressure on the SM pool.

Let S_{tot} be the number of SMs. In practice, the scheduler evaluates each candidate through its concrete launch footprint for the current shape and batch, including the number of CTA waves it generates and its per-CTA resource requirements. If the batch produces fewer CTA waves than the SM pool can use, per-GEMM expansion can fill otherwise idle SMs; once the batch already produces enough CTA waves, each GEMM needs a budget so that multiple instances can make progress concurrently. A solution must therefore separate shape-level CTA generation from batch-level exposure, and then decide how much generated work to expose for the current batch.

This motivates a solution that adapts the GEMM decomposition to the matrix shape and adjusts the resulting CTA count according to the batch size.

4 KABUMI

4.1 Overview

KABUMI is a CA batched GEMM framework for GPUs. Like the cuBLAS pointer-array interface evaluated in this paper, its batched-GEMM interface accepts P independent GEMMs with distinct addresses but common (M, N, K), datatype, and layout. KABUMI processes the batch in one CUDA launch. Its grouped-GEMM interface permits the expert-specific row count M_e, which represents the number of input tokens assigned to expert e, to vary while keeping (N, K) fixed by the model projection; all active experts are processed in one CUDA launch. For both interfaces, KABUMI selects a CA algorithm for the GEMMs sharing the GPU.

Figure 4 illustrates the overall design of KABUMI. The shape-aware kernel layer supplies candidates $\mathcal{H}_g$, their CTA counts $N_{\text{CTA}}(g, h)$, and estimated useful SM footprints $S_{\text{cap}}(g, h)$. The batch-aware scheduler then selects how much candidate parallelism to expose and realizes that budget through the output-space candidate or reduction-slice factor.

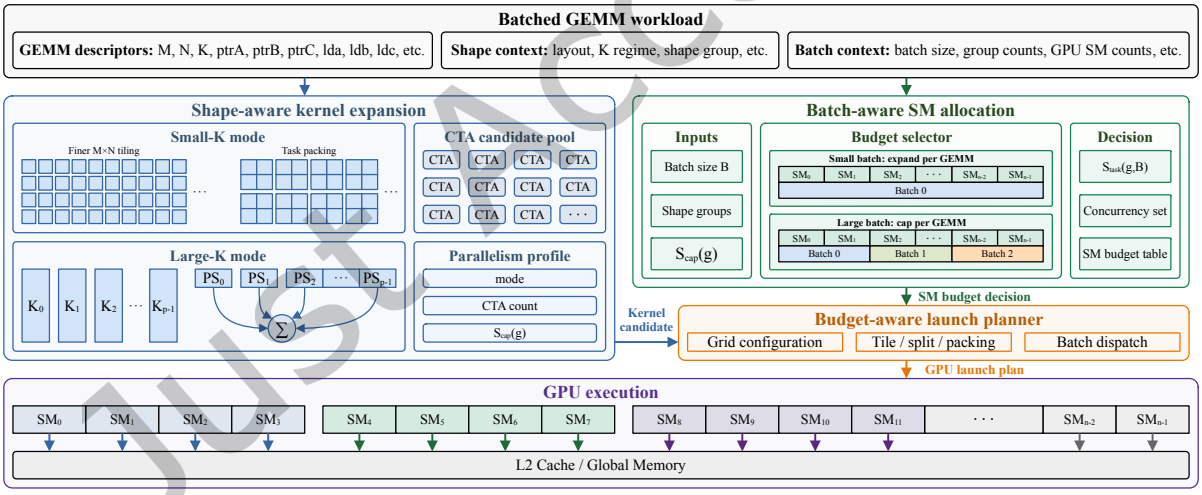


Fig. 4. Overview of KABUMI, the proposed CA batched and grouped GEMM framework.

For clarity, we first describe the scheduling policy for batched GEMM: P independent GEMMs that share (M, N, K), datatype, layout, and kernel mode. The core mechanism is the interaction between shape-aware single-GEMM CA expansion and batch-aware SM budgeting. Grouped GEMM applies the same mechanism to its expert descriptors.

Table 1 summarizes the notation used in this paper.

Just Accepted

Table 1. Notation used in this paper.

Symbol	Meaning
$\mathbf{A}_p$	Left operand of GEMM p, $\mathbf{A}_p \in \mathbb{R}^{M\times K}$.
$\mathbf{B}_p$	Right operand of GEMM p, $\mathbf{B}_p \in \mathbb{R}^{K\times N}$.
$\mathbf{C}_p$	Output of GEMM p, $\mathbf{C}_p \in \mathbb{R}^{M\times N}$.
M, N, K	GEMM dimensions; M is the local token or row dimension.
$I(M, N, K)$	Estimated arithmetic intensity, $2MNK/[e(MK + KN + MN)]$.
P	Number of GEMMs in the current batch.
p	Batch-element index, $0 \le p < P$.
g	GEMM shape/configuration, including dimensions, datatype, layout, and kernel mode.
h	Candidate expansion index.
S_{tot}	Total number of SMs on the GPU.
$\mathcal{H}_g$	CA expansion candidates available for shape/configuration g.
$N_{\text{CTA}}(g, h)$	CTAs generated per GEMM by candidate $h \in \mathcal{H}_g$.
$N_{\text{native}}(g)$	Native CTAs per GEMM before expansion.
$N_{\text{budget}}(P)$	Target per-GEMM CTA count for batch size P.
N_{sel}	CTA count of the selected candidate.
W_{target}	Target CTA waves per SM used by the planner.
$S_{\text{cap}}(g, h)$	Useful SM footprint estimated from generated CTA waves.
$S_{\text{task}}(g, P)$	Selected per-GEMM SM footprint for shape g at batch size P.
R_K	Number of reduction-axis slices in the large-K 3D CA reduction-slice path.
q	Reduction-slice index, $0 \le q < R_K$.
T_M, T_N	Row and column dimensions of a small-K output tile.
T_{block}	Number of threads in a multi-warp CTA.
m, n, k	Element indices along the M, N, and K dimensions.
i, t, u, v	Local indices for output elements and output-tile coordinates.
k_0, k_1	Start and end indices of a reduction slice.
e	Element size in bytes.

Just Accepted

4.2 Shape-Aware Single-GEMM CA Expansion

The first question is how much useful CA parallelism one GEMM can expose before batch-level sharing is considered. A conventional compute-oriented decomposition may emit too few CTAs for tiny-M decoding shapes because the output tile space is small. KABUMI's kernel layer instead treats SM participation and communication cost as joint objectives: it expands the CTA space when doing so can increase bandwidth utilization, but it stops when the extra CTAs would mostly add scheduling overhead, redundant traffic, or partial-reduction cost. For each candidate h, the kernel layer records the generated CTA count $N_{\text{CTA}}(g, h)$ and reports an estimated useful SM footprint to the scheduler.

The footprint is a property of both the shape and the selected expansion candidate. The implementation uses a CTA-wave model: a candidate with $N_{\text{CTA}}(g, h)$ independent CTAs can cover at most

$$S_{\text{cap}}(g, h) = \min\left(S_{\text{tot}}, \left\lceil \frac{N_{\text{CTA}}(g, h)}{W_{\text{target}}} \right\rceil\right) \tag{1}$$

useful SMs, where W_{target} is the desired number of CTA waves per SM. This cap should be read as a launch-planning estimate, not as a hard architectural partition: the GPU scheduler still assigns CTAs dynamically, but a

candidate with too few CTAs cannot occupy the whole SM pool, and a candidate with far too many tiny CTAs can add overhead without improving latency.

If K is short, applying 3D CA along the reduction dimension provides little useful work and introduces unnecessary partial sums. The small-K path therefore uses 1D CA output-space expansion: it creates CTAs over compact output strips or tiles, keeps the tiny **A** operand close to the CTA, and streams **B** columns without cross-CTA reduction. If K is long, the output space alone is still too small, but the reduction dimension contains enough work to create independent partial CTAs. The large-K path therefore uses 3D CA and accumulates partial outputs.

4.2.1 Small-K: 1D CA Output-Space Expansion. For small-K shapes, reduction-space decomposition is ineffective because the reduction is already short. This regime appears when model or tensor-parallel sharding leaves a local GEMM with a small reduction dimension. KABUMI therefore uses 1D CA output-space expansion rather than splitting K. Since M is tiny and N is wide, the dominant expansion axis is the output-column dimension: each candidate owns a disjoint strip or tile of the $M \times N$ output, keeps the tiny **A** operand close to the owning CTA, streams columns of **B**, writes directly to **C**, and avoids cross-CTA reduction.

Algorithm 1 Small-K 1D CA output-space execution

Require: Batched descriptors $\{\mathbf{A}_p, \mathbf{B}_p, \mathbf{C}_p\}_{p=0}^{P-1}$, small-$K$ shape (M, N, K), selected candidate h with tile $(T_M^{(h)}, T_N^{(h)})$

$U \leftarrow \lceil M/T_M^{(h)} \rceil, \quad V \leftarrow \lceil N/T_N^{(h)} \rceil$
for all CTA coordinates $(u, v, p) \in [0, U) \times [0, V) \times [0, P)$ in parallel **do**
  $M_u \leftarrow [uT_M^{(h)}, \min(M, (u+1)T_M^{(h)}))$
  $N_v \leftarrow [vT_N^{(h)}, \min(N, (v+1)T_N^{(h)}))$
  Assign this CTA as the unique owner of $\mathbf{C}_p[M_u, N_v]$
  Obtain $\mathbf{A}_p[M_u, :]$ using the register/cache/shared-memory policy of h
  Initialize the register accumulators for $\mathbf{C}_p[M_u, N_v]$
  for all short-K chunks in $[0, K)$ **do**
    Load the owned columns $\mathbf{B}_p[:, N_v]$ with coalesced transactions
    Accumulate the output tile with CUDA-core FMA or Tensor Core MMA
  end for
  Store the final tile directly to $\mathbf{C}_p[M_u, N_v]$
end for

For a fixed row tile u, the column partitions v form the 1D CA processor grid: the small row panel of **A** is logically replicated across column owners, while their **B** panels and **C** tiles are disjoint. Unlike the large-K path, this grid has no independently scheduled reduction coordinate and therefore produces no cross-CTA partial sums.

