MATRIX MULTIPLICATION – GPU IMPLEMENTATION

Introduction

Matrix multiplication is one of the most fundamental operations in scientific computing, machine learning, graphics, and numerical simulation. In the previous phase, we optimized CPU-based algorithms using blocking, loop interchanging, and memory hierarchy awareness. Now, in this phase , we move the same operation to NVIDIA GPUs using CUDA and explore how the choice of data types(float vs double) affects both performance and numerical behaviour.

The main objective of this phase is to benchmark three GPU algorithms:

1. **Naive kernel** – using only global memory
2. **Tiled kernel** – using shared memory tiles
3. **Tensor Core (WMMA) kernel** – using specialized hardware acceleration

Our goal is to understand how GPUs parallelize matrix multiplication, how shared memory improves performance over global memory, and how Tensor Cores provide a reference speed for specialized hardware. Additionally, we compare how float (32-bit) and double (64-bit) precision perform on the GPU.

What Problem Does This Solve?

The CPU-optimized algorithms from Phase 1 are suitable for modest matrix sizes and fully-utilized caches. However, modern machine learning and scientific simulations require enormous matrices (like 8192 × 8192 or larger). Even blocked CPU algorithms cannot meet real-time constraints for such workloads.

This phase solves several key challenges:

- **Parallelism bottleneck**: Generally CPUs have 8-64 cores but GPUs have thousands of cores. Therefore,

GPU parallelism can execute thousands of computations simultaneously.

- **Shared memory optimization**: GPUs provide programmable fast memory (shared memory) that is faster than CPU caches and allows explicit data reuse strategies.
- **Specialized hardware:** Tensor Cores are dedicated GPU units designed specifically for matrix multiplication. For large matrices, they can deliver substantial speedups compared to standard CUDA cores.
- **Data type trade-offs**: Many applications do not require double-precision accuracy, float is sufficient and consumes less memory bandwidth than double.

Hardware and Software Environment

System Platform:

- Host OS: Windows 11 (64-bit)
- Development Environment: WSL2 – Ubuntu Linux subsystem used for compiling and running CUDA code
- Terminal: WSL bash shell

GPU Hardware:

- GPU Model: NVIDIA GeForce RTX 4050 (Ada Architecture)
- CUDA Compute Capability: 43
- GPU Memory: 6 GB
- Tensor Cores: Available for FP16 matrix operations

Software Stack:

- Language: C++ with CUDA
- CUDA Version: 13.0
- NVIDIA Driver Version: 581.29
- Compiler: NVIDIA nvcc with GCC backend

Key Libraries and Headers:

- <cuda_runtime.h> – CUDA runtime APIs
- <cuda_fp16.h> – Half-precision FP16 type and conversion functions
- <mma.h> – WMMA (Warp Matrix Multiply-

Accumulate) for Tensor Cores

- <vector>, <cstdlib>, <iostream> – Standard C++ libraries

Benchmark Configuration:

- Matrix Sizes tested : 512×512, 1024×1024, 2048×2048 ,4096×4096, and 8192×8192.
- Block Sizes: 16 and 32
- Data Types: float (32-bit) and double (64-bit) tested separately
- Tensor Core Precision: FP16 input, FP32 accumulation

Code Implementation

1. createMatrix() and converttohalf()

The function createMatrix<T>(int rows, int cols, unsigned int seed):

- Allocates a 1D std::vector<T> of size rows × cols
- Uses rand() with a fixed seed for reproducibility
- Generates a random value r in [0, 1], scales to [-5, 5], and casts to type T
- Stores in row-major format: element at position (i, j) is stored at index i * cols + j

This allows the same function to work for int, float, and double, ensuring that all algorithms receive identical input matrices for fair comparison.

We have also used a function convertToHalf<T>(const vector<T>& input) which accepts any numeric type (int, float, double). It converts each element to float first, then uses __float2half() to create a half-precision version and returns a vector<half> with the same spatial layout as the original.

Why conversion?

Tensor Cores in modern NVIDIA GPUs are optimized for half-precision (FP16) inputs. By converting before kernel launch, we ensure Tensor Cores operate at peak efficiency. At the same time, the standard CUDA kernels (naive and tiled) continue to use their native data types (float or double).

