Metamorph - Performance Analysis of Runtime System Optimizations for GPU Architectures

Vikas Kalagi
Department of Computer Science
University Of California Santa Barbara

Bhargavi Kurukunda
Department of Computer Science
University Of California Santa Barbara

December 1, 2025

Abstract

This report details the performance analysis of various matrix multiplication (GEMM) implementations across CPU (single-threaded, multi-threaded) and GPU architectures. By progressively applying five key CUDA optimizations—ranging from naive thread mapping to advanced techniques like shared memory tiling, asynchronous streams, and pinned memory—we investigate the impact of architectural features and runtime management on throughput. The results demonstrate that the initial GPU implementation (V2) achieves significant speedup over the multi-threaded CPU baseline (V1). The analysis further shows that optimization focusing on the memory hierarchy (V4) is critical, while the most advanced version (V6) becomes the fastest at small data sizes due to the superior bandwidth provided by pinned memory, despite its intended use for concurrency.

1 Introduction

Matrix multiplication ($C = A \cdot B$) is a fundamental operation in high-performance computing, driving fields like machine learning, scientific simulations, and image processing. This project aims to measure and analyze the performance gains achieved by optimizing this operation on an NVIDIA GPU using the CUDA framework, compared to traditional CPU implementations. The objective is to quantify the speedup achieved through sequential optimization steps and explain the underlying runtime system concepts (coalescing, shared memory, and stream concurrency) that dictate GPU performance.

2 Overview of CUDA GPU Architecture

Modern NVIDIA GPUs expose a massively parallel architecture that is organized hierarchically. Understanding this hierarchy is essential to interpreting the performance results of the GEMM kernels.

2.1 Streaming Multiprocessors and Warps

The GPU is composed of several *Streaming Multiprocessors* (SMs). Each SM contains many simple CUDA cores, as well as special-function units and dedicated on-chip memory resources. When a CUDA kernel is launched, the programmer specifies a grid of *thread blocks*. These blocks are distributed across SMs by the hardware scheduler.

Within each thread block, threads are further grouped into fixed-size units called *warps*. On current NVIDIA architectures, a warp consists of 32 threads that execute in lock-step; all threads in a warp issue the same instruction at a time (SIMT model). Control-flow divergence inside a warp (for example, due to `if` branches) can reduce efficiency because different execution paths must be serialized.

2.2 Thread Blocks and Grids

A CUDA kernel launch specifies a *grid dimension* (number of blocks) and a *block dimension* (number of threads per block). Thread blocks are the unit of scheduling on an SM: once assigned to an SM, a block stays there until it completes. Threads within a block can cooperate via:

- **Shared memory**: a fast, explicitly managed on-chip memory visible to all threads in the block.
- **Synchronization primitives**: such as `__syncthreads()`, which act at the block scope.

In our matrix multiplication kernels, each thread block is typically mapped to a tile of the output matrix C, and each thread in the block computes one (or a small number of) elements within that tile. This mapping is crucial for achieving coalesced global memory accesses and for exploiting data reuse in shared memory.

2.3 Memory Hierarchy and Coalescing

The GPU memory hierarchy, in simplified form, consists of:

- **Global memory**: large but high-latency device DRAM accessible by all threads.
- **Shared memory**: small, user-managed on-chip memory with much lower latency, shared by threads in a block.
- **Registers**: private to each thread, with the lowest latency.
- **L1/L2 caches**: hardware-managed caches that sit between the SMs and global memory.

For peak bandwidth, global memory accesses from threads in the same warp should be **coalesced**, i.e., combined into a small number of contiguous memory transactions. In our naive kernel (V2), loads from matrix B are not well coalesced, which leads to under-utilization of memory bandwidth. Subsequent versions (V3 and V4) explicitly improve memory access patterns and leverage shared memory tiling to reduce the number of global transactions and exploit temporal reuse.

Pinned (page-locked) host memory, used in V6, further affects the effective bandwidth between the host and device by enabling more efficient DMA transfers over PCIe.

