BAG: Faster Matrix Multiplication on a Single GPU

Yao Liu
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
liuyao25e@ict.ac.cn

Yewen Li*
The Hong Kong University of Science and Technology
HongKong, China
yewenli@ust.hk

Zhonghai Zhang
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
zhonghaizhang@ict.ac.cn

Erlin Yao*
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
yaoerlin@ict.ac.cn

Guangming Tan
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
tgm@ict.ac.cn

Weile Jia
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
jiaweile@ict.ac.cn

Abstract

Matrix multiplication is a fundamental computation kernel in many parallel and sequential scientific applications. We target FP32 matrix multiplication on GPUs, a setting required by numerous HPC and scientific workloads. Alternative Basis Matrix Multiplication (ABMM) is a practical Strassen-like algorithm that reduces the number of additions while preserving the same asymptotic exponent and admitting a provable $O(n)$ component-wise error bound. However, optimized GPU support for ABMM remains underexplored. This paper presents BAG (Basis Alternative Matrix Multiplication on GPUs), a GPU-oriented implementation of ABMM for the NVIDIA Ampere architecture. A naive GPU implementation of ABMM suffers from excessive workspace and memory traffic, launch-dominated serialization across heterogeneous steps, and high sensitivity to recursion and blocking parameters. We design fused ABMM kernels to shrink workspace and eliminate redundant global-memory traffic, specialize register usage for different primitive patterns to expose intra/inter-kernel parallelism, and introduce a cost-model-based recursion policy together with a Roofline-guided blocking strategy to stabilize performance across problem sizes and shapes. On an NVIDIA A100, experimental results show that 1-level BAG reaches a break-even point against cuBLAS FP32 GEMM at dimension 1280 for square matrices, and recursive BAG achieves a 1.24× speedup over cuBLAS FP32 GEMM at dimension 4096. Our code is available at https://github.com/napleon-liu/bag.

*Corresponding authors.

This work is licensed under a Creative Commons Attribution 4.0 International License.
ICPP '26, Singapore, Singapore
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2657-6/26/09
https://doi.org/10.1145/3832810.3832836

CCS Concepts

• **Computing methodologies** → **Linear algebra algorithms**; *Massively parallel algorithms*; • **Software and its engineering** → **Software performance**.

Keywords

Matrix Multiplication, GEMM, Alternative Basis Matrix Multiplication, GPU Optimization

ACM Reference Format:
Yao Liu, Yewen Li, Zhonghai Zhang, Erlin Yao, Guangming Tan, and Weile Jia. 2026. BAG: Faster Matrix Multiplication on a Single GPU. In *Proceedings of the 55th International Conference on Parallel Processing (ICPP '26), September 28–October 01, 2026, Singapore, Singapore.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3832810.3832836

1 Introduction

General Matrix Multiplication (GEMM) is a cornerstone operation in high-performance computing (HPC) [18–20, 32]. It underpins a wide array of deep learning operators, including attention mechanisms [6, 17, 29, 30] and convolutions [3, 16, 28]. Consequently, the optimization of GEMM remains a critical research priority.

Fast matrix multiplication refers to a class of algorithms that achieve lower asymptotic complexity $O(n^{\omega})$ than the standard $O(n^3)$ approach. Strassen [24] introduced the first sub-cubic algorithm, reducing the computational complexity for square matrices from n^3 to $n^{\log_2 7} \approx n^{2.807}$. Since then, the pursuit of fast matrix multiplication algorithms with even lower exponents has been a highly active area of theoretical computer science, with the current upper bound improved to $\omega < 2.371$ [1, 4, 8, 27]. However, despite their lower theoretical exponents, these algorithms often fail to deliver practical performance gains due to their large hidden constants and the immense matrix dimensions required to realize asymptotic benefits. Indeed, recent work [2] demonstrates that the asymptotically optimal algorithms require such a large n that the leading constant exceeds the estimated number of atoms in the

observable universe. Furthermore, concerns regarding numerical stability and accuracy continue to hinder the adoption of fast matrix multiplication in practical HPC scenarios.

The recently proposed Alternative Basis Matrix Multiplication (ABMM) algorithm shows promise for high performance due to its small leading constant and numerical stability. Strassen's algorithm computes a 2×2 matrix product using 7 multiplications instead of 8, but with 18 additions/subtractions, resulting in arithmetic complexity of $O(7n^{\log_2 7})$. Winograd [31] provided a variant of this algorithm that reduces the number of additions required from 18 to 15, thus reducing the leading coefficient of the arithmetic costs from 7 to 6. Schwartz et al. proposed the ABMM algorithm that reduces the number of additions further from 15 to 12 at the cost of additional basis transformations, thus achieving a better arithmetic cost as $O(5n^{\log_2 7})$ [13, 14]. They also proved the optimality of 12 additions, i.e., cannot be reduced further under Strassen's framework. In [21], Schwartz et al. showed that ABMM has numerical stability comparable to Strassen's algorithm. These properties make ABMM a promising basis for practical GPU optimization.

Schwartz et al. implemented ABMM only on CPUs and demonstrated its advantages over MKL DGEMM as well as Strassen and Winograd variants [22, 23]. On GPUs, however, the key practical question is the break-even size: the smallest matrix dimension at which a sub-cubic algorithm outperforms conventional GEMM. Prior GPU implementations of Strassen-Winograd report a break-even size of 1536 on NVIDIA V100 GPUs [9–11]. In contrast, a naive ABMM implementation on an NVIDIA A100 remains slower than cuBLAS FP32 GEMM at $N = 1024, 2048, 4096$ and is only marginally faster at $N = 8192$, while still moving substantial DRAM traffic (Section 2.4). Since ABMM reduces the number of additions relative to Strassen-Winograd, we investigate whether a GPU-oriented ABMM implementation can turn this algebraic advantage into a lower practical break-even point.

Despite its reduced asymptotic arithmetic count, ABMM is difficult to realize efficiently on GPUs for three reasons:

- **Excessive workspace and memory traffic:** Materializing ABMM's intermediate linear combinations requires 21 temporary matrices per top-level call and incurs 38.8 MB–14.6 GB of DRAM traffic for $N \in [1024, 8192]$ in our naive FP32 profiling.
- **Launch serialization:** Naive ABMM launches ∼21 kernels per top-level call; dependency and write conflicts incur over 10× the launch and synchronization overhead of one cuBLAS call and limit GPU parallelism.
- **Recursion and parameter sensitivity:** Performance is sensitive to recursion cutoffs and blocking parameters, making fixed heuristics unreliable across matrix sizes and shapes.

To address these challenges, this paper presents **BAG** (**B**asis **A**lternative Matrix Multiplication on **G**PUs), a practical and high-performance GPU implementation of ABMM. The main contributions of this paper are as follows:

- **Memory-efficient execution:** We derive a dependency-aware BAG schedule that reduces temporary storage requirements, and fuse basis transformation and intermediate linear combinations into BAG primitives to eliminate unnecessary memory traffic.
- **GPU primitive design:** We formulate ABMM as a set of fused BAG primitives, and optimize their execution through inter-kernel parallelism and compile-time specialization to improve GPU resource utilization.
- **Adaptive control and autotuning:** We develop a cost-model-based recursion policy and a performance-model-guided blocking strategy to adapt BAG to different problem sizes and hardware characteristics.

Under our A100 FP32 evaluation setting, 1-level BAG reaches a crossover point against cuBLAS at dimension 1280, while recursive BAG achieves up to a 1.24× speedup over cuBLAS at dimension 4096 for square matrices.

2 ABMM Background and GPU Challenges

We begin by recapping the ABMM algebraic structure that BAG builds on and identifying where its theoretical savings are forfeited on GPUs. Sections 2.1–2.3 formalize the basis-transformation view and its per-level cost. Section 2.4 then reports a per-call breakdown on A100 that isolates the three concrete sources of the gap—launch count, DRAM traffic, and arithmetic intensity—and directly motivates the design of BAG in Sections 3–5.3.