The cached-A candidates keep the same 1D output-space CA principle but increase the work per CTA. The skinny cached-A kernel loads the tiny left operand once for the CTA and streams columns of **B**, while the wider cached-A kernel computes more adjacent N columns per CTA. These candidates reduce scheduling overhead and improve reuse when K grows within the small-K regime, or when the batch already provides enough CTA waves that finer output granularity would create excessive scheduling overhead and decrease throughput. KABUMI therefore selects among 1D CA candidates with different output tile sizes. Eligible low-precision candidates retain the same disjoint-output ownership while using direct Tensor Core tiles.

Just Accepted

For a small-K candidate h with output tile $(T_M^{(h)}, T_N^{(h)})$, the generated CTA count is

$$N_{\mathrm{CTA}}^{\mathrm{small}}(g,h) = \left\lceil \frac{M}{T_M^{(h)}} \right\rceil \left\lceil \frac{N}{T_N^{(h)}} \right\rceil. \tag{2}$$

Smaller tiles increase SM participation but also increase CTA scheduling overhead; coarser cached-A tiles reduce overhead and reuse the small left operand across more output columns. The batch-aware layer therefore chooses the candidate whose generated CTA count, estimated SM footprint, and profiled cost best match the current batch pressure.

4.2.2 Large-K: 3D CA Reduction-Slice Expansion. For large-K shapes, KABUMI uses a 3D CA reduction-slice structure inside one GPU. Classical 3D CA algorithms introduce a processor dimension along the reduction axis and then combine partial outputs; in KABUMI, this 3D CA dimension maps to reduction-slice CTAs and provides additional schedulable work for underfilled large-K GEMMs. Each output element is assigned to multiple reduction slices, where each CTA computes a partial sum over a slice of K, and the partial sums are accumulated into the same output element. This mode is useful when the output tile space alone cannot fill the GPU but the reduction dimension is long enough to amortize the extra accumulation work.

With reduction-slice factor R_K, the large-K 3D CA path generates

$$N_{\mathrm{CTA}}^{\mathrm{large}}(g,R_K) = R_K \left\lceil \frac{MN}{T_{\mathrm{block}}} \right\rceil \tag{3}$$

CTAs per GEMM in this scalarized description. Increasing R_K raises SM participation, but it also increases partial-output traffic and atomic accumulation pressure. The selected R_K is therefore the smallest value that approaches the desired CTA-wave budget, rather than the largest possible reduction-slice factor.

Algorithm 2 shows the large-K kernel. The launch grid uses the z dimension for the reduction-slice id. Each CTA maps its linear tile coordinate to an output element (m,n) and its reduction-slice coordinate to a range $[k_0,k_1)$ of the reduction dimension. The CTA accumulates that partial dot product and atomically adds it to $\mathbf{C}_p[m,n]$. The reduction-slice factor is selected only up to the point where increased SM participation offsets the extra traffic and accumulation overhead.

Algorithm 2 is a scalarized description of the 3D CA path. The optimized kernels preserve the same budgeted reduction-slice structure: CUDA-core variants vectorize adjacent outputs and row-tile tiny-M rows, while the Tensor Core variant groups several logical reduction slices inside one CTA and computes the partial dot products with MMA instructions.

This large-K mode uses the 3D CA structure with a GPU-specific objective. The additional reduction-worker dimension is useful in our setting because it increases the available CTA-level parallelism of an otherwise underfilled memory-bound GEMM and can improve memory-system and bandwidth utilization; it does not guarantee bandwidth saturation. We therefore use 3D CA primarily for its parallelism benefit inside one GPU, while still accounting for the extra partial-output traffic and accumulation cost. The best reduction-slice factor is not necessarily the one that maximizes isolated Tensor Core utilization; it is the one that minimizes latency under the scheduler's current CTA-wave budget.

4.3 Batch-Aware SM Budgeting

The second question is how much of the single-GEMM expansion should be exposed when P independent GEMMs share the GPU. KABUMI does not reserve SMs explicitly. Instead, it controls the number of CTA waves generated per GEMM. This matches how the implementation launches work: all descriptors in the batch are issued as one CUDA launch, with the output-tile dimension, the batch dimension, and, for large-K shapes, the 3D CA reduction-slice dimension forming the launch grid.

Just Accepted

Algorithm 2 Large-K 3D CA reduction-slice expansion

Require: Batched descriptors $\{\mathbf{A}_p, \mathbf{B}_p, \mathbf{C}_p\}_{p=0}^{P-1}$, shape (M, N, K), batch size P, reduction-slice factor R_K

Zero-initialize all output matrices $\mathbf{C}_p$
Use a conventional multi-warp CTA with $T_{\text{block}} = 256$ threads
for all CTA coordinates (t, p, q) in parallel **do**
  $i \leftarrow t \cdot T_{\text{block}} + \text{threadIdx.x}$
  if $i \geq M \cdot N$ **then**
    return
  end if
  $m \leftarrow \lfloor i/N \rfloor, \quad n \leftarrow i - mN$
  $k_0 \leftarrow \lfloor Kq/R_K \rfloor, \quad k_1 \leftarrow \lfloor K(q+1)/R_K \rfloor$
  $acc \leftarrow 0$
  for $k \leftarrow k_0$ **to** $k_1 - 1$ **do**
    $acc \leftarrow acc + \mathbf{A}_p[m, k] \cdot \mathbf{B}_p[k, n]$
  end for
  $\text{atomicAdd}(\mathbf{C}_p[m, n], acc)$
end for

4.3.1 CTA-Wave Budget Estimation. The planner estimates the required CTA count using a CTA-wave model. Let W_{target} be the desired number of CTA waves per SM; in our evaluation we use two waves per SM. Section 5.4 evaluates the sensitivity of this choice. A batch is underfilled when its native output mapping cannot generate enough CTAs to cover the target wave count,

$$P \cdot N_{\text{native}}(g) < W_{\text{target}} S_{\text{tot}}. \tag{4}$$

For the scalar output mapping used as the native large-K reference,

$$N_{\text{native}}(g) = \left\lceil \frac{MN}{T_{\text{block}}} \right\rceil. \tag{5}$$

The target per-GEMM CTA count for the current batch is therefore

$$N_{\text{budget}}(P) = \left\lceil \frac{W_{\text{target}} S_{\text{tot}}}{P} \right\rceil. \tag{6}$$

When P is small, N_{budget} is large and KABUMI exposes more intra-GEMM expansion. When P grows, the budget naturally decreases because inter-GEMM concurrency already supplies many CTAs.

For small-K shapes, N_{budget} determines a deterministic selection from the output-space candidates in $\mathcal{H}_g$. KABUMI uses a compact profiled selection rule over the measured candidates: fine CUDA-core output mappings, cooperative cached-**A** tiles, and direct Tensor Core tiles when the datatype and shape permit them. The rule prefers candidates whose $N_{\text{CTA}}(g, h)$ reaches the current CTA-wave budget and selects the lowest-latency candidate among them; if no candidate reaches the budget, it selects the candidate with the largest useful CTA footprint. In the CUDA implementation, this profiled rule is encoded as compact thresholds over M, N, K, and P for the common small-K cases.

For large-K shapes, the same budget maps directly to the reduction-slice factor. Since the 3D CA path generates approximately $R_K N_{\text{native}}(g)$ CTAs per GEMM in the scalarized model, the budget-mode reduction-slice factor is chosen as

$$R_K = 2^{\left\lceil \log_2 \max\left(1, \left\lceil \frac{N_{\text{budget}}(P)}{N_{\text{native}}(g)} \right\rceil\right) \right\rceil}, \tag{7}$$

Just Accepted

then clamped by the configured maximum, the reduction length, and the hardware SM count. The evaluated SGEMM rule also applies K-dependent minimum reduction-slice factors for common long-reduction cases, and the FP16 Tensor Core rule rounds the factor to a multiple of four so that four reduction-slice warps reduce inside one CTA.

Algorithm 3 CTA-wave budget policy for one batched GEMM

Require: GEMM shape g, batch size P, total SMs S_{tot}, target waves W_{target}

$N_{\text{budget}} \leftarrow \left\lceil W_{\text{target}} S_{\text{tot}} / P \right\rceil$
if g is a small-K shape **then**
  Select $h \in \mathcal{H}_g$ from the deterministic small-K selection rule
  $N_{\text{sel}} \leftarrow N_{\text{CTA}}(g, h)$
else
  $N_{\text{native}} \leftarrow \lceil MN / T_{\text{block}} \rceil$
  $R_K \leftarrow \text{nextPow2}\left(\max\left(1, \left\lfloor N_{\text{budget}} / N_{\text{native}} \right\rfloor\right)\right)$
  Clamp R_K by the configured maximum, K, and the SM count
  $h \leftarrow$ the 3D CA candidate with reduction-slice factor R_K
  $N_{\text{sel}} \leftarrow R_K N_{\text{native}}$
end if
$S_{\text{task}} \leftarrow \min\left(S_{\text{tot}}, \left\lceil N_{\text{sel}} / W_{\text{target}} \right\rceil\right)$
Launch all P GEMMs in the batch with candidate h

This policy creates three practical regimes. If $P \cdot N_{\text{native}}(g)$ already exceeds the CTA-wave target, the batch has enough native work and aggressive expansion should be capped. If the native batch is underfilled but $P \cdot N_{\text{CTA}}(g, h)$ reaches the target for some expanded candidate, expansion can fill the missing CTA waves. If even the expanded candidate remains below the target, KABUMI still uses the best available expansion, but the shape is fundamentally limited by the amount of independent work it can generate.