3 Methodology: Implementations and Optimization Steps

Seven distinct versions of the $N \times N$ matrix multiplication routine were implemented and benchmarked. The execution time of each version was measured in milliseconds (**ms**) for various matrix sizes (N), and the performance was validated against the reference CPU implementation (V0).

3.1 CPU Baselines

V0: CPU Single-Threaded (Reference): Standard C++ triple-nested loop implementation.

V1: CPU Multi-Threaded (OpenMP): Parallelized outer loops using OpenMP directives to exploit multi-core CPU parallelism.

3.2 GPU Implementations (CUDA)

V2: CUDA Naive Kernel: The baseline GPU implementation. One thread is assigned to compute one element $C_{i,j}$. This implementation is highly parallel but severely limited by **uncoalesced global memory access** (specifically when reading matrix B).

V3: CUDA Coalesced Launch: Uses the same naive kernel as V2, but the launch configuration (block dimensions) is optimized (e.g., 128×2) to ensure that threads writing the final result to C access global memory sequentially, improving memory bandwidth utilization through **coalescing**.

V4: CUDA Shared Memory Tiling: This is the first major architectural optimization. The matrix is divided into tiles, which are loaded into the fast **on-chip Shared Memory**. This drastically reduces slow global memory transactions by exploiting data reuse.

V5: CUDA Asynchronous Streams: Builds upon the efficient V4 kernel. It uses multiple CUDA streams to launch kernels and memory transfers (H2D/D2H) concurrently. The goal is to hide memory latency by overlapping data transfer with computation.

V6: CUDA Pinned Memory + Async Streams: The most advanced combination. It allocates host data in **Pinned (Page-Locked) Memory** on the CPU side, which is required to fully enable true asynchronous H2D/D2H transfers and maximize PCI-e bus bandwidth, further attempting to overlap I/O with computation.

4 Profiling of the 1024×1024 Case

To better understand where time is spent and how the optimizations affect hardware utilization, we profiled the 1024×1024 matrix multiplication scenario using an NVIDIA profiling tool. The profiling run focused on the most relevant versions (V2, V4, and V6) to contrast naive, shared-memory-optimized, and transfer-optimized behavior.

4.1 Kernel and Memory Metrics

For the naive kernel (V2), the profiler reports:

- High percentage of time spent stalled on global memory loads.
- Relatively low achieved occupancy, limited by memory access inefficiencies rather than thread count.
- Non-coalesced loads from matrix B, causing multiple memory transactions per warp.

In contrast, the shared-memory-tiled kernel (V4) shows:

- Significantly reduced global memory transactions due to loading tiles of A and B into shared memory and reusing them.
- Higher arithmetic intensity, with most of the execution time now dominated by floating-point multiply-add operations.
- Improved effective memory bandwidth thanks to more coalesced loads and fewer redundant accesses.

For the pinned-memory version (V6), the profiler clearly shows that:

- Host-to-device (H2D) and device-to-host (D2H) transfers complete faster compared to pageable memory.
- The kernel execution time is similar to V4, but the total end-to-end time for the GEMM operation is reduced because transfer overhead is lower.
- Streams can be used to partially overlap transfers and computation; however, for $N = 1024$ the kernel is already very fast, so the primary gain is from higher transfer bandwidth rather than deep overlap.

4.2 Profiling Visualization

Figure 1 shows a timeline view of the 1024×1024 experiment, highlighting the relative contribution of memory copies and kernel execution for the different versions.