ABMM is a recursive Strassen-like algorithm. It still uses seven products for each 2×2 block multiplication, but changes basis before and after the fast multiplication so that only 12 additions and subtractions are needed [21]. BAG preserves this algebraic structure: it computes $\tilde{A} = \psi(A)$ and $\tilde{B} = \psi(B)$, multiplies the transformed operands with the seven-product skeleton, and applies ψ^{-1} to recover C. The key GPU design question is where these transformations are evaluated; BAG avoids materializing transformed submatrices in global memory and instead folds the required linear combinations into fused primitives.

2.1 Basis Transformation of Matrix

The basis transformation of matrices A, B and inverse basis transformation of C are both linear transformations, so can be represented by transformation matrices, denoted by ψ and ψ^{-1}, respectively. For simplicity, in the following we only discuss square matrices and assume that the matrix dimension n is a power of 2, i.e., $n = 2^k$. For 2×2 partitioning,

$$A = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix}, B = \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}, C = \begin{pmatrix} C_1 & C_2 \\ C_3 & C_4 \end{pmatrix}, \tag{1}$$

Karstadt and Schwartz [14] gave the optimal basis-transformation matrices for the 2×2 base case. Let ψ denote the basis-transformation matrix:

$$\psi = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 1 \\ 0 & 0 & -1 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix} \quad \psi^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & -1 & 0 & 1 \\ 0 & -1 & 1 & 1 \end{pmatrix} \tag{2}$$

When applying ψ to matrix A, first we need to unroll matrix A to a vector:

$$\psi(A) = \psi\begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} = \psi \times \begin{pmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \end{pmatrix} = \begin{pmatrix} A_1 \\ A_2 - A_3 + A_4 \\ A_4 - A_3 \\ A_2 + A_4 \end{pmatrix} \tag{3}$$

Then the result is reorganized to a 2×2 matrix:

$$\begin{pmatrix} A_1 & A_2 - A_3 + A_4 \\ A_4 - A_3 & A_2 + A_4 \end{pmatrix} = \psi(A) = \tilde{A} \tag{4}$$

The transformation ψ^{-1} on matrix $\tilde{C}$ is defined analogously as:

$$\psi^{-1}(\tilde{C}) = \begin{pmatrix} \tilde{C}_1 & \tilde{C}_2 - \tilde{C}_3 \\ \tilde{C}_4 - \tilde{C}_2 & \tilde{C}_3 + \tilde{C}_4 - \tilde{C}_2 \end{pmatrix} = C \tag{5}$$

2.2 Fast Matrix Multiplication in ABMM

On the transformed matrices $\tilde{A}$ and $\tilde{B}$, ABMM follows the Strassen seven-product skeleton but reduces the additions/subtractions from 18 to 12. With transformed blocks denoted as A_i and B_i, the seven products are $M_1 = A_4B_4$, $M_2 = A_3B_3$, $M_3 = A_2B_2$, $M_4 = A_1B_1$, $M_5 = (A_2 - A_3)(B_4 - B_2)$, $M_6 = (A_2 - A_1)(B_2 - B_3)$, and $M_7 = (A_4 - A_2)(B_2 - B_1)$. The outputs are recovered as $C_1 = M_5 + M_4$, $C_2 = M_3 + M_5 - M_6 + M_7$, $C_3 = M_2 + M_7$, and $C_4 = M_1 - M_6$. Recursion terminates either at scalar subproblems or at a cutoff selected statically or dynamically by the implementation.

2.3 Algorithmic Advantages of ABMM

Schwartz et al. [14] proved the optimality of ABMM's 12 additions: the number cannot be reduced further under Strassen's framework. Table 1 compares ABMM with other Strassen-like algorithms, including Strassen [24] and Strassen-Winograd [31]. The table keeps only high-order arithmetic and I/O terms. Additions denote the number of additions and subtractions needed to multiply two 2×2 matrices; n is the matrix dimension and M is memory size.

Table 1: Complexity comparison of Strassen-like algorithms.

Algorithm	Additions	Arithmetic	I/O Complexity
Strassen [24]	18	$7n^{\log_2 7}$	$6(\sqrt{3}n/\sqrt{M})^{\log_2 7}M$
Winograd [31]	15	$6n^{\log_2 7}$	$5(\sqrt{3}n/\sqrt{M})^{\log_2 7}M$
ABMM [14]	12	$5n^{\log_2 7}$	$4(\sqrt{3}n/\sqrt{M})^{\log_2 7}M$

Table 1 shows that ABMM reduces the leading arithmetic term by 16.6% and the leading I/O term by 20% relative to Strassen-Winograd. These advantages motivate us to study whether ABMM can be made practical on modern GPUs.

2.4 GPU Challenges for ABMM

The arithmetic and I/O complexity of ABMM are both lower than those of the direct algorithm, yet translating this algebraic advantage into GPU throughput is far from automatic. A naive FP32 implementation that explicitly materializes intermediate submatrices and invokes separate kernels for additions and multiplications reaches only 0.391×, 0.844×, and 0.931× the throughput of cuBLAS FP32 GEMM at $N = 1024, 2048, 4096$, respectively, and becomes only 1.067× faster at $N = 8192$ (Figure 1). The remainder of this section dissects why this marginal crossover is insufficient and distills the three optimization targets addressed by BAG.

Figure 1 reports a per-call breakdown of cuBLAS FP32 GEMM and a naive FP32 ABMM implementation on the A100. Effective TFLOPS is computed from the dense GEMM count $2N^3$ for both methods. The naive schedule dispatches **21 kernel launches per top-level call** (two element-wise additions and one GEMM for each of the seven primitives), inflating launch and synchronization overhead by more than an order of magnitude relative to cuBLAS's single launch. Each intermediate linear combination is explicitly materialized, so every element of A and B is read once for transformation and again for GEMM while primitive outputs are written and re-read during accumulation. The resulting naive-ABMM DRAM traffic grows from 38.8 MB at $N = 1024$ to 14.6 GB at $N = 8192$. Across the evaluated sizes, the separate-kernel baseline transfers 3.59–4.33× the cuBLAS DRAM traffic while still reaching only 0.931× cuBLAS throughput at $N = 4096$.

The bottleneck, therefore, is not peak DRAM bandwidth (dense GEMM is compute-bound on either side) but the *ratio* of memory traffic to useful computation together with launch serialization. This diagnosis maps each BAG design decision in Sections 4.1–4.2 to a concrete source of the gap: fusing basis transformation and output accumulation into the GEMM kernel removes the materialization traffic quantified above; collapsing the 21 launches into a handful of fused primitive invocations attacks the launch overhead; and exposing inter-kernel parallelism lets the remaining launches overlap rather than serialize.

Beyond the launch-and-traffic gap, naive ABMM is also highly sensitive to recursion depth and blocking parameters, motivating the adaptive control in Section 5.1. At $N = 4096$, a fixed-depth naive ABMM sweeps from 0.48× cuBLAS throughput at depth 1 down to 0.29× at depth 4: each additional level halves the subproblem but also multiplies launch count by seven and shrinks arithmetic intensity, so a depth chosen without awareness of the current subproblem size degrades rapidly once tiles become too small to saturate the SMs. The same effect appears in blocking: across the seven candidate strategies in Table 4, naive ABMM throughput at $N = 4096$ varies by 2.3× between the best and worst strategy, and the best strategy itself shifts between *Huge* and *L2-Huge* as N moves from 2048 to 8192. Fixed heuristics therefore cannot deliver robust speedups, which is why BAG pairs its fused primitives with the cost-model-based recursion policy and the Roofline-guided blocking strategy described in Section 5.3.

3 Memory-Efficient Execution for BAG

A direct implementation of ABMM allocates fresh buffers for the seven submatrix products $M_1, \ldots, M_7$ and materializes every input linear combination in global memory, inflating workspace and DRAM traffic by the factors quantified in Section 2.4. BAG eliminates both sources of overhead in one pass: a dependency-aware schedule (Section 3.1) overwrites each buffer as soon as its producers retire, while the remaining basis transformations are pushed into the prologue and epilogue of the GEMM kernel (Section 3.2) so that no intermediate matrix is ever written back to global memory.