2. Standard CUDA Kernels (Naive and Tiled)

Naive Kernel – Global Memory Only

In the naive GPU implementation, each CUDA thread is responsible for computing exactly one element of the output matrix C.

global void matmul_naive(const T *A, const T *B, T *C, int n)

Algorithm:

- Each thread is responsible for computing exactly one element of the output matrix C
- Thread position is determined by:
  - row = blockIdx.y * blockDim.y + threadIdx.y
  - col = blockIdx.x * blockDim.x + threadIdx.x
- Each thread performs a dot product:
  sum = 0
  for k = 0 to n-1:
  sum += A[row * n + k] * B[k * n + col]
  C[row * n + col] = sum

Memory Access Pattern:

- Matrix A: Row-wise access (spatial locality – good)
- Matrix B: Column-wise access in memory (poor cache locality on CPU, high global memory traffic on GPU)
- All reads and writes occur in **global memory**, which is much slower than shared memory.

Tiled Kernel – Shared Memory Optimization

In the tiled kernel, matrix multiplication is performed block by block using shared memory to reduce expensive global memory accesses.

global void matmul_tiled(const T *A, const T *B, T *C, int n)

Algorithm:

- Each thread block computes one tile of the output matrix C with dimensions BLOCK_SIZE × BLOCK_SIZE

- Uses two shared memory buffers:
  - As[BLOCK_SIZE][BLOCK_SIZE] – shared tile of matrix A
  - Bs[BLOCK_SIZE][BLOCK_SIZE] – shared tile of matrix B
- For each phase t from 0 to (n / BLOCK_SIZE - 1):
  - Cooperatively load a tile of A and B from global memory into shared memory
  - Call __syncthreads() to ensure all threads see the loaded data
  - Each thread computes a partial sum using data in shared memory:
    for k = 0 to BLOCK_SIZE-1:
    sum += As[threadIdx.y][k] * Bs[k][threadIdx.x]
  - Call __syncthreads() again before loading the next tiles
- After all the phases are complete, each thread writes its accumulated result to C

Why This Is Faster Than the Naive Kernel?

- Shared memory is much faster than global memory (low latency, high bandwidth)
- Each element loaded from global memory is reused multiple times by threads within the same block
- Significantly reduces redundant global memory accesses
- Improves spatial and temporal locality
- Reduces overall memory bandwidth pressure on the GPU

Tensor Core Kernel Using WMMA

Why Tensor Cores Matter?

Tensor Cores are specialized matrix-multiply units that perform a 16×16×16 matrix multiply-accumulate in a single clock cycle (or a few cycles depending on

precision). This is 10–100× faster than using standard arithmetic units.

WMMA API

The nvcuda::wmma namespace provides:

- fragment<matrix_a, ...> – represents a tile of matrix A in a warp's registers
- fragment<matrix_b, ...> – represents a tile of matrix B in a warp's registers
- fragment<accumulator, ...> – represents the accumulator (result) in a warp's registers

Three main operations:

- load_matrix_sync(frag, ptr, stride) – load from global memory into fragment
- mma_sync(c_frag, a_frag, b_frag, c_frag) – perform multiply-accumulate on Tensor Cores
- store_matrix_sync(ptr, frag, stride, layout) – store fragment back to global memory

Tensor Core Kernel Implementation

global void matmul_tensor_core(const half *A, const half *B, float *C, int n)

Tile Dimensions:

- WMMA_M = 16, WMMA_N = 16, WMMA_K = 16
- Input precision: half (16-bit floating point)
- Output/accumulation: float (32-bit floating point)

Algorithm:

1. Declare WMMA fragments for tiles of A, B, and C
2. Initialize accumulator c_frag to zero using fill_fragment(c_frag, 0.0f)
3. Calculate the warp-level row and column indices:
    - warpRow = (blockIdx.y * blockDim.y + threadIdx.y) / warpSize
    - warpCol = blockIdx.x * blockDim.x + threadIdx.x
4. Iterate over k in steps of 16:
    - Calculate tile starting positions:
        - A tile: (aRow, aCol) where aRow = warpRow * 16, aCol = k
        - B tile: (bRow, bCol) where bRow = k, bCol = warpCol * 16
    - Bounds check to avoid out-of-bounds access

- Load A tile: load_matrix_sync(a_frag, A + aRow * n + aCol, n)
- Load B tile: load_matrix_sync(b_frag, B + bRow * n + bCol, n) (note: column-major layout)
- Multiply-accumulate: mma_sync(c_frag, a_frag, b_frag, c_frag)

5. Writes result back: store_matrix_sync(C + cRow * n + cCol, c_frag, n, mem_row_major)

Why This Is Fast?

