

Dense GEMM on GPU: Naive, Tiled, and cuBLAS Implementations

Keda Liu
COMP 468/568 – Spring 2026
January 31, 2026

1 Introduction

In this experiment, we implement dense General Matrix Multiplication (GEMM) on an NVIDIA GPU using CUDA. We compare three approaches: (1) a naive CUDA kernel where each thread computes one output element, (2) a tiled CUDA kernel using shared memory (block size 32×32), and (3) the cuBLAS `sgemm` routine as a reference baseline. Performance is reported in terms of execution time and throughput (GFLOP/s), using the standard GEMM operation count $2MNK$.

2 Implementation

2.1 Host-side orchestration (`main.cu`)

The host code parses runtime parameters (M , N , K , implementation type, and verification flag), allocates device memory, copies input matrices to the GPU, and launches the selected GEMM kernel. CUDA events are used to measure kernel execution time. After execution, results are copied back to the host. If verification is enabled, cuBLAS is used to compute a reference result and the maximum absolute error is reported.

2.2 Naive CUDA kernel

The naive kernel assigns one CUDA thread to compute one element of the output matrix C . Each thread computes a full dot product between a row of matrix A and a column of matrix B :

$$C_{ij} = \sum_{t=0}^{K-1} A_{it} B_{tj}.$$

This implementation is straightforward but performs many redundant global memory accesses, leading to limited performance.

2.3 Tiled CUDA kernel with shared memory

The tiled kernel improves performance by loading sub-tiles of matrices A and B into shared memory. Threads within a block cooperatively load data and reuse it to compute partial dot products. This reduces global memory traffic and increases arithmetic intensity. Boundary checks ensure correctness for arbitrary matrix sizes.

```

==== ENV ====
bb6u15g1
Sat Jan 31 20:02:02 2026

```

NVIDIA-SMI 535.216.03			Driver Version: 535.216.03			CUDA Version: 12.2		
GPU	Name	Persistence-M	Bus-Id	Disp.A	Volatile	Uncorr.	ECC	
Fan	Temp	Pwr:Usage/Cap		Memory-Usage	GPU-Util	Compute M.	MIG M.	
0	NVIDIA L40S	On	00000000:C2:00.0	Off	0%	Default	N/A	
N/A	31C	32W / 350W	0MiB / 46068MiB					

Processes:							
GPU	GI	CI	PID	Type	Process name	GPU Memory	Usage
	ID	ID					
No running processes found							

Figure 1: GPU environment on the Rice University cluster, including driver/CUDA version and NVIDIA L40S device information (`nvidia-smi`).

3 Experimental Setup

Experiments were conducted on the Rice University GPU cluster using an NVIDIA L40S GPU. The program was compiled using `nvcc` via the provided Makefile and executed on a GPU-enabled node. Square matrix sizes of 256, 512, and 2048 were evaluated. Performance is reported as execution time (milliseconds) and throughput:

$$GFLOP/s = \frac{2MNK}{(timeinms) \times 10^6}.$$

```

nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2025 NVIDIA Corporation
Built on Tue_May_27_02:21:03_PDT_2025
Cuda compilation tools, release 12.9, V12.9.86
Build cuda_12.9.r12.9/compiler.36037853_0
===== CORRECTNESS =====
MaxAbsError=7.629395e-05
Impl=naive M=256 N=256 K=256 Time(ms)=0.08 GFLOP/s=420.10
MaxAbsError=6.866455e-05
Impl=tiled M=256 N=256 K=256 Time(ms)=0.08 GFLOP/s=442.81
===== PERF 512 =====
Impl=naive M=512 N=512 K=512 Time(ms)=0.12 GFLOP/s=2263.52
Impl=tiled M=512 N=512 K=512 Time(ms)=0.11 GFLOP/s=2496.61
Impl=cublas M=512 N=512 K=512 Time(ms)=28.20 GFLOP/s=9.52
===== PERF 2048 =====
Impl=naive M=2048 N=2048 K=2048 Time(ms)=3.12 GFLOP/s=5515.19
Impl=tiled M=2048 N=2048 K=2048 Time(ms)=2.50 GFLOP/s=6878.73
Impl=cublas M=2048 N=2048 K=2048 Time(ms)=27.14 GFLOP/s=632.91
~

```

Figure 2: Program output showing numerical correctness (MaxAbsError) and performance (time and GFLOP/s) for naive, tiled, and cuBLAS implementations across multiple matrix sizes.

Table 1: Performance summary of GEMM implementations (from Figure 2).

M	N	K	Implementation	Time (ms)	GFLOP/s
256	256	256	naive	0.08	420.10
256	256	256	tiled	0.08	442.81
512	512	512	naive	0.12	2263.52
512	512	512	tiled	0.11	2496.61
512	512	512	cusblas	28.20	9.52
2048	2048	2048	naive	3.12	5515.19
2048	2048	2048	tiled	2.50	6878.73
2048	2048	2048	cusblas	27.14	632.91

4 Results

All custom CUDA implementations produced results numerically consistent with the cuBLAS reference. The maximum absolute error across all experiments was on the order of 10^{-4} , which is acceptable for single-precision floating-point arithmetic (Figure 2).

Table 1 summarizes the measured performance. The tiled kernel consistently outperforms the naive implementation, with the performance gap increasing for larger matrices.

Table 2: Effect of tile size on the tiled GEMM kernel for $M = N = K = 2048$.

Tile Size	Time (ms)	GFLOP/s	MaxAbsError
16	2.53	6792.40	1.59×10^{-3}
32	2.59	6631.23	1.71×10^{-3}

5 Observations and Analysis

The tiled implementation achieves higher performance than the naive kernel due to improved memory locality and data reuse enabled by shared memory. As matrix size increases, performance improves because larger workloads better utilize GPU resources and amortize kernel launch overheads.

In this experiment, the cuBLAS implementation appears slower than the custom kernels. A likely explanation is that one-time overheads (e.g., handle creation and library setup) dominate when only a single timed call is used. A fairer comparison would include warm-up iterations and multiple repeats, timing only the steady-state GEMM calls.

5.1 Effect of Tile Size

We compared tile sizes 16 and 32 for the shared-memory tiled kernel. Both configurations produce numerically correct results with maximum absolute error on the order of 10^{-3} . For $M = N = K = 2048$, tile size 16 achieves 6792.40 GFLOP/s while tile size 32 achieves 6631.23 GFLOP/s, an improvement of approximately 2.4% for tile size 16.

A tile size of 64 is not directly applicable to the current kernel mapping because it would require a 64×64 thread block (4096 threads), exceeding CUDA’s limit of 1024 threads per block. Supporting a 64×64 output tile would require a different mapping (e.g., register blocking), so we focus on 16 vs 32 under the same kernel structure for a fair comparison.

6 Conclusion

This experiment demonstrates the effectiveness of shared memory tiling for accelerating dense matrix multiplication on GPUs. The tiled kernel significantly outperforms the naive implementation while maintaining numerical correctness. The results highlight the importance of memory hierarchy awareness and careful benchmarking when evaluating GPU performance.

Reproducibility

Build and run examples:

```
make
./bin/dgemm --m 2048 --n 2048 --k 2048 --impl tiled
./bin/dgemm --m 2048 --n 2048 --k 2048 --impl cublas
./bin/dgemm --m 2048 --n 2048 --k 2048 --impl tiled --tile-size 16
./bin/dgemm --m 2048 --n 2048 --k 2048 --impl tiled --tile-size 32
```