As shown in the previous section, conventional ABMM implementations require 21 temporary matrices for the intermediate S

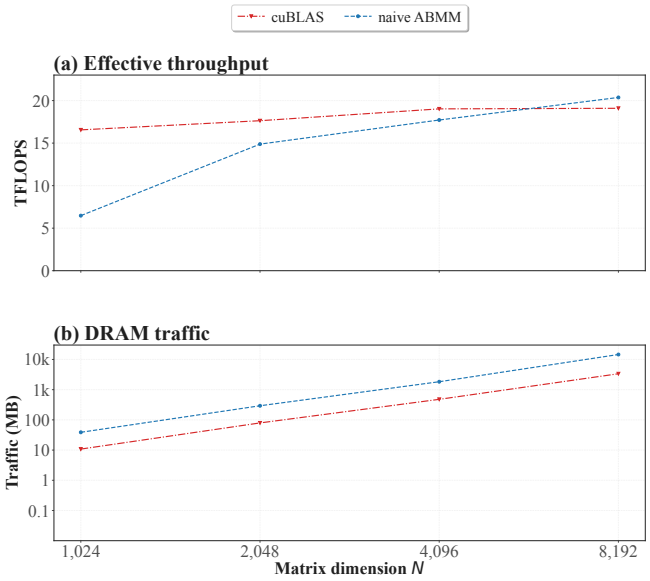


Figure 1: FP32 throughput and DRAM traffic of cuBLAS GEMM and naive ABMM on the A100. TFLOPS uses the dense GEMM count $2N^3$, and DRAM traffic is measured with Nsight Compute.

Table 2: Logical two-buffer dependency schedule of BAG. The T_1 and T_2 entries denote tile-wise linear combinations in the fused primitive rather than materialized global-memory matrices.

Step	Operation	Description
1	$T_1 \leftarrow A_4 - A_2$	Compute S_7
2	$T_2 \leftarrow B_2 - B_1$	Compute T_7
3	$C_2 \leftarrow T_1 \times T_2$	Compute M_7
4	$C_3 \leftarrow C_3 + C_2$	C_3 += M_7
5	$T_1 \leftarrow A_2 - A_1$	Compute S_6
6	$T_2 \leftarrow B_2 - B_3$	Compute T_6
7	$C_4 \leftarrow T_1 \times T_2$	Compute M_6
8	$C_2 \leftarrow C_2 - C_4$	C_2 -= M_6
9	$T_1 \leftarrow A_2 - A_3$	Compute S_5
10	$T_2 \leftarrow B_4 - B_2$	Compute T_5
11	$C_1 \leftarrow T_1 \times T_2$	Compute M_5
12	$C_2 \leftarrow C_2 + C_1$	C_2 += M_5
13	$C_4 \leftarrow A_4 \times B_4 - C_4$	$C_4 = M_1 - M_6$
14	$C_3 \leftarrow C_3 + A_3 \times B_3$	C_3 += M_2
15	$C_2 \leftarrow C_2 + A_2 \times B_2$	C_2 += M_3
16	$C_1 \leftarrow C_1 + A_1 \times B_1$	C_1 += M_4

and T series used by Strassen-like algorithms [7]. This memory footprint not only exhausts limited device memory but also introduces synchronization bottlenecks that hinder massive parallelism on GPUs. BAG addresses this challenge by combining a dependency-aware schedule with kernel-level fusion to shorten the lifetime of intermediate data and reduce global-memory traffic.

3.1 Memory-Efficient Scheduling via Dependency Analysis

The ABMM algorithm decomposes a matrix multiplication $C = A \times B$ into seven sub-matrix products $M_1, \ldots, M_7$, each computed from linear combinations of submatrices of A and B. Traditional implementations, such as those by Douglas et al. [7], explicitly materialize all intermediate linear combinations into temporary matrices before invoking multiplication kernels, requiring 21 temporary matrices in total. While Boyer et al. [5] proposed schemes to eliminate temporary storage by overwriting input matrices or increasing operation counts, such methods introduce write-after-read hazards that necessitate fine-grained synchronization, significantly degrading GPU performance.

By exploiting the commutative and associative properties of matrix addition, we can reorder sub-matrix multiplications and accumulations while respecting dependencies. Building on Lai et al. [15], Table 2 presents a logical two-buffer schedule: T_1 and T_2 denote A-side and B-side linear combinations, not allocated matrices. In fused BAG, their values are constructed tile by tile in the primitive prologue and retained only in shared memory and registers.

If materialized, the logical two-buffer schedule would require $mk + kn$ FP32 bytes per level, whereas the 21-temporary baseline requires substantially more. This comparison does not describe BAG's allocated workspace: BAG forms each combination tile by tile, consumes it in the GEMM pipeline, and introduces no full-matrix global-memory workspace beyond the outputs. The schedule also exposes inter-kernel parallelism (Section 4.2).

3.2 Fused Basis Transformation

As discussed in the background section, the ABMM algorithm requires basis transformation $\psi(\cdot)$ as a preprocessing step to construct composite submatrices from input matrices A and B. For an $m \times k$ matrix A partitioned into four block submatrices $A_{11}, A_{12}, A_{21}, A_{22}$, the transformation $\psi(A)$ constructs four composite submatrices (e.g., $A_1 = A_{11}$, $A_2 = A_{12}$, $A_3 = A_{21}$, $A_4 = A_{22}$) that serve as operands for the seven BAG primitives.

A naive implementation would compute $\psi(A)$ and $\psi(B)$ in separate preprocessing kernels, materializing all composite submatrices in global memory before invoking the BAG primitives. This approach doubles the memory bandwidth requirement: each element of A and B is read once during transformation and read again during GEMM computation.

To eliminate this redundancy, we integrate the basis transformation directly into the fused BAG primitive kernel. Operand construction is performed inside the tiled GEMM pipeline: the kernel stages the participating source tiles, forms $(X + \delta Y)$ or $(V + \varepsilon W)$ with FP32 add/subtract instructions, multiplies the resulting tiles, and accumulates the product into one or two output submatrices. CUDA `cp.async` is used only for global-to-shared prefetching; arithmetic remains in FP32 instructions. Compile-time predicates (`if constexpr`) remove unused sources when $\delta = 0$ or $\varepsilon = 0$, reducing memory traffic and register pressure.

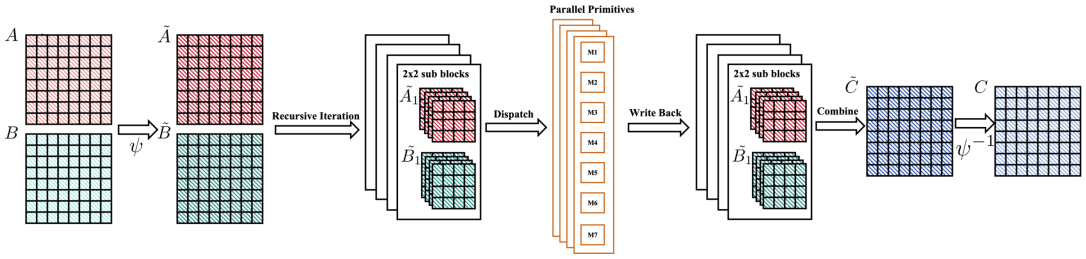


Figure 2: Overview of the BAG framework showing the key optimization components and their interactions.

4 GPU-Oriented Primitive Design and Parallel Execution

4.1 Fused BAG Primitives

The execution schedule in Table 2 reveals that the seven sub-matrix multiplications $M_1, \ldots, M_7$ follow a common computational pattern: each requires forming linear combinations of input submatrices, performing a matrix multiplication, and accumulating the result into multiple output submatrices with different scaling factors. We abstract this pattern as a *BAG primitive*:

$$M = (X + \delta Y)(V + \varepsilon W); \quad D \leftarrow D + \gamma_1 M; \quad E \leftarrow E + \gamma_2 M, \quad (6)$$

where X, Y are submatrices of A (or temporary matrix T_1), V, W are submatrices of B (or T_2), D, E are output submatrices, and $\delta, \varepsilon \in \{-1, 0, 1\}$ control input linear combinations while $\gamma_1, \gamma_2 \in \{-1, 0, 1\}$ control output accumulation.

A naive implementation would decompose Equation 6 into three separate kernel launches: (1) compute $P = X + \delta Y$ and $Q = V + \varepsilon W$, writing them to global memory; (2) invoke cuBLAS to compute $M = P \times Q$; and (3) launch element-wise kernels to accumulate M into D and E. This approach incurs $2mn + 2mk + 2kn$ extra memory traffic for the intermediate matrices P and Q, significantly degrading bandwidth efficiency.