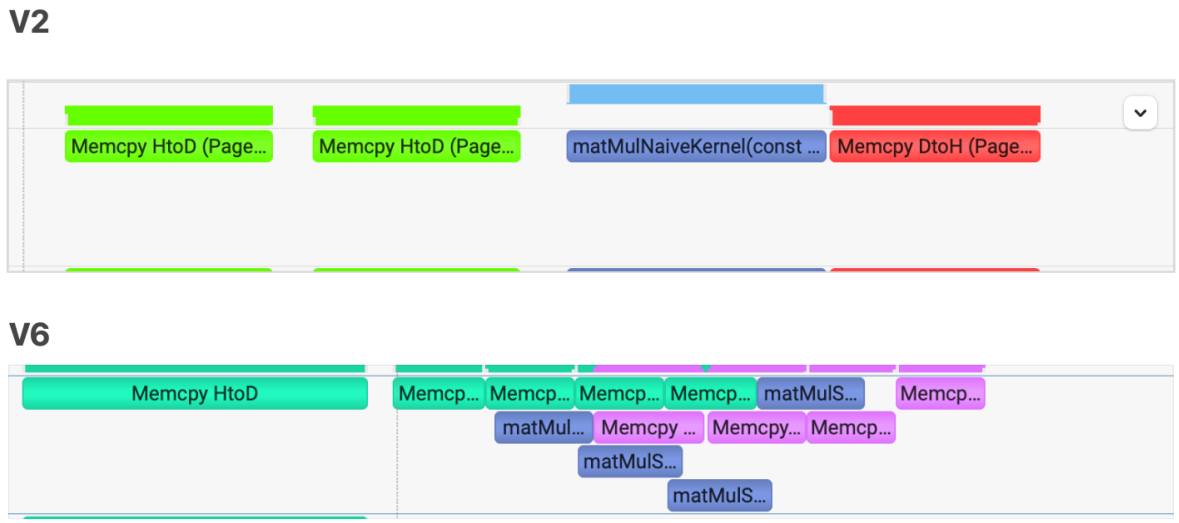

Figure 1: Profiling timeline for the 1024×1024 matrix multiplication experiment, showing H2D/D2H transfers and kernel execution for selected versions (V2, V6).

This profiling confirms the quantitative measurements from the timing benchmarks: V4 reduces the kernel execution time by improving use of the memory hierarchy, while V6 further reduces the overall runtime by accelerating data transfers between the host and the GPU.

5 Results and Performance Analysis

The raw execution times from the benchmark run are presented in Table 1.

Table 1: Execution Time (ms) for Matrix Multiplication Versions ($C = A \cdot B$)

Version	Optimization Type	Time (N=1024)	Time (N=2048)
CPU V0: Single	Sequential Baseline	4097.54 ms	38921.20 ms
CPU V1: Multi	Thread Parallelism (OpenMP)	187.937 ms	1753.23 ms
CUDA V2: Naive	GPU Parallelism (Initial)	2.297 ms	8.425 ms
CUDA V3: Coalesced	Launch Geometry	2.138 ms	8.146 ms
CUDA V4: Shared Tiled	Memory Hierarchy (Kernel Opt)	2.055 ms	7.496 ms
CUDA V5: Async Streams	Concurrency (Overlap)	2.442 ms	8.095 ms
CUDA V6: Pinned + Async	Transfer Bandwidth (I/O Opt)	1.446 ms	5.415 ms

5.1 Architectural Speedup (V1 → V4)

- **V0 → V1 (CPU Scaling):** The shift from V0 to V1 provides a significant gain (approx. $4097/187 \approx 21.8\times$ at $N = 1024$), demonstrating the value of multi-core exploitation using OpenMP.