4.4 Budget-Aware Launch Realization

The selected SM budget is not a hard architectural partition. It is realized by choosing launch parameters that control the number of CTAs per GEMM, yielding an estimated useful SM footprint of approximately S_{task}. The GPU scheduler still assigns CTAs dynamically, but controlling the generated CTA waves prevents a single GEMM from over-exposing work when the batch itself already provides enough independent CTAs.

For small-K shapes, the selected candidate fixes the output tile size and therefore the number of CTAs generated over the (M, N, P) grid. For large-K shapes, the selected reduction-slice factor adds the 3D CA reduction-slice dimension, producing approximately $R_K N_{\text{native}}(g)$ CTAs per GEMM in the scalarized model. In both cases, the whole batch is issued as one CUDA launch; KABUMI shapes the amount of schedulable work rather than assigning SMs explicitly.

4.5 CUDA Implementation

KABUMI implements CUDA interfaces for batched and grouped GEMM. The batched path launches pointer-array GEMMs that share (M, N, K), datatype, and layout. The grouped path shares (N, K) but accepts expert-specific row counts M_e; prefix row offsets and compact tile descriptors map output tiles to experts, allowing one compute launch to process all active experts. Both paths select a common small-K or large-K regime per workload and clamp the generated CTA waves using the runtime SM count and batch-aware budget, without reserving SMs.

Just Accepted

For small-K shapes, independently scheduled CTAs own disjoint output tiles. Fine CUDA-core mappings expose additional output-column work, while cooperative tiles retain the tiny left operand in registers or shared memory and reuse it across adjacent columns fetched with vector transactions. Eligible FP16, BF16, and FP8 paths instead stage instruction fragments and issue inline Tensor Core MMA. Accumulators remain in registers and every route writes its final output directly, so the 1D CA path requires neither cross-CTA reduction nor a global partial-sum workspace.

For large-K shapes, the selected R_K maps to a reduction-worker coordinate. Each worker owns one output tile and a contiguous K interval, stages aligned operand panels, and accumulates its partial tile in registers. Workers are combined within a CTA when possible; cross-CTA routes use atomic accumulation or a bounded partial workspace. Across the Tensor Core routes, KABUMI uses inline PTX `mma.sync` and `ldmatrix` instructions rather than the WMMA API; selected pipelined paths use `cp.async` and padded or swizzled shared-memory layouts to reduce bank conflicts. These choices improve operand delivery without changing the 3D CA ownership or batch-aware split policy.

FP32 paths accumulate in FP32, while the evaluated FP16 paths use FP16 accumulation and output. The BF16 and E4M3 FP8 Tensor Core paths use FP32 accumulation, with BF16 output for FP8. FP8 scale construction and weight prepacking are setup operations outside the measured compute region. All precision-specific microkernels share the descriptor scheduler and the same 1D/3D CA policy.

When the complete workload exceeds available device memory, KABUMI divides independent batched instances or expert groups into chunks that fit in device memory while preserving the output layout. This partitioning does not change the underlying kernels.

5 Evaluation

5.1 Experimental Setup

In the evaluation, batched GEMM denotes a collection of GEMMs with common (M, N, K), whereas grouped GEMM denotes the MoE workloads with expert-specific M_e and common (N, K).

Table 2 summarizes the platforms. We compile KABUMI with `-O3`, targeting `sm_120` on RTX 5090 and `sm_100` on B200. On both GPUs, we compare KABUMI with pointer-array batched GEMM from cuBLAS. The RTX 5090 run uses cuBLAS 13.3.0.5 with the default math mode, while the B200 run uses the corresponding paths from CUDA 13.0. Both platforms additionally evaluate CUTLASS 4.5.1 batched GEMM, CUTLASS grouped GEMM, and regular pointer-array batched GEMM from MAGMA 2.9.0 on RTX 5090 and MAGMA 2.10.0 on B200. MAGMA and CUTLASS expose column-major GEMM descriptors in these tools, so we run the mathematically equivalent transposed problem with dimensions (N, M, K) and report results under the paper's row-major shape (M, N, K). For each shape, we report the fastest CUTLASS configuration that passes correctness verification. Each paired comparison uses identical warmup and measurement counts, and CUDA events time kernels only.

CUTLASS operation metadata identifies the selected FP32 kernels as SIMT and the selected FP16 kernels as predominantly TensorOp. Inspection of the evaluated MAGMA sources shows tuned CUDA SIMT templates for FP32, with padded shared-memory leading dimensions, and WMMA-based Tensor Core templates for FP16, with register and shared-memory staging rather than inline PTX. These conventional optimizations improve per-tile efficiency but do not add KABUMI's inter-CTA reduction parallelism; because cuBLAS does not expose its selected kernel implementation, we report its public numerical mode without inferring undocumented WMMA/PTX or shared-memory details.

For batched GEMM, we evaluate CUTLASS 4.5.1 strided-batched GEMM and CUTLASS grouped GEMM on the same inputs. CUTLASS Stream-K is evaluated on compatible $P = 1$ cases. For grouped GEMM with expert-specific M_e and common (N, K), FP16/BF16 compare with cuBLAS and CUTLASS grouped GEMM; BF16 additionally

Just Accepted

compares with FlashInfer grouped GEMM, and FP8 with its SM120 FP8 path at commit `3bbfeba`. To support E4M3 inputs on SM120, we replace the unsupported mixed-FP8 MMA atom with an E4M3 MMA atom.

Table 2. Experimental platforms.

Characteristic	RTX 5090	B200
Architecture	NVIDIA Blackwell	NVIDIA Blackwell
Compute capability	12.0	10.0
SM count	170	148
Device memory	32 GB GDDR7	180 GB HBM3e
Memory bandwidth	1.792 TB/s	7.7 TB/s
CUDA compiler	13.2.51, `sm_120`	13.0.88, `sm_100`

Table 3 lists the benchmark suites. The RTX 5090 batched-GEMM evaluation uses one small-K suite and one large-K suite with aligned M and P ranges, and all libraries are measured on the same shapes. The B200 evaluation repeats these FP32/FP16 batched-GEMM cases with the same four baselines. The suites are organized around the two algorithmic decisions in KABUMI: whether a single GEMM should expand over output coordinates or over the reduction dimension, and whether the current batch already supplies enough independent work. We report geometric mean speedups across test cases and use win rates to show how often KABUMI outperforms each baseline. We additionally evaluate Stream-K on 80 compatible points and grouped GEMM on 28 model-derived Up/Down and decode-batch combinations per precision.

Table 3. Benchmark suites used in the evaluation.

Suite	Shape range	Cases
Small-K	$M \in \{1, 2, 4, 8, 16\}$, $N \in \{256, 512, 1024, 2048\}$, $K \in \{4, 8, 16, 32, 64\}$, $P \in \{1, 8, 32, 64\}$	400 per precision
Large-K	$M \in \{1, 2, 4, 8, 16\}$, $N \in \{32, 64, 128, 256\}$, $K \in \{512, 1024, 2048, 4096\}$, $P \in \{1, 8, 32, 64\}$	320 per precision
Grouped GEMM	Seven model-derived TP=8 Up/Down pairs; device-local decode batch $\{1, 4, 16, 64\}$, variable M_e	28 per precision
Batch-aware ablation	One small-K output-expansion shape and one large-K 3D CA reduction-slice shape, swept over batch size with CA-only and budget-aware policies	28

Just Accepted

5.2 Overall Comparison

5.2.1 Batched GEMM. Figure 5 summarizes the RTX 5090 batched-GEMM results across the small-K and large-K suites in FP32 and FP16. Each baseline is evaluated on the same 1,440 cases: 400 for each small-K precision and 320 for each large-K precision. KABUMI achieves 1.60×, 2.45×, 2.64×, and 7.00× geomean speedup over cuBLAS batched GEMM, CUTLASS batched GEMM, CUTLASS grouped GEMM, and MAGMA batched GEMM, respectively. Across the four combinations of precision and K regime, the largest geometric mean speedups over these baselines are 1.82×, 7.08×, 6.10×, and 10.17×.