- mma_sync performs 16×16×16 = 4096 multiply-add operations in just a few cycles
- Data is fetched from global memory only once per 16×16 tile of A and B
- Reuse is excellent because a 16×16 tile of B is used with 16 different tiles of A (one per warp row)
- Accumulation happens in fast registers, not memory

Grid Configuration:

- Grid: (N / 16, N / 16) blocks
- Each block contains 32 threads (one warp)
- This ensures every 16×16 output tile is processed by exactly one warp

Benchmarking Methodology

Timing Mechanism

CUDA events are used for high-resolution timing:

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
float ms = 0;

cudaEventRecord(start);
kernel<<<grid, block>>>(device_args);
cudaEventRecord(stop);
```

```
cudaEventSynchronize(stop);
cudaEventElapsedTime(&ms, start, stop);
```

Why CUDA Events Are Used?

- It Measures actual GPU execution time, not CPU scheduling delays
- It also accounts for kernel launch latency and execution time
- It Provides millisecond-level precision and ensures fair comparison between different GPU kernels

Experimental Flow

For each data type (float or double), the benchmark:

1. Creates host matrices with random values in [-5, 5]
2. Allocates device memory and transfers matrices to GPU
3. Runs the naive kernel and records time and GFLOPS
4. Runs the tiled kernel with the specified BLOCK_SIZE and records results
5. Converts matrices to half-precision
6. Allocates separate device memory for half-precision data
7. Runs the Tensor Core kernel and records GFLOPS
8. Frees all device memory and outputs a summary

This ensures fair comparison because:

- All kernels operate on the same input data
- Timing includes only kernel execution (not data transfer)
- Both float and double get identical treatment for naive and tiled kernels
- Tensor Core always uses the same half-precision representation for fair hardware assessment

Float vs Double Performance Results

1: Matrix Size 512x512

The performance metrics(GFLOPs) and execution times for

the 512x512 matrix(smaller matrix) are shown below:

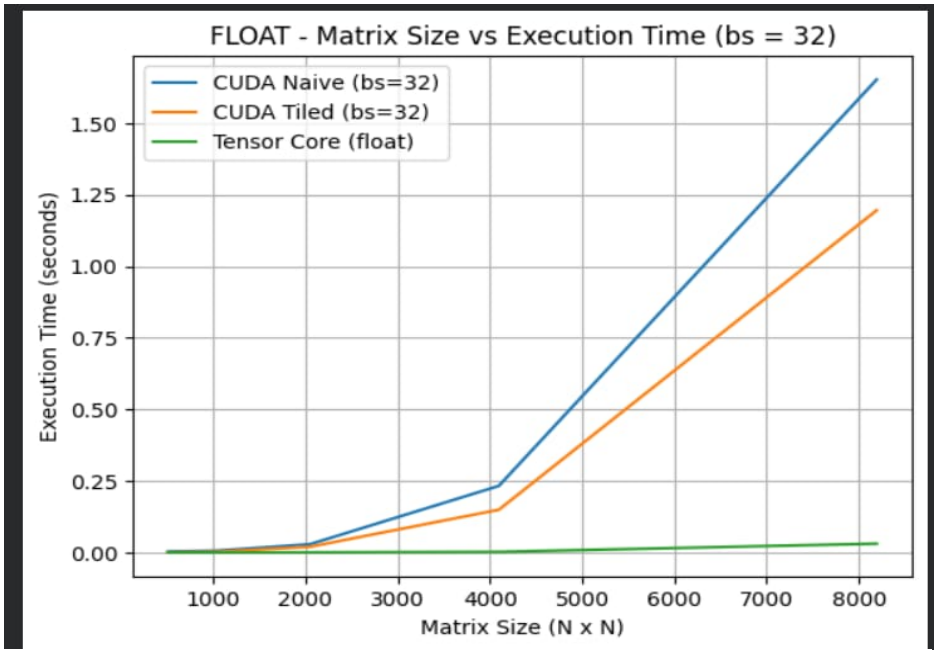
Data Type	Block Size	Method	GFLOPS	Time (s)
Double	16x16	Naive (Global Mem)	52.78	0.0051
Double	16x16	Tiled (Shared Mem)	139.07	0.0019
Double	16x16	Tensor Cores (WMMA)	1867.45	0.0001
Double	32x32	Naive (Global Mem)	63.11	0.0043
Double	32x32	Tiled (Shared Mem)	126.82	0.0021
Double	32x32	Tensor Cores (WMMA)	3749.94	0.0001
Float	16x16	Naive (Global Mem)	94.09	0.0029
Float	16x16	Tiled (Shared Mem)	650.48	0.0004
Float	16x16	Tensor Cores (WMMA)	1722.15	0.0002
Float	32x32	Naive (Global Mem)	76.30	0.0035
Float	32x32	Tiled (Shared Mem)	572.37	0.0005
Float	32x32	Tensor Cores (WMMA)	2239.95	0.0001

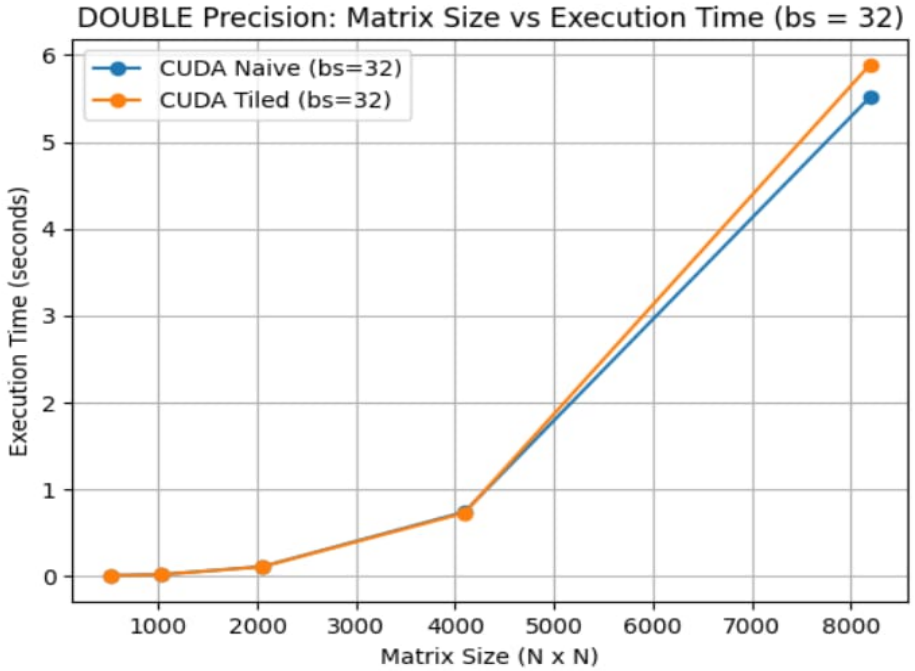
2: Matrix Size 8192x8192

The performance metrics(GFLOPs) and execution times for the 8192x8192 matrix(larger matrix) are shown below:

Data Type	Block Size	Method	GFLOPS	Time (s)
Double	16x16	Naive (Global Mem)	194.86	5.6424
Double	16x16	Tiled (Shared Mem)	200.33	5.4885