To eliminate this overhead, we design a *fused BAG primitive* kernel that integrates all three stages—input linear combination, GEMM computation, and output accumulation—into a single kernel launch. Figure 3 illustrates the architecture of this fused primitive in comparison to a standard CUTLASS GEMM kernel, and Algorithm 1 details its execution flow.

4.2 Inter-Kernel Parallelism

A straightforward implementation of BAG launches the seven primitive kernels sequentially, relying solely on intra-kernel parallelism across thread blocks, warps, and threads. While this approach works well for large matrices where each kernel fully saturates the GPU, it underutilizes hardware resources for small to medium problem sizes ($m = n = k \leq 6{,}000$) and during recursive decomposition, where individual kernels cannot generate sufficient work to occupy all SMs.

To improve GPU utilization, we exploit inter-kernel parallelism by launching multiple independent primitives concurrently. As shown in Table 2, the seven primitives exhibit output dependencies: multiple primitives may accumulate into the same output submatrix (e.g., M_5 and M_3 both write to C_2). However, as long as we resolve write conflicts, primitives without output dependencies can execute in parallel.

Algorithm 1 Fused BAG primitive kernel for computing $M = (X + \delta Y)(V + \varepsilon W)$ with accumulation to output matrices D and E using coefficients γ_1 and γ_2.

1: **Registers**: $vec_A[2][m_R]$, $vec_B[2][n_R]$, $res[m_R \times n_R]$
2: **Shared Memory**: $tile_A[k_S \times m_S]$, $tile_B[k_S \times n_S]$
3: Load summed tiles $(X + \delta Y) \rightarrow tile_A$ and $(V + \varepsilon W) \rightarrow tile_B$
4: Initialize $res \leftarrow 0$; load initial subvectors into $vec_A[0]$ and $vec_B[0]$
5: **for** $block_k = 0$ **to** K **step** k_S **do**
6: Prefetch eligible source tiles from global memory to shared memory
7: **for** $warp_k = 0$ **to** $k_S - 1$ **do**
8: Load next subvectors from shared memory into vec_A and vec_B
9: $res \leftarrow res + vec_A[warp_k \,\%\, 2] \times vec_B[warp_k \,\%\, 2]$
10: **end for**
11: Form required source-tile sums in shared memory using FP32 add/subtract instructions
12: `__syncthreads()`
13: **end for**
14: Write back res to global memory: $D \leftarrow D + \gamma_1 \cdot res$ and $E \leftarrow E + \gamma_2 \cdot res$

Scheme 1: Stage-Based Parallelism. We partition the seven primitives into four synchronization stages based on the dependency structure revealed in Table 2. Table 3 shows this partitioning: Stage 0 launches M_1, M_2, M_3, M_4 concurrently on four CUDA streams, as they write to disjoint output submatrices (C_1, C_2, C_3, C_4). Stages 1–3 execute M_5, M_6, M_7 sequentially, as each depends on the output of the previous stage. This scheme exposes moderate parallelism (4-way at Stage 0) without requiring atomic operations.

Scheme 2: Atomic-Based Full Parallelism. While Scheme 1 works well for large matrices, it underutilizes the GPU for small to medium problem sizes where 4-way parallelism is insufficient. To expose maximal parallelism, we employ a second scheme that resolves write conflicts at a finer granularity using atomic operations. Specifically, we replace the conventional accumulation operations in the BAG primitive epilogue with `atomicAdd` instructions for global memory writes. This allows all seven primitives to execute concurrently across seven CUDA streams, achieving roughly 1.75×

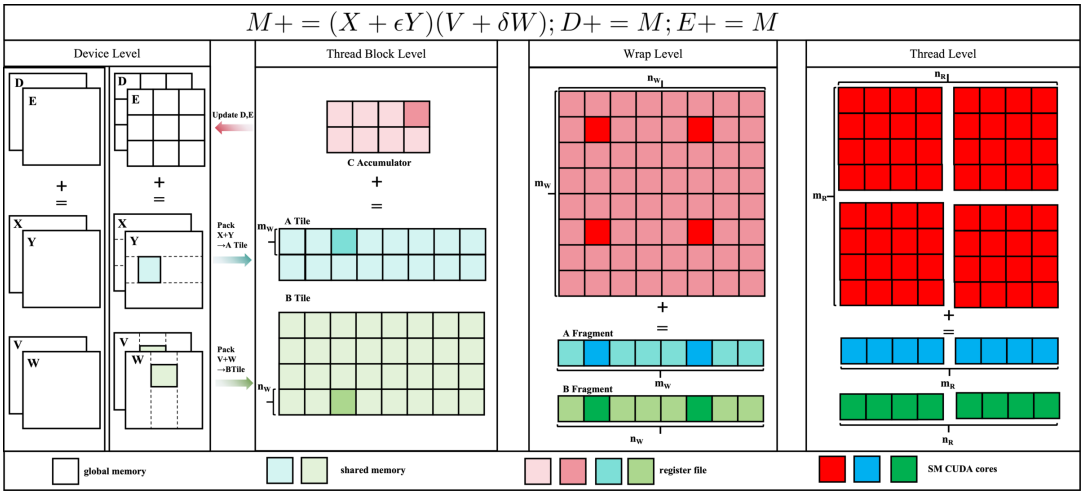


Figure 3: Architecture of the BAG primitive for fused matrix multiplication with operand-level linear combinations.

Table 3: Parallel execution stages of the BAG algorithm with four-stage scheduling across seven primitive instances.

Stage	Operation	Stream
0	① $M_1 = A_4B_4$; C_4 += M_1	[0]
	② $M_2 = A_3B_3$; C_3 += M_2	[1]
	③ $M_3 = A_2B_2$; C_2 += M_3	[2]
	④ $M_4 = A_1B_1$; C_1 += M_4	[3]
1	⑤ $M_5 = (A_2 - A_3)(B_4 - B_2)$; C_1 += M_5; C_2 += M_5	[0]
2	⑥ $M_6 = (A_2 - A_1)(B_2 - B_3)$; C_2 −= M_6; C_4 −= M_6	[0]
3	⑦ $M_7 = (A_4 - A_2)(B_2 - B_1)$; C_2 += M_7; C_3 += M_7	[0]

higher parallelism than Scheme 1. The overhead of atomic operations is amortized by improved SM occupancy, particularly at small matrix sizes where arithmetic intensity is low and memory latency dominates execution time.

4.3 Compile-Time Register Optimization

The fused BAG primitive (Equation 6) is parameterized by four coefficients $\delta, \varepsilon, \gamma_1, \gamma_2 \in \{-1, 0, 1\}$ that control input linear combinations and output accumulations. When any coefficient equals zero, the corresponding operation becomes redundant: $\delta = 0$ eliminates the need to fetch Y and compute $X + \delta Y$; $\varepsilon = 0$ similarly eliminates W; $\gamma_1 = 0$ or $\gamma_2 = 0$ skips accumulation to D or E, respectively. Exploiting these zero coefficients can significantly reduce register pressure and memory traffic.