The two shape regimes expose different sources of benefit. In the small-K suite, KABUMI uses 1D CA output-space expansion to create disjoint output CTAs without reduction traffic. In the large-K suite, it uses 3D CA reduction-slice expansion when the native output grid underfills the GPU. The particularly large FP32 large-K

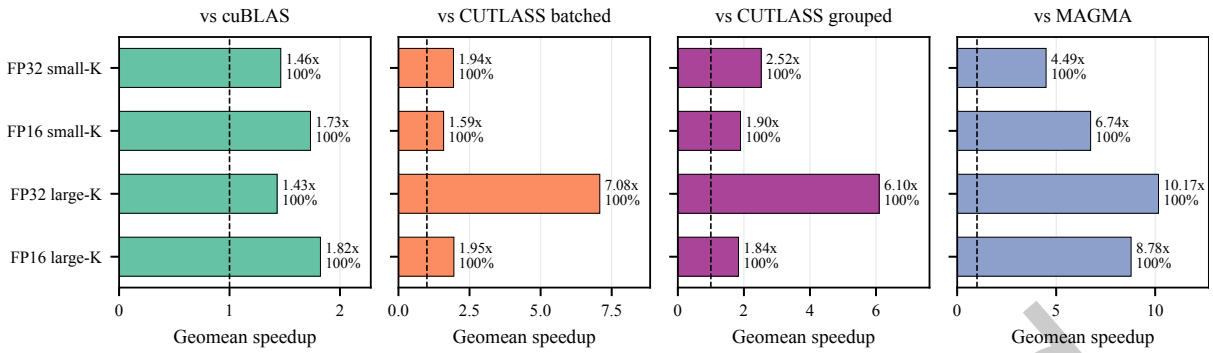


Fig. 5. Batched-GEMM performance on RTX 5090. Each panel reports KABUMI's geomean speedup over one baseline for FP32 and FP16 in the small-K and large-K regimes. KABUMI uses CUDA cores for FP32 and CUDA or Tensor Cores for FP16. Higher is better.

gains over CUTLASS and MAGMA reflect this controlled use of the long reduction dimension. CUTLASS Stream-K supports 80 FP16 large-K, $P = 1$ cases; KABUMI achieves 1.34× geomean speedup on these cases.

We also evaluate batched GEMM on B200 using the same FP32/FP16 cases and baselines as on RTX 5090. Figure 6 presents the overall results.

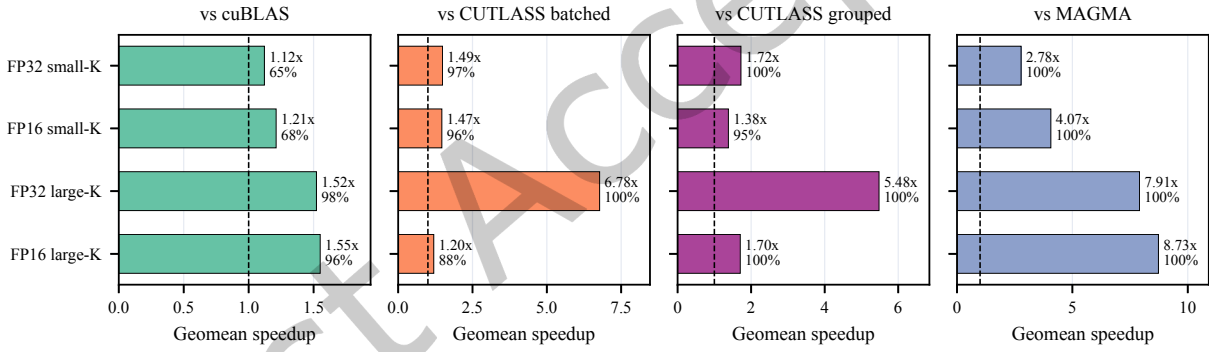


Fig. 6. Batched-GEMM performance on B200. Each panel reports KABUMI's geomean speedup over one baseline for FP32 and FP16 in the small-K and large-K regimes. KABUMI uses CUDA cores for FP32 and CUDA or Tensor Cores for FP16. Higher is better.

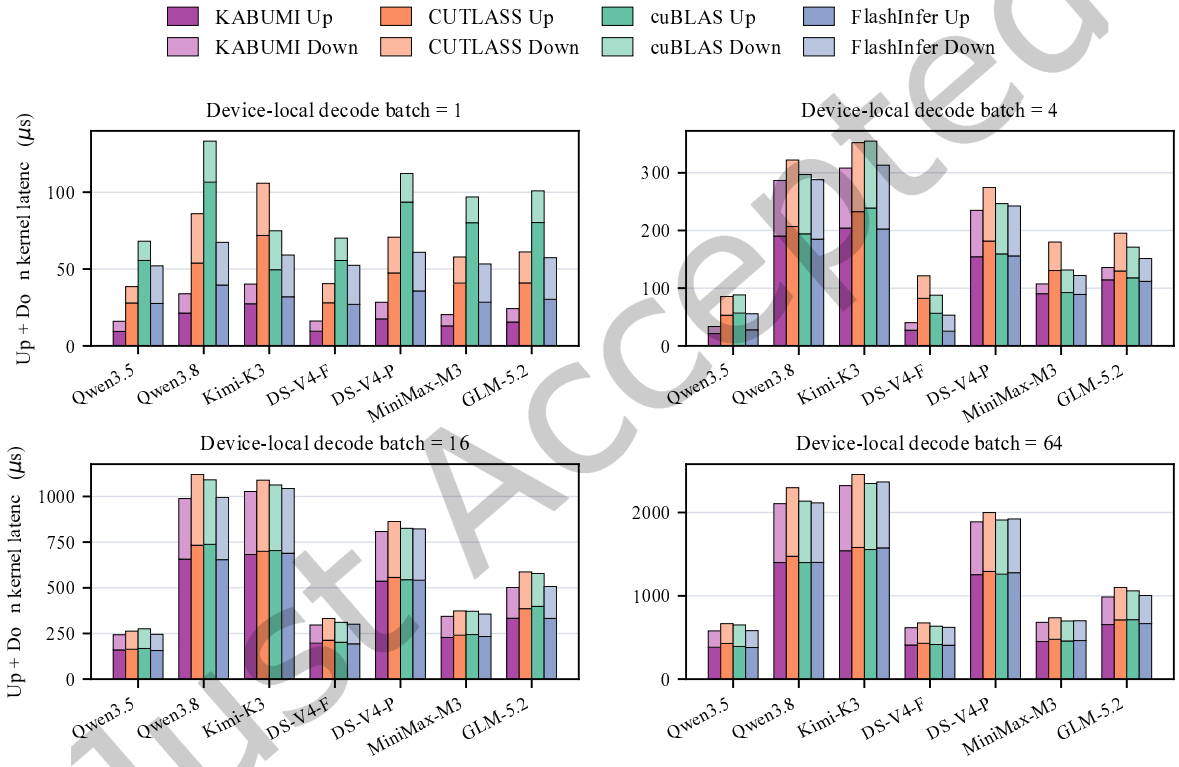
Across the 1,440 B200 cases, KABUMI achieves 1.32×, 1.98×, 2.09×, and 5.03× geomean speedup over cuBLAS, CUTLASS batched, CUTLASS grouped, and MAGMA, respectively. The corresponding highest speedups are 7.47×, 32.97×, 24.22×, and 36.53×. The strongest gains against both CUTLASS interfaces occur in FP32 large-K, where 3D CA exposes reduction-axis work beyond the native output grid.

5.2.2 Grouped GEMM. The grouped-GEMM evaluation uses seven model-derived MoE Up/Down shape pairs. We derive the expert projection shapes using TP=8 and evaluate device-local token batches of $\{1, 4, 16, 64\}$. The evaluation measures local GEMM computation without modeling distributed expert-parallel execution. For each model and batch, the tokens are routed among the local experts by the same controlled Dirichlet process, producing expert-specific row counts M_e that are supplied unchanged to every implementation. The reported

Just Accepted

paired compute latency is the sum of the pure Up and Down kernel times under the same routing and M_e, so Up exercises the large-K path and Down exercises the small-K path in one decode workload. All implementations use identical binary inputs, and correctness is verified for every workload. The FP8 path uses E4M3 block scaling, FP32 accumulation, and BF16 output. Across the 28 paired cases, KABUMI achieves 1.58× and 1.19× geomean speedup over cuBLAS and CUTLASS grouped GEMM in FP16; the corresponding BF16 speedups are 1.56× and 1.49×, with an additional 1.30× over FlashInfer. In FP8, KABUMI achieves a 1.11× geomean speedup over FlashInfer.

Figure 7 shows all 28 BF16 model-batch cases, the precision for which all three grouped library paths are available. Each bar stacks the Up and Down kernel compute times. The larger gains at batch 1 show the value of exposing intra-GEMM work when routing yields little native parallelism. As the batch grows, batch-level work increases and the gap narrows, while the batch-aware policy limits unnecessary expansion.



Just Accepted

Fig. 7. BF16 grouped-GEMM compute latency on RTX 5090 for all 28 model-derived MoE decode cases. The four panels correspond to device-local decode batches 1, 4, 16, and 64. Each bar stacks the Up (large-K) and Down (small-K) kernel times. Lower is better.

5.3 Shape and Batch Breakdown

Aggregate speedup hides the absolute throughput reached at individual shapes, so Figures 8 and 9 show representative shape-by-batch slices with all methods. Each panel reports effective GFLOPS on a shared log-scaled y-axis; higher bars indicate higher throughput, and bars are ordered as cuBLAS, CUTLASS, MAGMA, and KABUMI. The

vertical labels above bars show each method's throughput as a percentage of KABUMI for the same shape and batch.

In the FP32 small-K slice at $K = 8$, KABUMI is the fastest method at all 36 displayed points. This slice isolates the regime where reduction slicing would add overhead without useful additional arithmetic. KABUMI therefore partitions the wide output dimension among CTAs, each of which computes and writes its final output tile without cross-CTA reduction. The gains are strongest when the native output space is small enough that library kernels expose too few useful CTAs, but the batch is not so large that inter-GEMM concurrency already fills the device. As N or P grows, the baseline libraries receive more native or batch-level parallelism, so the relative gap becomes a test of whether KABUMI's output expansion adds work without creating redundant partial outputs.