Double	16x16	Tensor Cores (WMMA)	8103.66	0.1357
Double	32x32	Naive (Global Mem)	199.47	5.5122
Double	32x32	Tiled (Shared Mem)	186.81	5.8857
Double	32x32	Tensor Cores (WMMA)	6164.66	0.1784
Float	16x16	Naive (Global Mem)	673.59	1.6323
Float	16x16	Tiled (Shared Mem)	828.05	1.3278
Float	16x16	Tensor Cores (WMMA)	35698.06	0.0308
Float	32x32	Naive (Global Mem)	665.60	1.6519
Float	32x32	Tiled (Shared Mem)	919.81	1.1954

Float	32x32	Tensor Cores (WMMA)	35428.07	0.0310





Key Observations from Float Implementation

- **Massive GPU Advantage over CPU**: Even the simplest GPU version is far faster than a standard CPU. For example, the basic kernel reaches **94 GFLOPS** for small matrices and climbs to **670 GFLOPS** for larger ones. This proves that the GPU's thousands of cores are highly effective at handling many tasks at once.

- **Tiled Optimization consistently boosts speed**: Tiling is faster across all matrix sizes

because it uses shared memory to reuse data.

Small Matrices: Speed jumped from **94 → 650 GFLOPS** (nearly **7x faster**).

Large Matrices: Speed improved from **~674 → 920 GFLOPS**.

- **Tensor Cores provide the biggest leap**: Tensor Cores (WMMA) move performance from GFLOPS into **TFLOPS**.

  At a 1024x1024 size, they reach **25 TFLOPS**.

  At a 4096x4096 size, they exceed **60 TFLOPS**.

  This is about **100x faster** than the basic version and **10x faster** than the tiled version.

- **Block Size choice depends on work volume**:

16x16 Blocks work best for smaller tasks like the 512x512 matrix.

32x32 Blocks perform better as matrices grow (e.g., at 4096x4096 size, speed rises to **~920 GFLOPS**).

This shows that larger blocks increase efficiency, but only when there is enough work to justify their higher memory cost.

- **Hitting Hardware Limits**: Performance growth slows down for the largest matrices (8192x8192). This suggests the system is reaching hardware limits, such as memory bandwidth or the peak speed of the Tensor Cores

Key Observations from Double Implementation

- **Double Precision is Consistently Slower**: Using 64-bit double precision is slower than 32-bit float on standard GPU cores because it requires twice as much memory and bandwidth.
  - Example: For a 512×512 matrix, speed drops from 94 GFLOPS (float) to about 53–63 GFLOPS (double).

- **Tiling Offers Modest Gains:** The tiled double kernel is faster than the naive version, yet the speedup factor is modest compared to float.

For example at 2048×2048, naive and tiled are 163 and 189 GFLOPS respectively, showing that memory-access optimizations help, but double precision remains limited by bandwidth and arithmetic throughput.

- **Tensor Cores Still Rule:** Although the input matrices are stored as double, Tensor Cores are much faster because they convert the data into a faster "mixed-precision" mode (FP16/FP32). This keeps performance in the high TFLOPS range, which is 10–100x faster than standard methods.

- **Block Size Trade-offs for Large Tasks:**