Examining the seven primitives in Table 2, we observe that $M_1 - M_4$ (steps 13–16) all have the form $M_i = A_i \times B_i$ with a single output accumulation, corresponding to $\delta = \varepsilon = \gamma_1 = 0$ and $\gamma_2 = \pm 1$. In contrast, M_5, M_6, M_7 (steps 11, 7, 3) involve multi-term input combinations and dual output accumulations. Based on this observation, we classify the seven primitives into two variants:

Variant	Predicate Pattern and Instances
Var#0	$\delta = 0,\ \varepsilon = 0,\ \gamma_1 = 0, \gamma_2 = \pm 1$; applies to M_1, M_2, M_3, M_4
Var#1	$\delta = \pm 1,\ \varepsilon = \pm 1,\ \gamma_1 = \pm 1, \gamma_2 = \pm 1$; applies to M_5, M_6, M_7

We implement each variant as a separate CUDA kernel template instantiated with the coefficients as non-type template parameters. Within the kernel, we use C++17's `if constexpr` to conditionally compile different code paths based on coefficient values. The CUDA compiler performs dead code elimination (DCE) during PTX and SASS generation, removing unreachable branches and unused register allocations. As a result, Var#0 primitives (M_1–M_4) consume the same number of registers as a standard cuBLAS GEMM kernel, while Var#1 primitives (M_5–M_7) require additional registers only for the extra operand Y (or W) and the second output accumulator. This compile-time specialization eliminates warp divergence and reduces instruction cache footprint compared to a unified runtime-branching implementation. To quantify the register benefit, we measure per-thread register usage via `cuobjdump -dump-resource-usage` on SM80 binaries. A cuBLAS FP32 GEMM kernel at the same tile configuration uses 128 registers per thread; BAG Var#0 (M_1–M_4) matches this at 128 registers, incurring no additional pressure, while Var#1 (M_5–M_7) uses 168 registers to accommodate the extra operand Y (or W) and the second output accumulator. A unified runtime-branching implementation compiled from the same source would instead allocate registers for the worst-case path on *all* seven primitives, inflating Var#0 usage by over 30% and reducing achievable occupancy from 4 to 3 blocks per SM on the A100.

5 Adaptive Control and Autotuning

The practical benefit of ABMM hinges on two design choices that no fixed heuristic can set optimally across problem sizes: how deeply to recurse, and which blocking parameters to use at each level. BAG resolves both with cost-driven policies. A Roofline-based cost model

compares the estimated time of descending one more ABMM level against invoking the direct BAG primitive or cuBLAS, and selects the cheaper branch per subproblem (Section 5.1); a lightweight padding scheme extends this decision to non-power-of-two shapes without wasting computation (Section 5.2); and a performance-model-guided search prunes the blocking space to a single winning tile per recursion level (Section 5.3).

5.1 Adaptive Recursion Control

ABMM reduces arithmetic complexity only when the savings from replacing eight submatrix multiplications with seven outweigh the overhead of basis transformation, extra linear combinations, and additional kernel launches. On GPUs, this balance depends strongly on the current subproblem shape and on whether the primitive kernels can still maintain sufficient occupancy. Consequently, BAG does not rely on a fixed recursion depth. Instead, we use a cost-model-based recursion policy that compares the estimated execution cost of one additional ABMM level against the cost of terminating recursion and invoking the direct BAG primitive or conventional GEMM.

For a subproblem of shape (m, n, k), the recursion policy evaluates two alternatives. The direct path estimates the cost of computing the subproblem without further decomposition, using the performance model of the selected primitive configuration (Section 5.3). The recursive path estimates the cost of performing basis transformation, launching the seven child multiplications, and executing the associated output accumulation and synchronization steps.

Let $T_{\text{prim}}(m, n, k)$ be the primitive-kernel time predicted by the Roofline model under the best tile strategy for shape (m, n, k). The direct-path cost is

$$T_{\text{dir}}(m, n, k) \;=\; T_{\text{prim}}(m, n, k) \;+\; T_{\text{launch}}, \tag{7}$$

where T_{launch} is the per-call kernel-launch overhead calibrated once on the target GPU (measured as $\approx 4.7\,\mu s$ on the A100). The recursive-path cost aggregates the seven child multiplications, the input/output linear-combination traffic, and the fixed per-level launch and synchronization overheads:

$$\begin{aligned} T_{\text{rec}}(m, n, k) \;=\; & \sum_{i=1}^{7} T_{\text{dir}}\!\left(\tfrac{m}{2}, \tfrac{n}{2}, \tfrac{k}{2}\right) \\ & + T_{\psi}(m, n, k) + T_{\psi^{-1}}(m, n, k) \\ & + N_{\text{stage}} \cdot T_{\text{sync}}, \end{aligned} \tag{8}$$

where T_{ψ} and $T_{\psi^{-1}}$ are the basis-transformation times bounded by the global-memory-bandwidth term of the Roofline model, $N_{\text{stage}} = 4$ is the number of synchronization stages from Table 3, and T_{sync} is the measured stream-synchronization latency ($\approx 2.8\,\mu s$ on the A100). Under the fused primitive, T_{ψ} and $T_{\psi^{-1}}$ collapse into the prologue and epilogue of the child kernels and are charged only at the top recursion level.

Recursion proceeds only when

$$T_{\text{rec}}(m, n, k) \;<\; T_{\text{dir}}(m, n, k) \quad \text{and} \quad \min(m, n, k)/2 \;\geq\; N_{\text{min}}, \tag{9}$$

where N_{min} is the smallest dimension at which a single primitive kernel still saturates the A100's SMs (empirically $N_{\text{min}} = 512$ for square tiles under the Huge strategy). The second condition prevents BAG from over-recursing on moderate problem sizes, where shrinking arithmetic intensity would negate the asymptotic advantage of ABMM. This policy is evaluated at every recursion level on the current subproblem shape, so BAG terminates adaptively rather than at a global fixed depth. This adaptive termination directly addresses the 1.4–2.1× fixed-depth variability documented in Section 2.4.

In practice, the policy favors recursion for large square matrices, while terminating earlier for shapes whose subproblems would become too small or too imbalanced after partitioning. This behavior is especially important near the break-even region, where a fixed cutoff can easily recurse one level too deep and lose the benefit of reduced arithmetic count. By selecting recursion depth adaptively, BAG preserves the asymptotic advantage of ABMM on large problems while avoiding unnecessary decomposition overhead on smaller ones.

5.2 Handling Non-Power-of-Two Dimensions

Recursive ABMM decomposition divides matrix dimensions by two at each level, producing odd-dimensional submatrices when the original dimensions are not powers of two. Traditional approaches handle these "fringes" through padding (extending matrices to even dimensions with zero rows/columns [12]) or peeling (extracting odd rows/columns and handling them separately [26]). Padding introduces redundant computation on zero elements, while peeling requires separate fix-up GEMM calls, increasing kernel launch overhead.

We adopt an integrated approach that handles fringes within the fused BAG primitive kernel without additional memory allocation or separate fix-up kernels. When loading tiles from global memory into shared memory, we pad incomplete tiles (those at the fringes of odd-dimensional submatrices) with zeros directly in the shared memory buffer. Similarly, when accumulating results back to global memory, we use predicated stores to avoid writing padding elements. This strategy leverages the fact that tile packing and accumulation are already performed in high-performance GEMM implementations; we extend the existing logic with boundary checks that are confined to boundary tiles, limiting the overhead to the edges of the computation.

5.3 Performance Modeling and Autotuning

The performance of the fused BAG primitive depends critically on blocking parameters—the tile sizes at thread-block level (m_S, n_S, k_S), warp level (m_W, n_W), and thread level (m_R, n_R)—which determine register usage, shared memory consumption, and arithmetic intensity. To systematically select these parameters for the NVIDIA A100 architecture, we develop a Roofline-based performance model and use it to guide autotuning.

Performance Model. We model the execution time of a thread block as $T = \max(T_{\text{flop}}, T_{\text{smop}}, T_{\text{gmop}})$, covering arithmetic throughput, shared-memory bandwidth, and global-memory bandwidth. The flop term adds primitive overheads to GEMM work as $N_{\text{flop}} = N^{\times}_{\text{flop}} + N^{+}_{\text{flop}}(A) + N^{+}_{\text{flop}}(B) + N^{+}_{\text{flop}}(C)$, where $N^{\times}_{\text{flop}} = 2m_S n_S k$ and the $N^{+}_{\text{flop}}(\cdot)$ terms account for input/output linear combinations. Memory terms are counted in bytes. We use A100 FP32 peak throughput (19.5 TFLOPS), global-memory bandwidth (2.04 TB/s), aggregate

shared-memory bandwidth (19.5 TB/s), and up to 8 active thread blocks per SM.

Blocking Parameter Selection. Similar to Tan et al. [25] and Zhang et al. [33], we select blocking parameters by analyzing register capacity, shared-memory capacity, L2 reuse, and memory bandwidth. Table 4 summarizes the candidate strategies; BAG selects among them according to the current subproblem shape and recursion level rather than using one globally fixed configuration.