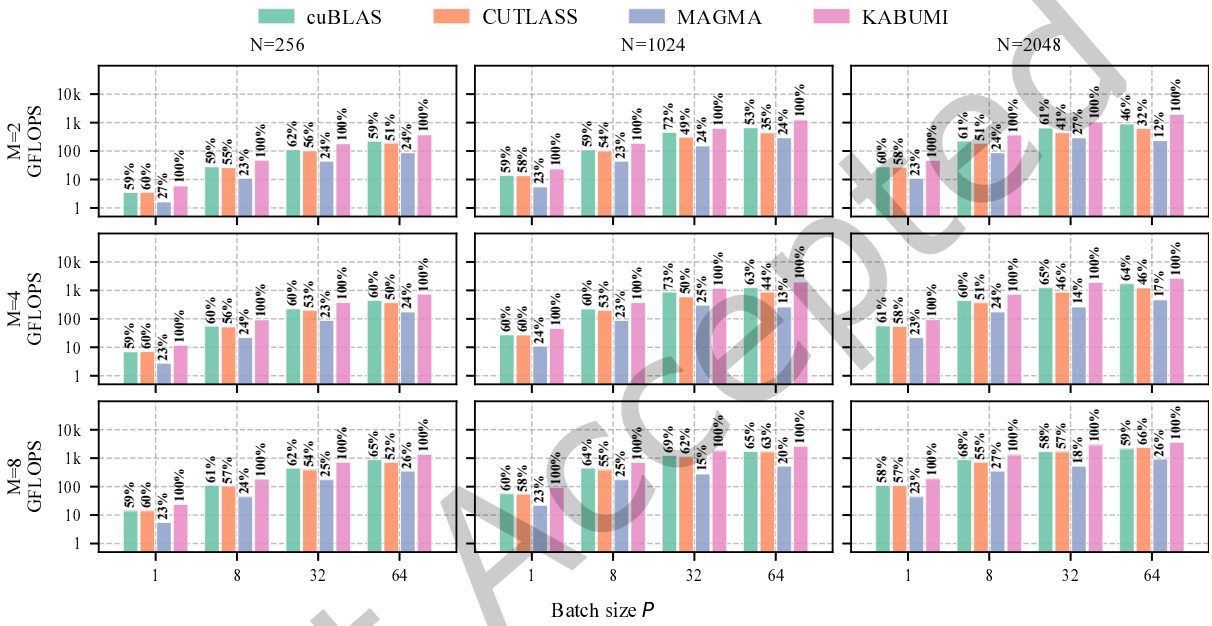


Fig. 8. FP32 small-K throughput breakdown on RTX 5090 for $K = 8$. Rows show representative $M \in \{2, 4, 8\}$ values, columns fix $N \in \{256, 1024, 2048\}$, and each panel sweeps $P \in \{1, 8, 32, 64\}$. The y-axis reports effective GFLOPS on a shared log scale; higher is better, and vertical bar labels report throughput relative to KABUMI.

The large-K slice in Figure 9 uses $K = 4096$ and representative $M \in \{2, 4, 8\}$ rows. Here the bottleneck is different from the small-K case: each output tile contains enough reduction work, but the tiny-M and narrow-N shapes can still expose too few output CTAs. 3D CA reduction-slice expansion converts the long K dimension into additional schedulable CTAs for the same output tile, then pays the partial-accumulation cost only where the extra SM participation is useful. The selected N columns include both narrow-output and wider-output cases, so the figure shows where this tradeoff remains favorable as native output parallelism and batch pressure increase. In this slice, KABUMI is the fastest method at all 36 displayed points; the geomean speedups over cuBLAS, CUTLASS, and MAGMA are 2.06×, 8.02×, and 11.59×, respectively.

5.4 Batch-Aware Scheduling Ablation and Sensitivity

Figure 10 isolates the two design steps behind KABUMI: CA expansion creates more single-GEMM parallelism, and batch-aware SM budgeting decides how much of that expansion should be exposed when many GEMMs

Just Accepted

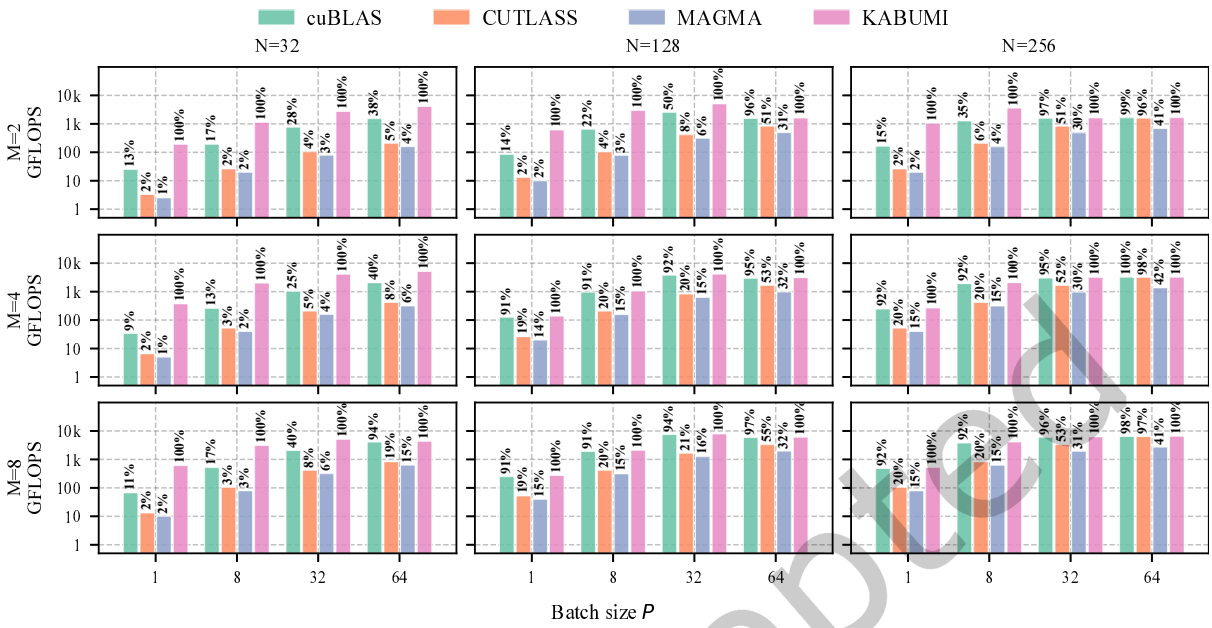


Fig. 9. FP32 large-K throughput breakdown on RTX 5090 at $K = 4096$. Rows show representative $M \in \{2, 4, 8\}$ values, columns fix $N \in \{32, 128, 256\}$, and each panel sweeps $P \in \{1, 8, 32, 64\}$. The y-axis reports effective GFLOPS on a shared log scale; higher is better, and vertical bar labels report throughput relative to KABUMI.

Just Accepted

share the GPU. The figure reports each method's performance as a percentage of KABUMI for the same shape and batch size, so values below 100% show the cost of removing a design component. The important comparison is not only KABUMI versus cuBLAS, but also KABUMI versus CA-only: if fixed CA expansion were sufficient, the CA-only curve would remain close to KABUMI across all batch sizes.

For the small-K shape $(M, N, K) = (4, 1024, 32)$, CA-only uses a fixed fine-grained 1D CA output-space expansion. KABUMI instead selects the coarser cooperative direct-write route across this sweep; CA-only latency is 1.28× that of KABUMI at $P = 1$. As P grows, however, fixed fine-grained expansion creates too many CTAs per batch; at $P = 64$, CA-only is 2.13× slower than KABUMI. Across the seven batch sizes, KABUMI achieves 1.42× geomean speedup over cuBLAS. This shows that the small-K decision is not simply to maximize CTA count: the selected 1D CA candidate must retain operand reuse and limit scheduling overhead as batch-level concurrency grows.

For the large-K shape $(M, N, K) = (4, 64, 1024)$, CA-only fixes an aggressive $R_K = 128$ chosen to expose reduction-space work for an underfilled single GEMM. This gives useful 3D CA parallelism at small batch sizes and is nearly identical to KABUMI for $P \leq 4$. Once batch-level concurrency grows, the fixed split over-expands each GEMM: at $P = 64$, CA-only is 1.17× slower than KABUMI, while KABUMI remains 1.23× faster than cuBLAS. KABUMI reduces the selected split from $R_K = 128$ for $P \leq 4$ to $R_K = 64$ at $P = 8$ and $R_K = 32$ for $P \geq 16$, preserving the benefit of 3D CA while avoiding high-batch over-expansion. The large-K ablation therefore exposes the central scheduler tradeoff: reduction slicing is valuable when it converts otherwise idle SMs into useful partial-dot-product work, but it becomes counterproductive when the extra partial outputs compete with enough batch-level work already in the queue.