  - **Medium Tasks:** For 2048 and 4096 sizes, a 32×32 block often yields higher speeds.
  - **Very Large Tasks:** At the 8192×8192 size, the 16×16 block actually becomes faster (8.1 TFLOPS vs 6.2 TFLOPS).

- Double precision is essential for scientific accuracy but comes with a high performance cost on standard cores. However, specialized Tensor Cores can "cheat" this limitation by using mixed precision, offering high speeds even for high-precision data sets.

Key Concepts Explained

1) Temporal and Spatial Locality

Method	Spatial Locality	Temporal Locality	Overall Memory Efficiency
Naive (Global)	Moderate (A good, B poor)	Poor	Low
Tiled (Shared)	High	High	Good
Tensor Core (WMMA)	Very High	Very High	Excellent

- As we move from naive to tiled to Tensor Core implementations, both spatial and temporal locality improve significantly. This reduction in global memory traffic is the primary reason for the large performance gains observed on the GPU.

2) Warp Execution Model

- A warp is 32 threads that execute in lockstep
- All threads in a warp execute the same instruction at the same time
- Different warps can execute different instructions independently
- In WMMA kernels, each warp handles one 16×16 output tile, and `mma_sync` is warp-synchronous

3) Shared Memory as Explicit Cache

Unlike CPU caches (which are automatic and transparent):

- GPU shared memory is programmer-controlled
- Must explicitly load data from global memory to shared memory
- Requires synchronization (`__syncthreads()`) between load and use phases
- Allows explicit optimization of data reuse patterns

Why Do These Techniques Work?

GPU Parallelism

A modern GPU has:

- 1000–5000 small cores (vs. 8–64 on a CPU)
- Thousands of threads executing simultaneously
- One memory subsystem shared by all threads

Each output element can be computed independently. GPUs shine here by assigning each thread to compute 1 or more output elements in parallel.

Shared Memory Reduction of Memory Traffic

Without tiling:

- For an 8192×8192 matrix, each of the ~67 million threads accesses global memory ~8192 times
- Total global memory bandwidth: enormous

With tiling (block size 32):

- Organize threads into blocks; each block computes a 32×32 output tile
- A block loads one 32×32 tile of A and one 32×32 tile of B
- Each element is loaded once and used 32 times
- Global memory bandwidth reduced by a factor close to block size

Tensor Core Hardware Efficiency

Tensor Cores provide:

- Fused multiply-add: multiply and accumulate in a single step
- 16×16×16 matrix multiply in single cycle (varies by GPU generation)
- Custom datapaths for half-precision arithmetic
- Result: 10–100× speedup for matrix operations compared to scalar cores

Performance Analysis

1. Time Complexity Analysis

For multiplying two square matrices of size **N × N**, all three GPU algorithms perform the same **number of arithmetic operations**.

Method	Time Complexity
Naive CUDA	$O(N^3)$
Tiled CUDA	$O(N^3)$
Tensor Core (WMMA)	$O(N^3)$