For large matrices on the A100, the model first filters candidates that satisfy the global-memory bandwidth bound ($m_S, n_S \geq 58.2$) and remain feasible under the selected shared-memory buffering mode. It then trades off block-level occupancy against arithmetic intensity and L2 locality. The $128 \times 128 \times 8$ Huge strategy is the default high-occupancy configuration for most large subproblems, while the $256 \times 256 \times 32$ L2-Huge strategy is used only for top-level or sufficiently large square subproblems whose working set benefits from higher L2 reuse. This distinction avoids using a single tile size for both top-level and recursively generated subproblems.

Register usage per thread remains strictly bounded. Each thread requires $m_R \times n_R$ registers for accumulators, plus additional registers for prefetching and double buffering. Given the 255-register limit, we use $m_R = n_R = 8$ for most large-square BAG variants, while smaller or rectangular strategies use reduced register tiles to preserve occupancy.

Table 4: Candidate blocking strategies considered by the BAG autotuner on the NVIDIA A100 GPU architecture.

Strategy	m_S	n_S	k_S	m_W	n_W	m_R	n_R
Small	16	16	16	8	16	2	2
Medium	32	32	8	16	32	4	4
Large	64	64	8	32	64	8	8
Huge	128	128	8	32	64	8	8
L2-Huge	256	256	32	64	64	8	8
Tall	128	32	8	64	16	8	4
Wide	32	128	8	16	64	4	8

6 Experimental Evaluation

We evaluate BAG on an NVIDIA A100 GPU (80GB, PCIe) running CUDA 12.4, comparing against cuBLAS v12.4 and CUTLASS v4.3.0. The end-to-end, numerical-stability, and ablation experiments in this section use IEEE FP32 precision; TF32 is disabled for all implementations, and cuBLAS uses the `CUBLAS_COMPUTE_32F` compute path. Input matrices are drawn from a uniform distribution $\mathcal{U}(-1, 1)$. We report the median performance over 10 runs after 3 warm-up iterations to ensure stable measurements. Unless otherwise stated, reported throughput is computed using the conventional GEMM operation count $2mnk$ divided by the measured end-to-end runtime; this user-visible TFLOPS metric is used for cuBLAS, CUTLASS, naive ABMM, and BAG, rather than counting the reduced number of arithmetic operations executed internally by the fast algorithms. The evaluation covers three aspects: end-to-end performance, numerical stability, and ablation studies to isolate the contribution of individual optimizations.

6.1 End-to-End Performance

We benchmark BAG on three representative matrix shapes: square ($m = n = k$), wide ($m = n = 4096, k$ varying), and tall ($m = n$ varying, $k = 4096$). These cases cover common computational patterns in scientific computing and deep learning.

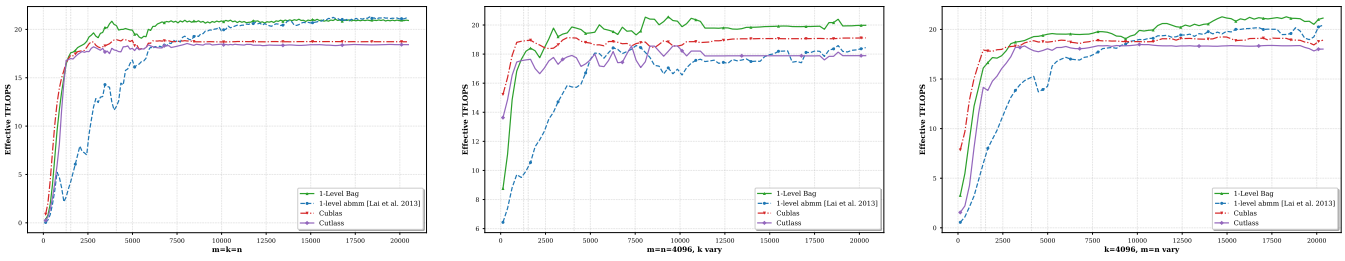
Single-Level BAG Performance. Figure 4 shows the performance of 1-level BAG (non-recursive) across the three matrix shapes. For square matrices, BAG reaches a break-even point at dimension 1280 against cuBLAS FP32 GEMM under our A100 evaluation setting. Prior Strassen-Winograd GPU work reported a break-even point of 1536 on NVIDIA V100 GPUs [11]; because this result was obtained on a different hardware and software stack, we use it only as contextual reference rather than a direct head-to-head comparison. The earlier crossover relative to our naive ABMM baseline is enabled by the fused kernel design: by folding the 12 intermediate linear combinations of Table 2 into the GEMM pipeline, BAG removes the explicit materialization traffic quantified in Figure 1 and collapses the 21 separate naive launches into a small number of fused primitive invocations, allowing the computational savings of ABMM to manifest at smaller problem sizes.

Recursive Scalability (N-level). Figure 5 presents the fully recursive BAG algorithm with adaptive recursion control (Section 5.1). By exploiting ABMM's sub-cubic complexity ($O(n^{2.807})$), recursive BAG reaches a 1.24× speedup over cuBLAS at dimension 4096 for square matrices. This confirms that ABMM's asymptotic advantage can translate to measurable GPU performance gains when recursion is properly controlled.

Non-square Behavior. The wide and tall settings in Figures 4 and 5 show less monotonic behavior than the square case because one dimension becomes the limiting factor after ABMM partitioning. In the wide setting ($m = n = 4096$ with varying k), small and medium k values shorten the reduction loop of each child GEMM, so the fused operand-combination loads, epilogue accumulation, and kernel-launch overhead are amortized over fewer multiply-add operations. In the tall setting ($k = 4096$ with varying $m = n$), the left side of the curves has too few output tiles to fully occupy all SMs after the subproblems are halved. As the varying dimension grows, the rectangular Wide/Tall blocking strategies in Table 4 recover enough arithmetic intensity and parallelism for BAG to overtake cuBLAS. For recursive BAG, the cost model often terminates earlier on these rectangular subproblems when a child shape would fall below $N_{\min}$ or lose occupancy; the vertical transition points in the plots therefore correspond to changes in selected recursion depth or blocking strategy, explaining the observed fluctuations relative to the smoother square-matrix trend.

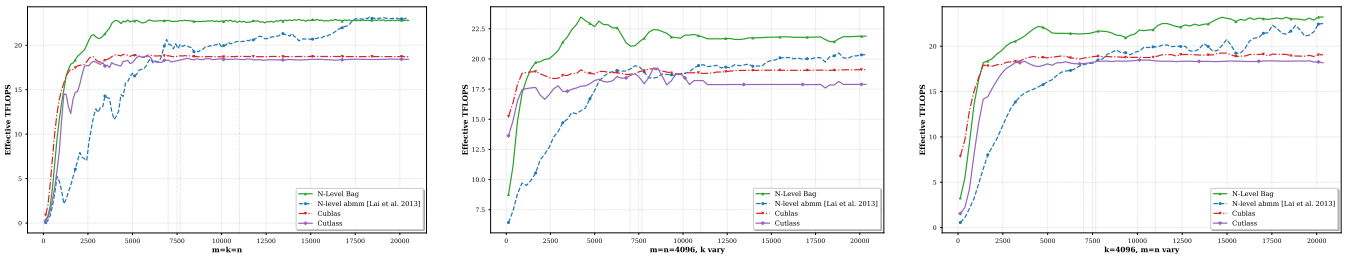
6.2 Numerical Stability

We evaluate BAG's numerical stability relative to a high-precision reference and compare it with cuBLAS FP32 GEMM. For each square size $n \in \{256, 512, 1024, 2048, 4096, 8192\}$, we use 20 independently seeded uniform inputs. Let $\widehat{C}_X$ denote the FP32 output produced by method X, and let C_{ref} denote the FP64 DGEMM reference product computed from the same FP32 inputs. We report the relative



(a) Square matrices ($m = n = k$) **(b) Wide setting ($m = n = 4096$, k varies)** **(c) Tall setting ($k = 4096$, $m = n$ varies)**

Figure 4: Performance comparison of 1-level BAG against cuBLAS and CUTLASS libraries across different matrix shapes.



(a) Square matrices ($m = n = k$) **(b) Wide setting ($m = n = 4096$, k varies)** **(c) Tall setting ($k = 4096$, $m = n$ varies)**

Figure 5: Performance comparison of N-level recursive BAG against cuBLAS across different matrix shapes.