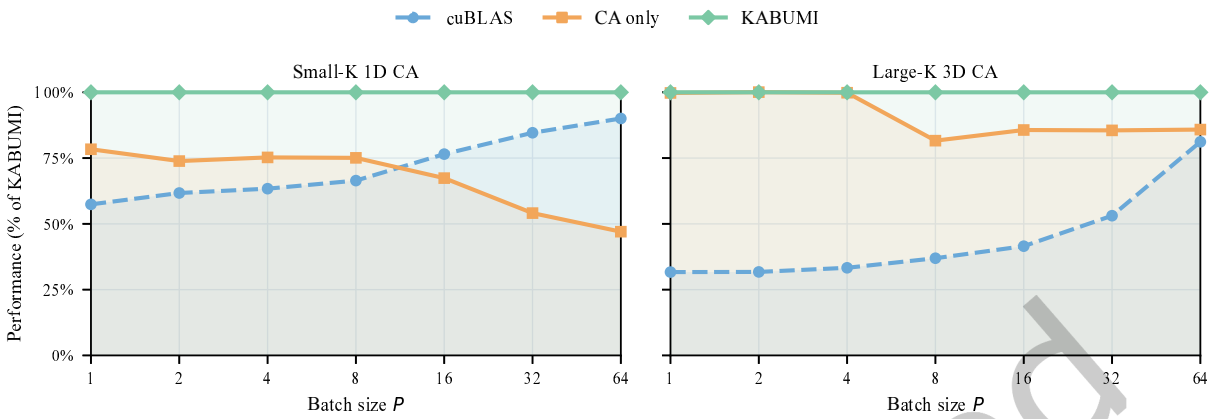


Fig. 10. Incremental ablation of CA expansion and batch-aware SM budgeting on RTX 5090. Performance is normalized to full KABUMI for each shape and batch size, with KABUMI fixed at 100%; higher is better. Both panels compare cuBLAS, fixed CA-only expansion, and full KABUMI.

We also evaluate $W_{\text{target}} \in \{1, 2, 3, 4\}$ over all 320 FP32 large-K cases. $W_{\text{target}} = 2$ and $W_{\text{target}} = 3$ select the same execution configuration for every test case. By contrast, $W_{\text{target}} = 1$ and $W_{\text{target}} = 4$ change the execution configuration for 39 and 50 test cases, respectively, and can increase latency by as much as 33% relative to $W_{\text{target}} = 2$. These results show that the scheduling decisions are insensitive to increasing W_{target} from 2 to 3, whereas decreasing it to 1 or increasing it to 4 can degrade performance. We therefore use $W_{\text{target}} = 2$ as the default.

5.5 Performance Characterization

To better understand the sources of KABUMI's performance gains, we use NVIDIA Nsight Compute to characterize representative small-K and large-K executions. Table 4 summarizes matched KABUMI/library profiles in terms of CTA counts, achieved occupancy, DRAM utilization, and L2 and L1 cache hit rates. The discussion below additionally examines Tensor Core and scheduler activity, executed instructions, memory traffic, and reduction requests to explain how the two CA algorithms affect device participation and data movement. CUDA-event measurements provide all latency and speedup results.

Communication volume and communication time are distinct. KABUMI uses CA to mean avoiding unnecessary communication on the critical path, not that every configuration transfers fewer bytes. The small-K path eliminates cross-CTA reduction through disjoint output ownership. The large-K path may increase partial-output traffic to expose more CTA parallelism; this expansion is used only when it reduces latency and is limited once batch-level parallelism is sufficient.

The small-K results show that KABUMI affects data movement differently across the memory hierarchy, rather than reducing traffic at every level. Against the selected CUTLASS kernel, FP16 KABUMI exposes four times as many output CTAs, increases DRAM utilization, and executes 49% fewer instructions with nearly equal DRAM bytes, while generating 2.61× as much L1/TEX traffic. Although its Tensor Pipe utilization is lower (0.69% versus 7.37%), it is 2.26× faster, confirming that this tiny-M point is limited by device participation and operand delivery rather than Tensor Core throughput. At high batch pressure, the FP32 route uses the same CTA count as cuBLAS but increases issue activity from 13.59% to 15.84% and DRAM utilization from 26.27% to 36.28%, with 21% fewer

Just Accepted

Table 4. Representative Nsight Compute profiles for KABUMI and the corresponding baselines. Pair-valued cells show KABUMI/baseline. CTA counts include all device-side components of the logical operation.

Precision, (M, N, K, P)	Baseline	Speedup	CTAs	Occ. (%)	DRAM (%)	L2 hit (%)	L1 hit (%)
FP16 $(8, 512, 16, 64)$	CUTLASS batched	2.26×	512/128	22.44/8.32	22.78/15.57	24.19/18.36	68.75/0.00
FP16 $(8, 256, 2048, 1)$	Stream-K	1.45×	260/178	9.55/8.74	13.72/10.17	54.50/27.05	32.90/0.00
FP32 $(4, 2048, 8, 64)$	cuBLAS	1.54×	2048/2048	80.15/35.12	36.28/26.27	8.80/9.45	47.63/14.58
FP32 $(1, 32, 4096, 1)$	cuBLAS	3.04×	129/4	3.02/16.61	6.17/1.95	40.35/19.54	11.98/25.34

instructions. Together, these results show that 1D CA direct-write decomposition and batch-aware coarsening improve performance through increased device utilization and more efficient instruction execution.

The large-K rows expose the 3D CA tradeoff. Against Stream-K, FP16 KABUMI increases CTA supply and DRAM activity and is 1.45× faster despite executing 2.15× as many instructions and issuing 1024 versus 12 global reduction requests. For FP32 $(1, 32, 4096, 1)$, KABUMI expands 4 CTAs to 129 and raises DRAM utilization from 1.95% to 6.17%, producing a 3.04× speedup even though achieved occupancy within active SMs is lower. Occupancy alone therefore does not reveal whole-device underfill. At the high-batch boundary $(1, 256, 1024, 64)$, KABUMI and cuBLAS both launch 256 CTAs, move equal DRAM bytes, and reach 88.34% and 89.01% DRAM utilization; KABUMI remains 1.19× faster while executing 59% fewer instructions. Here, the remaining gain does not come from further CTA expansion.

6 Related Work

6.1 LLM Inference and MoE Systems

MoE models increase parameter capacity by activating only a subset of experts for each token. Early sparse expert models [53], GShard [36], and Switch Transformer [20] established MoE as a practical scaling technique, and recent large models such as Mixtral [32] and DeepSeek [15] further demonstrate its importance in production-scale LLMs. This sparsity changes the local linear algebra problem: after routing, each expert sees only the tokens assigned to it, so decoding often produces many tiny-M GEMMs rather than a single large dense GEMM.

System work on MoE has mainly optimized distributed execution, routing, communication, and serving. DeepSpeed-MoE [52], FasterMoE [25], Tutel [30], MegaBlocks [21], and related systems improve training or inference throughput by changing how tokens, experts, and communication are organized. Inference serving systems such as vLLM [34], SGLang [71], and recent MoE serving optimizations [38, 44] address scheduling and memory management at a higher level of the stack. Serving optimizations also extend to intra-device parallelism, using nano-batching or prefill–decode overlap with interference-aware GPU resource allocation [41, 73]. Our work is complementary: after routing, batching, and sharding have formed local GEMM groups on one GPU, we study how the resulting batched GEMMs should be decomposed and scheduled across SMs.

6.2 GPU GEMM Libraries

Dense GEMM has a long history in numerical libraries, from ATLAS, GotoBLAS, OpenBLAS, and BLIS to LAPACK, MAGMA, and SLATE [17, 22, 23, 46, 59, 65, 68]. On GPUs, cuBLAS and CUTLASS [47, 48] provide highly optimized vendor kernels and reusable templates, while Triton [57] and newer AI-oriented libraries such as DeepGEMM [14] expose flexible routes for custom matrix kernels. A large body of GPU GEMM work studies tiling, blocking, memory hierarchy usage, and Tensor Core utilization [9, 28, 31, 35, 37, 39, 56, 61, 63, 69].

Conventional tiling strategies are highly effective for large, regular matrices, where enough output tiles exist to fill the GPU and arithmetic intensity is high. Decoding-oriented work also optimizes flat GEMMs through shape-specific kernels and double buffering [13]. For the tiny-M shapes studied here, the native output tile space

Just Accepted

can expose too few CTAs, and the best decomposition depends on whether the local K dimension is short or long. KABUMI therefore selects CA algorithms that balance data reuse with CTA-level parallelism to improve memory-system utilization while keeping overhead low for short reductions.

6.3 Batched and Small GEMM

Small-matrix multiplication has motivated both specialized kernels and batched execution. On CPUs, LIBXSMM uses runtime code generation to specialize small matrix multiplications [26]. On GPUs, early batched GEMM work combined specialized kernel designs with autotuning for fixed- and variable-size batches [2]. Batched BLAS efforts also examined interfaces and data layouts for efficiently processing many small operations together [19]. Batching reduces launch overhead and exposes parallelism across independent matrices, while kernel specialization improves the efficiency of each multiplication. Subsequent FP16 batched GEMM designs adapted kernel organization to Tensor Cores, including cases where very small matrices cannot fully utilize the matrix units [4].

Variation in matrix dimensions introduces further challenges in distributing work across the GPU. Variable-size batched BLAS handles differently sized matrices within one kernel through a common grid structure and adaptations to each problem's dimensions [3]. Tiling and batching are also closely coupled: larger tiles favor data reuse, while smaller tiles expose more parallel work. Li et al. [39] explicitly coordinate these choices through a tiling engine that partitions GEMMs and a batching engine that assigns tiles to thread blocks.

Beyond tile assignment within a batched kernel, resource sharing also matters when independent GEMMs execute as separate kernels. GOLDYLOC [51] couples resource-aware kernel tuning with dynamic selection of the number of concurrently executing GEMMs, accounting for contention that is absent when a kernel is tuned in isolation. These studies connect the efficiency of individual GEMMs with the parallelism and resources available to the workload as a whole. KABUMI studies this interaction for decoding-oriented tiny-M workloads, adapting CA decomposition and per-GEMM CTA counts to matrix shape and batch or group size.