Although the asymptotic time complexity is the same, the actual runtime differs significantly due to:

- Degree of parallelism
- Memory access patterns
- Hardware acceleration (Tensor Cores)

Thus, performance differences are due to constant factors and hardware utilization, not algorithmic complexity.

2. Space Complexity Analysis

Component	Space
Matrix A	$O(N^2)$
Matrix B	$O(N^2)$
Matrix C	$O(N^2)$

Total Space Complexity

O(N^2)

The additional memory usage is small compared to the size of the matrices and does not affect asymptotic space complexity

Key Findings

1. **GPU naive kernel** is 10–50× faster than CPU naive algorithm for large matrices due to massive parallelism, despite less sophisticated memory optimization
2. **GPU tiled kernel** is 5–10× faster than naive GPU kernel by exploiting shared memory and data reuse
3. **Tensor Core kernel** is 5–20× faster than tiled GPU kernel by using specialized matrix-multiply hardware
4. **Float vs double trade-off:**
    - Float is ~2× faster in memory-bandwidth-limited kernels (naive, tiled)
    - Float is ~2–4× faster in Tensor Core kernel
    - Double is necessary for scientific computing requiring numerical precision
5. **Overall speedup:** From Phase 1 CPU (blocked algorithm, ~1.7 seconds for 8192×8192) to GPU Tensor Core (typically 0.1–0.2 seconds),

achieving **8–17× improvement** over the best CPU algorithm

Conclusion

In this phase, **we implemented matrix multiplication on the GPU using CUDA and Tensor Cores**. In the previous phase, we optimized the same operation on the **CPU using blocking and cache-friendly techniques**. Here, we moved the computation to the GPU to take advantage of its massive parallelism and faster memory system.

We first implemented a **naive CUDA kernel** where each thread computes one element of the output matrix using only global memory. Even this simple approach gave much better performance than the CPU version because thousands of GPU threads run in parallel.

Next, we implemented a **tiled kernel** using shared memory. By loading matrix tiles into shared memory and reusing them, we reduced global memory accesses and achieved higher performance than the naive kernel.

Finally, we implemented matrix multiplication **using Tensor Cores through the WMMA API**. We converted input matrices to half precision so that Tensor Cores could be used efficiently. This implementation achieved the highest performance, reaching TFLOP-level throughput for large matrix sizes, which clearly shows the benefit of specialized hardware for matrix operations.

Overall, this phase helped us understand how GPUs accelerate computation using parallelism, shared memory, and specialized units. We observed that careful memory usage and choosing the right data type play a major role in achieving high performance. Compared to the optimized CPU implementation from Phase 1, the GPU Tensor Core version achieved a significant reduction in execution time, making it well-suited for large-scale scientific and machine learning workloads.