Frobenius error

$$E_F(X) = \frac{\|\widehat{C}_X - C_{\text{ref}}\|_F}{\|C_{\text{ref}}\|_F}, \tag{10}$$

For each matched input seed, we also report the paired ratio

$$R_F(X) = \frac{E_F(X)}{E_F(\text{cuBLAS})}. \tag{11}$$

The adaptive curve follows the size-dependent depth schedule recorded by this stability sweep; it is used to expose error accumulation and is distinct from the performance-optimal cutoff evaluated in Section 5.1.

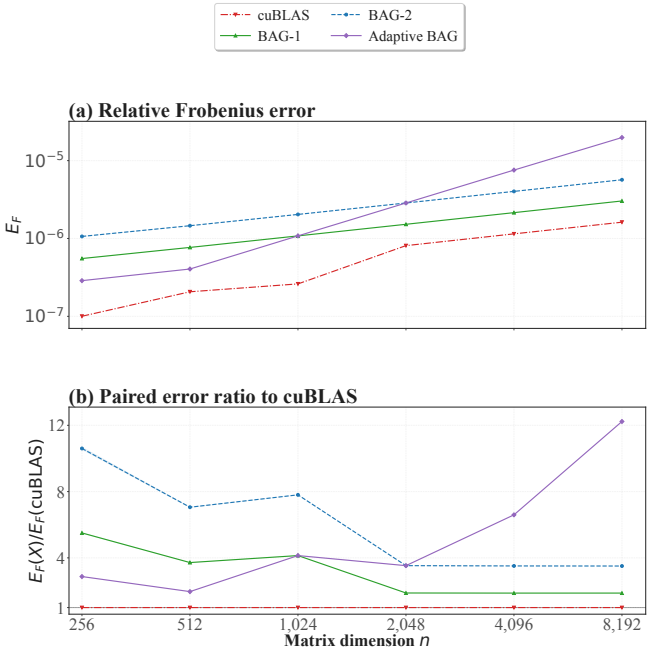


Figure 6: Numerical stability in FP32 over 20 input seeds. Lines show medians and shaded bands show the 25th–75th percentiles. BAG-1 and BAG-2 use fixed recursion depths; the adaptive sweep selects depths 0, 0, 1, 2, 3, 4 as n increases. (a) Relative Frobenius error. (b) Paired error ratio to cuBLAS.

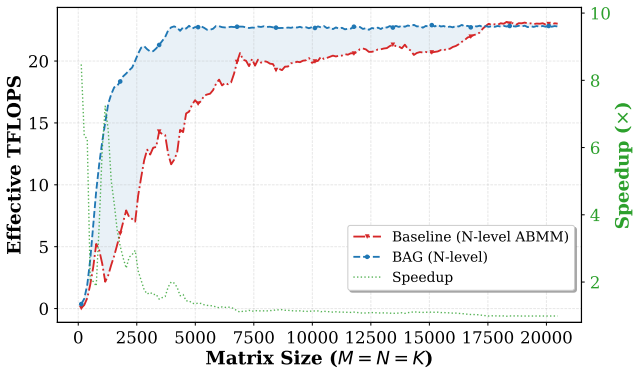
Figure 6 shows a controlled accuracy–depth tradeoff. The median E_F of BAG-1 increases from 5.53×10^{-7} at $n = 256$ to 3.03×10^{-6} at $n = 8192$, while BAG-2 increases from 1.06×10^{-6} to 5.68×10^{-6}; the corresponding cuBLAS medians are 1.00×10^{-7} and 1.62×10^{-6}. Across all six sizes, the paired median ratio ranges are 1.87–5.50× for BAG-1 and 3.51–10.60× for BAG-2. For the larger cases ($n \geq 2048$), these ranges narrow to 1.87–1.88× and 3.51–3.53×, respectively; the larger ratios at small n partly reflect the smaller cuBLAS denominator. The adaptive-depth variant reaches $E_F = 1.98 \times 10^{-5}$, or 12.23× cuBLAS, at depth 4 and $n = 8192$. The interquartile width is at most 2.14% of the median over the plotted method–size pairs, indicating that these trends are consistent across seeds. Thus, additional recursion predictably compounds rounding error while keeping the absolute relative error small for the tested uniform inputs.

6.3 Ablation Study

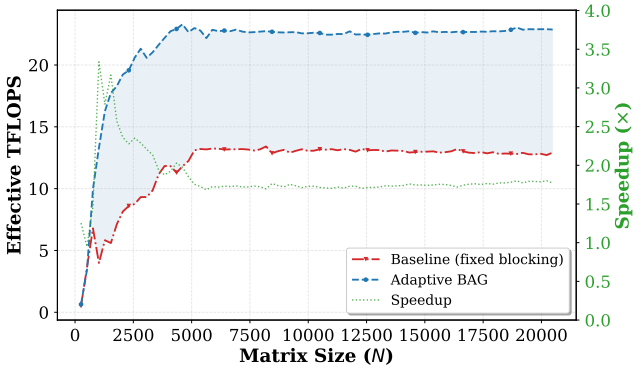
To isolate and quantify the contribution of individual optimizations, we conduct a systematic ablation study comparing BAG against two baseline implementations: (1) a naive ABMM implementation that uses separate kernels for linear combinations and GEMM, and (2) a BAG variant with fixed tiling parameters instead of adaptive autotuning.

Impact of Fused Primitives. Figure 7a compares BAG against the naive ABMM implementation characterized in Figure 1, which decomposes each BAG primitive (Equation 6) into three separate kernel launches (element-wise addition for input linear combinations, cuBLAS GEMM for matrix multiplication, and element-wise addition for output accumulation), for a total of 21 launches per top-level call. At dimension 4096, the breakdown shows that this separate-kernel FP32 baseline moves 1.83 GB of DRAM traffic, 3.83× that of cuBLAS, and reaches only 0.931× cuBLAS effective throughput. The fused BAG primitive eliminates the write-back and re-read of intermediate matrices $P = X + \delta Y$ and $Q = V + \varepsilon W$, yielding a 1.46× speedup over the naive implementation in the ablation.

Benefits of Adaptive Autotuning. Figure 7b compares adaptive BAG with a fixed-parameter baseline. The baseline uses the high-occupancy Huge configuration ($m_S = n_S = 128, k_S = 8, m_R = n_R = 8$) and a fixed recursion cutoff. The adaptive framework provides a 1.70× performance boost at dimension 4096. The gain has two sources. First, the adaptive cutoff avoids excessive recursion overhead: the fixed-cutoff baseline recurses to depth 4 at dimension 4096 and incurs 128 kernel launches, while BAG terminates at depth 2 with only 56 launches. Second, the performance-model-guided tile selection (Section 5.3) chooses the L2-Huge configuration ($m_S = n_S = 256, k_S = 32, m_R = n_R = 8$) at the top level. This keeps the working set below the A100's 40 MB L2 capacity and achieves a 91.3% L2 hit rate, compared to 67.4% for the fixed-parameter baseline.



(a) Fused primitives.



(b) Adaptive tuning.

Figure 7: Ablation study results quantifying the performance impact of fused primitives and adaptive autotuning.

7 Related Work

Fast matrix multiplication has evolved from Strassen's $O(n^{2.807})$ algorithm [24] to variants with lower addition counts and improved practical constants. On GPUs, Huang et al. [10, 11] report a Strassen-Winograd break-even point of 1536 on V100 by fusing linear combinations with GEMM kernels, but their method still uses 15 additions per level and relies on fixed recursion cutoffs. ABMM [13, 14, 21] reduces this count to the optimal 12 additions while retaining Strassen-like stability, yet prior reported implementations focus on CPUs. BAG fills this gap by mapping ABMM to GPUs with fused primitives, memory-efficient scheduling, and adaptive autotuning for FP32 GEMM.

8 Conclusions and Future Work

BAG makes ABMM practical for FP32 GEMM on GPUs by eliminating unnecessary workspace traffic, fusing basis transformations into GEMM primitives, and selecting recursion and blocking parameters with performance models. On A100 for all baselines, BAG reaches a 1280 break-even point against cuBLAS for square matrices and achieves a 1.24× speedup at dimension 4096, indicating that ABMM's reduced addition count can translate into practical GPU gains when paired with architecture-aware fusion and tuning. Future work will extend BAG to mixed-precision Tensor Core paths and distributed-memory settings.