6.4 CA Algorithms

Several GEMM decompositions improve load balance by changing how work is assigned to SMs. Stream-K [49] is representative of work-centric scheduling: instead of assigning work only through conventional output tiles, it distributes the GEMM iteration space more evenly across the GPU. Such approaches are closely related in spirit to our goal of avoiding underfilled SMs, but our setting is different because the unit of scheduling is a batch of tiny-M GEMMs whose single-GEMM expansion must be balanced against inter-GEMM concurrency.

CA algorithms provide another line of related work. Theoretical lower bounds and algorithms for reducing data movement have been studied extensively [7, 8, 18, 42]; practical distributed matrix multiplication algorithms include SUMMA, 2.5D variants, and near-optimal redistribution schemes [33, 55, 58]. KAMI [62] develops CA algorithms for single-GPU small GEMM, whereas KABUMI studies batch-aware CTA expansion for batched and grouped GEMMs.

7 Conclusion and Future Work

The performance of decoding-oriented tiny-M GEMMs depends on both CTA-level parallelism and efficient use of the memory system. In batched and grouped execution, the appropriate amount of parallelism within each GEMM also depends on how many GEMMs share the GPU.

We proposed KABUMI, a CA batched GEMM framework for this regime. Its single-GEMM kernels create more useful CTA-level work through shape-aware CA expansion, using 1D CA output-space expansion for small-K shapes and 3D CA reduction-slice expansion for large-K shapes. Its batch-aware scheduler adapts each GEMM's SM footprint to the batch or group size, improving latency for small batches and throughput for larger batches. On RTX 5090, KABUMI achieves 1.60×–7.00× geomean speedup over the batched-GEMM baselines and

Just Accepted

1.11×–1.58× over the evaluated grouped-GEMM baselines. On B200, it achieves 1.32×–5.03× geomean speedup over the batched-GEMM baselines. These results show that shape-aware CA expansion and batch-aware SM budgeting are effective tools for decoding-oriented batched GEMM on modern GPUs.

Future work will broaden the evaluation of KABUMI across GPU architectures to study performance portability, and extend its low-precision support and comparisons with low-precision GEMM libraries.

Acknowledgments

We greatly appreciate the invaluable comments of all reviewers. Weifeng Liu is the corresponding author of this paper. This work was partially supported by the National Key R&D Program of China (Grant No. 2023YFB3001605) and the National Natural Science Foundation of China (Grant No. U23A20301 and No. 62372467).

References

[1] Ahmad Abdelfattah, Hartwig Anzt, Erik G Boman, Erin Carson, Terry Cojean, Jack Dongarra, Alyson Fox, Mark Gates, Nicholas J Higham, Xiaoye S Li, Jennifer Loe, Piotr Luszczek, Srikara Pranesh, Siva Rajamanickam, Tobias Ribizel, Barry F Smith, Kasia Swirydowicz, Stephen Thomas, Stanimire Tomov, Yaohung M Tsai, and Ulrike Meier Yang. 2021. A survey of numerical linear algebra methods utilizing mixed-precision arithmetic. *The International Journal of High Performance Computing Applications* 35, 4 (2021), 344–369. https://doi.org/10.1177/10943420211003313

[2] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. 2016. Performance, Design, and Autotuning of Batched GEMM for GPUs. In *International Conference on High Performance Computing (ISC)*. 21–38. https://doi.org/10.1007/978-3-319-41321-1_2

[3] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. 2017. Novel HPC Techniques to Batch Execution of Many Variable Size BLAS Computations on GPUs. In *ACM International Conference on Supercomputing (ICS)*. 1–10. https://doi.org/10.1145/3079079.3079103

[4] Ahmad Abdelfattah, Stanimire Tomov, and Jack Dongarra. 2019. Fast Batched Matrix Multiplication for Small Sizes Using Half-Precision Arithmetic on GPUs. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 111–122. https://doi.org/10.1109/IPDPS.2019.00022

[5] Jacob T Adriaens, Katherine Compton, Nam Sung Kim, and Michael J Schulte. 2012. The case for GPGPU spatial multitasking. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA.2012.6168946

[6] Hartwig Anzt, Jack Dongarra, Goran Flegar, and Enrique S Quintana-Ortí. 2017. Variable-size batched LU for small matrices and its integration into block-Jacobi preconditioning. In *International Conference on Parallel Processing (ICPP)*. https://doi.org/10.1109/ICPP.2017.18

[7] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. 2011. Minimizing communication in numerical linear algebra. *SIAM J. Matrix Anal. Appl.* 32, 3 (2011), 866–901. https://doi.org/10.1137/090769156

[8] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. 2014. Communication costs of Strassen's matrix multiplication. *Commun. ACM* 57, 2 (2014), 107–114. https://doi.org/10.1145/2556647.2556660

[9] Noel Chalmers, Jakub Kurzak, Damon McDougall, and Paul Bauman. 2023. Optimizing high-performance linpack for exascale accelerated architectures. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. https://doi.org/10.1145/3581784.3607066

[10] Jieyang Chen, Nan Xiong, Xin Liang, Dingwen Tao, Sihuan Li, Kaiming Ouyang, Kai Zhao, Nathan DeBardeleben, Qiang Guan, and Zizhong Chen. 2019. TSM2: optimizing tall-and-skinny matrix-matrix multiplication on GPUs. In *ACM International Conference on Supercomputing (ICS)*. https://doi.org/10.1145/3330345.3330355

[11] Krishna Teja Chitty-Venkata, Sylvia Howland, Golara Azar, Daria Soboleva, Natalia Vassilieva, Siddhisanket Raskar, Murali Emani, and Venkatram Vishwanath. 2025. MoE-Inference-Bench: Performance Evaluation of Mixture of Expert Large Language and Vision Models. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SCW)*. https://doi.org/10.1145/3731599.3767706

[12] Jack Choquette. 2023. NVIDIA Hopper H100 GPU: Scaling Performance. *IEEE Micro* 43, 3 (2023), 9–17. https://doi.org/10.1109/mm.2023.3256796

[13] Guohao Dai, Ke Hong, Qiuli Mao, Xiuhong Li, Jiaming Xu, Haofeng Huang, Hongtu Xia, Xuefei Ning, Shengen Yan, Yun Liang, and Yu Wang. 2025. FlashDecoding++Next: High Throughput LLM Inference With Latency and Memory Optimization. *IEEE Trans. Comput.* 74, 10 (2025), 3263–3276. https://doi.org/10.1109/TC.2025.3585339

[14] DeepSeek-AI. 2025. DeepGEMM: Clean and Efficient GEMM Kernels for Deep Learning. https://github.com/deepseek-ai/DeepGEMM.

[15] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo

Just Accepted

Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. 2024. DeepSeek-V3 Technical Report. *arXiv preprint arXiv:2412.19437* (2024). https://doi.org/10.48550/arXiv.2412.19437

[16] Eliezer Dekel, David Nassimi, and Sartaj Sahni. 1981. Parallel matrix and graph algorithms. *SIAM Journal on computing* 10, 4 (1981), 657–675. https://doi.org/10.1137/0210049

[17] James Demmel. 1991. LAPACK: A portable linear algebra library for high-performance computers. *Concurrency: Practice and Experience* 3, 6 (1991), 655–666. https://doi.org/10.1002/cpe.4330030610

[18] Jim Demmel. 2012. Communication avoiding algorithms. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. https://doi.org/10.1109/sc.companion.2012.351

[19] Jack Dongarra, Sven Hammarling, Nicholas J Higham, Samuel D Relton, Pedro Valero-Lara, and Mawussi Zounon. 2017. The design and performance of batched BLAS on modern high-performance computing systems. *Procedia Computer Science* 108 (2017), 495–504. https://doi.org/10.1016/j.procs.2017.05.138

[20] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research* 23, 120 (2022), 1–39. https://doi.org/10.48550/arXiv.2101.03961

[21] Trevor Gale, Deepak Narayanan, Cliff Young, and Matei Zaharia. 2023. Megablocks: Efficient sparse training with mixture-of-experts. In *Machine Learning and Systems (MLSys)*. https://doi.org/10.48550/arXiv.2211.15841

[22] Mark Gates, Jakub Kurzak, Ali Charara, Asim YarKhan, and Jack Dongarra. 2019. SLATE: Design of a modern distributed and accelerated linear algebra library. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. https://doi.org/10.1145/3295500.3356223

[23] Kazushige Goto and Robert A van de Geijn. 2008. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Software* 34, 3 (2008), 1–25. https://doi.org/10.1145/1356052.1356053

[24] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J Higham. 2018. Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. https://doi.org/10.1109/sc.2018.00050

[25] Jiaao He, Jidong Zhai, Tiago Antunes, Haojie Wang, Fuwen Luo, Shangfeng Shi, and Qin Li. 2022. Fastermoe: modeling and optimizing training of large-scale dynamic pre-trained models. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. https://doi.org/10.1145/3503221.3508418

[26] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. LIBXSMM: accelerating small matrix multiplications by runtime code generation. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. https://doi.org/10.1109/sc.2016.83

[27] Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Kangdi Chen, Yuhan Dong, and Yu Wang. 2024. FlashDecoding++: Faster Large Language Model Inference with Asynchronization, Flat GEMM Optimization, and Heuristics. In *Machine Learning and Systems (MLSys)*. https://doi.org/10.48550/arXiv.2311.01282