Acknowledgments

The authors would like to thank all anonymous reviewers for their valuable comments and helpful suggestions.

This work is supported by the following funding: National Science Foundation of China (92270206, 62372435, T2125013), and the Innovation Funding of ICT, CAS. Part of the training is performed on the robotic AI-Scientist platform of Chinese Academy of Sciences.

References

[1] Josh Alman, Ran Duan, Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. 2025. More Asymmetry Yields Faster Matrix Multiplication. In *Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2025, New Orleans, LA, USA, January 12-15, 2025*. SIAM, 2005–2039. doi:10.1137/1.9781611978322.63

[2] Josh Alman and Hantao Yu. 2025. Improving the Leading Constant of Matrix Multiplication. In *Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2025, New Orleans, LA, USA, January 12-15, 2025*. SIAM, 1933–1971. doi:10.1137/1.9781611978322.61

[3] Andrew Anderson, Aravind Vasudevan, Cormac Keane, and David Gregg. 2017. Low-memory gemm-based convolution algorithms for deep neural networks. arXiv:1709.03395 https://arxiv.org/abs/1709.03395

[4] Austin R. Benson and Grey Ballard. 2015. A Framework for Practical Parallel Fast Matrix Multiplication. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 42–53. doi:10.1145/2688500.2688513

[5] Brice Boyer, Jean-Guillaume Dumas, Clément Pernet, and Wei Zhou. 2009. Memory efficient scheduling of Strassen-Winograd's matrix multiplication algorithm. In *Proceedings of the 2009 international symposium on Symbolic and algebraic computation*. 55–62.

[6] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems* 35 (2022), 16344–16359.

[7] Craig C Douglas, Michael Heroux, Gordon Slishman, and Roger M Smith. 1994. GEMMW: A portable level 3 BLAS Winograd variant of Strassen's matrix-matrix multiply algorithm. *J. Comput. Phys.* 110, 1 (1994), 1–10.

[8] Ran Duan, Hongxun Wu, and Renfei Zhou. 2023. Faster Matrix Multiplication via Asymmetric Hashing. In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023, Santa Cruz, CA, USA, November 6-9, 2023*. IEEE, 2129–2138. doi:10.1109/FOCS57990.2023.00130

[9] Jianyu Huang, Leslie Rice, Devin A. Matthews, and Robert A. van de Geijn. 2017. Generating Families of Practical Fast Matrix Multiplication Algorithms. In *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*. IEEE Computer Society, 656–667. doi:10.1109/IPDPS.2017.56

[10] Jianyu Huang, Tyler M. Smith, Greg M. Henry, and Robert A. van de Geijn. 2016. Strassen's algorithm reloaded. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*. IEEE Computer Society, 690–701. doi:10.1109/SC.2016.58

[11] Jianyu Huang, Chenhan D Yu, and Robert A van de Geijn. 2020. Strassen's algorithm reloaded on GPUs. *ACM Transactions on Mathematical Software (TOMS)* 46, 1, Article 1 (2020), 22 pages. doi:10.1145/3372419

[12] S. Huss-Lederman, E.M. Jacobson, J.R. Johnson, A. Tsao, and T. Turnbull. 1996. Implementation of Strassen's Algorithm for Matrix Multiplication. In *Supercomputing '96:Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*. 32–32. doi:10.1109/SUPERC.1996.183534

[13] Elaye Karstadt and Oded Schwartz. 2017. Matrix Multiplication, a Little Faster. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2017, Washington DC, USA, July 24-26, 2017*. ACM, 101–110.

[14] Elaye Karstadt and Oded Schwartz. 2020. Matrix multiplication, a little faster. *Journal of the ACM (JACM)* 67, 1, Article 1 (2020), 31 pages. doi:10.1145/3364504

[15] Pai-Wei Lai, Humayun Arafat, Venmugil Elango, and P. Sadayappan. 2013. Accelerating Strassen-Winograd's matrix multiplication algorithm on GPUs. In *20th Annual International Conference on High Performance Computing*. 139–148. doi:10.1109/HiPC.2013.6799109

[16] Andrew Lavin and Scott Gray. 2016. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4013–4021. doi:10.1109/CVPR.2016.435

[17] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. 2023. Sequence parallelism: Long sequence training from system perspective. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2391–2404.

[18] Xiuhong Li, Yun Liang, Shengen Yan, Liancheng Jia, and Yinghan Li. 2019. A coordinated tiling and batching framework for efficient GEMM on GPUs. In *Proceedings of the 24th symposium on principles and practice of parallel programming*. 229–241. doi:10.1145/3293883.3295734

[19] Yuyao Niu, Zhengyang Lu, Haonan Ji, Shuhui Song, Zhou Jin, and Weifeng Liu. 2022. TileSpGEMM: A tiled algorithm for parallel sparse general matrix-matrix multiplication on GPUs. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 90–106. doi:10.1145/3503221.3508431

[20] David Parello, Olivier Temam, and Jean-Marie Verdun. 2002. On Increasing Architecture Awareness in Program Optimizations to Bridge the Gap between Peak and Sustained Processor Performance: Matrix-Multiply Revisited. In *SC'02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. IEEE, 1–11.

[21] Oded Schwartz, Sivan Toledo, Noa Vaknin, and Gal Wiernik. 2024. Alternative Basis Matrix Multiplication is Fast and Stable. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS 2024, San Francisco, CA, USA, May 27-31, 2024*. IEEE, 38–51. doi:10.1109/IPDPS57955.2024.00013

[22] Oded Schwartz and Noa Vaknin. 2023. Pebbling Game and Alternative Basis for High Performance Matrix Multiplication. *SIAM J. Sci. Comput.* 45, 6 (2023), C277–C303. doi:10.1137/22M1502719

[23] Oded Schwartz and Eyal Zwecher. 2025. Towards Faster Feasible Matrix Multiplication by Trilinear Aggregation. arXiv:2508.01748 doi:10.48550/ARXIV.2508.01748

[24] Volker Strassen. 1969. Gaussian elimination is not optimal. *Numerische mathematik* 13, 4 (1969), 354–356.

[25] Guangming Tan, Linchuan Li, Sean Triechle, Everett Phillips, Yungang Bao, and Ninghui Sun. 2011. Fast implementation of DGEMM on Fermi GPU. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. Article 35, 11 pages. doi:10.1145/2063384.2063431

[26] M. Thottethodi, S. Chatterjee, and A.R. Lebeck. 1998. Tuning Strassen's Matrix Multiplication for Memory Efficiency. In *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. 36–36. doi:10.1109/SC.1998.10045

[27] Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. 2024. New Bounds for Matrix Multiplication: from Alpha to Omega. In *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024, Alexandria, VA, USA, January 7-10, 2024*. SIAM, 3792–3835. doi:10.1137/1.9781611977912.134

[28] Aravind Vasudevan, Andrew Anderson, and David Gregg. 2017. Parallel multi channel convolution using general matrix multiplication. In *2017 IEEE 28th international conference on application-specific systems, architectures and processors (ASAP)*. IEEE, 19–24.

[29] 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 30*. Curran Associates, Inc., 5998–6008. https://proceedings.neurips.cc/paper_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[30] Sinong Wang, Belinda Z Li, Madian Khabsa, Han Fang, and Hao Ma. 2020. Linformer: Self-attention with linear complexity. arXiv:2006.04768 https://arxiv.org/abs/2006.04768

[31] Shmuel Winograd. 1971. On Multiplication of 2×2 Matrices. *Linear Algebra Appl.* 4, 4 (1971), 381–388. doi:10.1016/0024-3795(71)90009-7

[32] Du Wu, Jintao Meng, Wenxi Zhu, Minwen Deng, Xiao Wang, Tao Luo, Mohamed Wahib, and Yanjie Wei. 2024. autoGEMM: Pushing the Limits of Irregular Matrix Multiplication on Arm Architectures. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15. doi:10.1109/SC41406.2024.00027

[33] 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 *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 31–43. doi:10.1145/3018743.3018755