[28] Huanqi Hu, Bowen Xiao, Shixuan Sun, Jianian Yin, Zhexi Zhang, Xiang Luo, Chengquan Jiang, Weiqi Xu, Xiaoying Jia, Xin Liu, and Minyi Guo. 2025. LiquidGEMM: Hardware-Efficient W4A8 GEMM Kernel for High-Performance LLM Serving. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. https://doi.org/10.1145/3712285.3759852

[29] Hua Huang and Edmond Chow. 2022. CA3DMM: a new algorithm based on a unified view of parallel matrix multiplication. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. https://doi.org/10.1109/sc41404.2022.00033

[30] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, HoYuen Chau, Peng Cheng, Fan Yang, Mao Yang, and Yongqiang Xiong. 2023. Tutel: Adaptive mixture-of-experts at scale. In *Machine Learning and*

Just Accepted

Systems (MLSys). https://doi.org/10.48550/arXiv.2206.03382
[31] Abhinav Jangda and Mohit Yadav. 2024. Fast kronecker matrix-matrix multiplication on gpus. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. https://doi.org/10.1145/3627535.3638489
[32] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2024. Mixtral of experts. *arXiv preprint arXiv:2401.04088* (2024). https://doi.org/10.48550/arXiv.2401.04088
[33] Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefler. 2019. Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. https://doi.org/10.1145/3295500.3356181
[34] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Symposium on Operating Systems Principles (SOSP)*. https://doi.org/10.1145/3600006.3613165
[35] E Scott Larsen and David McAllister. 2001. Fast matrix multiplies using graphics hardware. In *The ACM/IEEE Conference on Supercomputing (SC)*. https://doi.org/10.1145/582034.582089
[36] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2020. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668* (2020). https://doi.org/10.48550/arXiv.2006.16668
[37] Jiajia Li, Xingjian Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. 2012. An optimized large-scale hybrid DGEMM design for CPUs and ATI GPUs. In *ACM International Conference on Supercomputing (ICS)*. https://doi.org/10.1145/2304576.2304626
[38] Kexin Li, Wenkan Huang, Qinggang Wang, Long Zheng, Xiaofei Liao, Hai Jin, and Jingling Xue. 2025. Diff-MoE: Efficient Batched MoE Inference with Priority-Driven Differential Expert Caching. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. https://doi.org/10.1145/3712285.3759903
[39] Xiuhong Li, Yun Liang, Shengen Yan, Liancheng Jia, and Yinghan Li. 2019. A coordinated tiling and batching framework for efficient GEMM on GPUs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. https://doi.org/10.1145/3293883.3295734
[40] Yun Liang and Xiuhong Li. 2017. Efficient Kernel Management on GPUs. *ACM Transactions on Embedded Computing Systems* (2017). https://doi.org/10.1145/3070710
[41] Zejia Lin, Hongxin Xu, Guanyi Chen, Zhiguang Chen, Yutong Lu, and Xianwei Zhang. 2026. Bullet: Boosting GPU Utilization for LLM Serving via Dynamic Spatial-Temporal Orchestration. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 290–306. https://doi.org/10.1145/3779212.3790135
[42] Benjamin Lipshitz, Grey Ballard, James Demmel, and Oded Schwartz. 2012. Communication-avoiding parallel Strassen: Implementation and performance. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. https://doi.org/10.1109/SC.2012.33
[43] Weile Luo, Ruibo Fan, Zeyu Li, Dayou Du, Qiang Wang, and Xiaowen Chu. 2024. Benchmarking and dissecting the nvidia hopper gpu architecture. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. https://doi.org/10.1109/ipdps57955.2024.00064
[44] Songkai Ma, Zhaorui Zhang, Sheng Di, Benben Liu, Xiaodong Yu, Xiaoyi Lu, and Dan Wang. 2025. Compression Error Sensitivity Analysis for Different Experts in MoE Model Inference. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SCW)*. https://doi.org/10.1145/3731599.3767377
[45] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient large-scale language model training on gpu clusters using megatron-lm. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. https://doi.org/10.1145/3458817.3476209
[46] Rajib Nath, Stanimire Tomov, and Jack Dongarra. 2010. An Improved Magma Gemm For Fermi Graphics Processing Units. *The International Journal of High Performance Computing Applications* 24, 4 (2010), 511–515. https://doi.org/10.1177/1094342010385729
[47] NVIDIA Corporation 2026. *cuBLAS Library User Guide*. NVIDIA Corporation. https://docs.nvidia.com/cuda/cublas/
[48] NVIDIA Corporation 2026. *CUTLASS: CUDA Templates for Linear Algebra Subroutines*. NVIDIA Corporation. https://github.com/NVIDIA/cutlass
[49] Muhammad Osama, Duane Merrill, Cris Cecka, Michael Garland, and John D Owens. 2023. Stream-k: Work-centric parallel decomposition for dense matrix-matrix multiplication on the gpu. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. https://doi.org/10.1145/3572848.3577479
[50] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. 2013. Improving GPGPU Concurrency with Elastic Kernels. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 407–418. https://doi.org/10.1145/2451116.2451160

Just Accepted

[51] Suchita Pati, Shaizeen Aga, Nuwan Jayasena, and Matthew D. Sinclair. 2025. GOLDYLOC: Global Optimizations & Lightweight Dynamic Logic for Concurrency. *ACM Transactions on Architecture and Code Optimization* 22, 2 (2025), 1–28. https://doi.org/10.1145/3730584

[52] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. 2022. DeepSpeed-MoE: Advancing mixture-of-experts inference and training to power next-generation AI scale. In *International Conference on Machine Learning (ICML)*. https://doi.org/10.48550/arXiv.2201.05596

[53] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538* (2017). https://doi.org/10.48550/arXiv.1701.06538

[54] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019). https://doi.org/10.48550/arXiv.1909.08053

[55] Edgar Solomonik and James Demmel. 2011. Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms. In *European Conference on Parallel and Distributed Computing (Euro-Par)*. https://doi.org/10.1007/978-3-642-23397-5_10

[56] Guangming Tan, Linchuan Li, Sean Triechle, Everett Phillips, Yungang Bao, and Ninghui Sun. 2011. Fast implementation of DGEMM on Fermi GPU. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. https://doi.org/10.1145/2063384.2063431

[57] Philippe Tillet, 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*. Association for Computing Machinery, New York, NY, USA, 10–19. doi:10.1145/3315508.3329973

[58] Robert A Van De Geijn and Jerrell Watts. 1997. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience* 9, 4 (1997), 255–274. https://doi.org/10.1002/(sici)1096-9128(199704)9:4<255::aid-cpe250>3.0.co;2-2

[59] Field G Van Zee and Robert A Van De Geijn. 2015. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Trans. Math. Software* 41, 3 (2015), 1–33. https://doi.org/10.1145/2764454

[60] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems (NeurIPS)*. https://doi.org/10.48550/arXiv.1706.03762

[61] Vasily Volkov and James Demmel. 2008. Benchmarking GPUs to tune dense linear algebra. In *The ACM/IEEE conference on Supercomputing (SC)*. https://doi.org/10.1109/sc.2008.5214359

[62] Hemeng Wang, Yang Du, Sidu Li, Xiaowen Tian, Qingxiao Sun, and Weifeng Liu. 2025. KAMI: Communication-Avoiding General Matrix Multiplication within a Single GPU. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. https://doi.org/10.1145/3712285.3759895

[63] Weihu Wang, Yaqi Xia, Donglin Yang, Xiaobo Zhou, and Dazhao Cheng. 2025. MXBLAS: Accelerating 8-bit Deep Learning with a Unified Micro-Scaled GEMM Library. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. https://doi.org/10.1145/3712285.3759809

[64] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2016. Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA.2016.7446078

[65] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. 2001. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Comput.* 27, 1–2 (2001), 3–35. https://doi.org/10.1016/S0167-8191(00)00087-9

[66] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76. https://doi.org/10.1145/1498765.1498785

[67] Hao Wu, Weizhi Liu, Huanxin Lin, and Cho-Li Wang. 2020. A model-based software solution for simultaneous multiple kernels on GPUs. *ACM Transactions on Architecture and Code Optimization* (2020). https://doi.org/10.1145/3377138

[68] Xianyi Zhang. 2016. OpenBLAS: An optimized BLAS library. Retrieved April 7, 2025 from http://www.openmathlib.org/OpenBLAS/

[69] Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. 2017. Understanding the gpu microarchitecture to achieve bare-metal performance tuning. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. https://doi.org/10.1145/3018743.3018755

[70] Zheng Zhang, Hulin Wang, Hongming Xu, Donglin Yang, Xiaobo Zhou, and Dazhao Cheng. 2025. HyTiS: Hybrid Tile Scheduling for GPU GEMM with Enhanced Wave Utilization and Cache Locality. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. https://doi.org/10.1145/3712285.3759771

[71] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. Sglang: Efficient execution of structured language model programs. In *Advances in neural information processing systems (NeurIPS)*. https://doi.org/10.52202/079017-2000

[72] Jianlong Zhong and Bingsheng He. 2013. Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling. *IEEE Transactions on Parallel and Distributed Systems* (2013). https://doi.org/10.1109/TPDS.2013.257

Just Accepted

[73] Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Tian Tang, Qinyu Xu, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Ziren Wang, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. 2025. NanoFlow: Towards Optimal Large Language Model Serving Throughput. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 749–765. https://doi.org/10.48550/arXiv.2408.12757

Received 1 June 2026; revised 2 September 2026; accepted 2 October 2026

Just Accepted