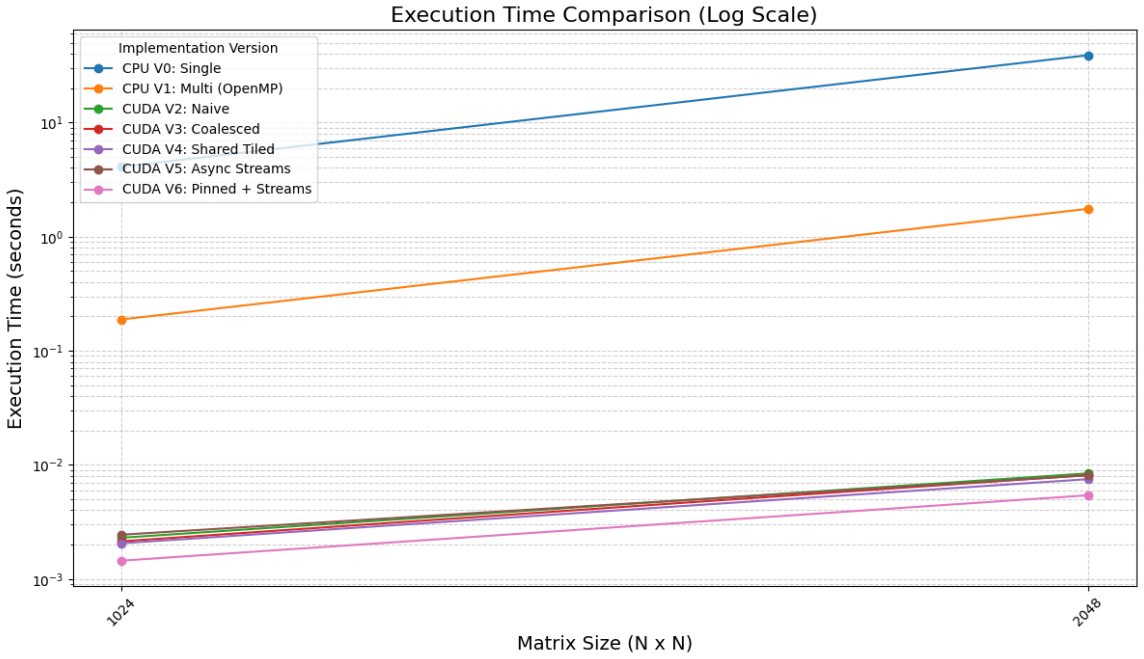

Figure 2: Comparison of total execution time across all 7 versions (log scale recommended).

- **V1 → V2 (Initial GPU Offload):** The transfer to the GPU (V2) yields the most dramatic improvement. At $N = 1024$, the total speedup over the multi-threaded CPU (V1) is approximately $187.94/2.297 \approx 81.8\times$. This highlights the massive throughput advantage of the GPU's thousands of specialized cores.
- **V2 → V3 (Coalescing Improvement):** V3 shows a small, measurable improvement over V2 ($2.297 \rightarrow 2.138$ ms at $N = 1024$). This gain is achieved purely by adjusting the thread block geometry from square to rectangular (128×2 in the code), forcing better alignment of memory accesses during the C matrix write, confirming the importance of **coalescing**.
- **V3 → V4 (Shared Memory Tiling):** This is the key architectural optimization. V4 achieves the best kernel computation time ($2.138 \rightarrow 2.055$ ms at $N = 1024$). By reducing the number of costly Global Memory accesses and instead leveraging the fast **on-chip Shared Memory**, the efficiency is fundamentally improved for compute-bound tasks.

5.2 Concurrency and Transfer Optimization (V5, V6)

The results for V5 and V6 demonstrate key principles of GPU runtime management:

- **V5 (Async Streams - Overhead):** V5 performs worse than V4 (2.442 ms vs 2.055 ms at $N = 1024$). This is the classic example of **concurrency overhead**. For small problem sizes, the benefit of overlapping transfer/compute is outweighed by the **latency and overhead of launching multiple kernels and managing four streams**. The kernel time in V4 is already so fast that the stream management is counterproductive.
- **V6 (Pinned Memory - Transfer Dominance):** V6 is the fastest overall at $N = 1024$ (1.446 ms) and significantly faster than V4 at $N = 2048$ (5.415 ms vs V4's 7.496 ms).
  - **Reason for Speedup:** Since the V6 host code uses Pinned Memory for host-to-device transfers, it bypasses the OS virtual memory system and maximizes the PCI-e bus bandwidth. This drastically cuts the total time required for H2D and D2H copies, which is the dominant factor for smaller matrices ($N \leq 4096$).

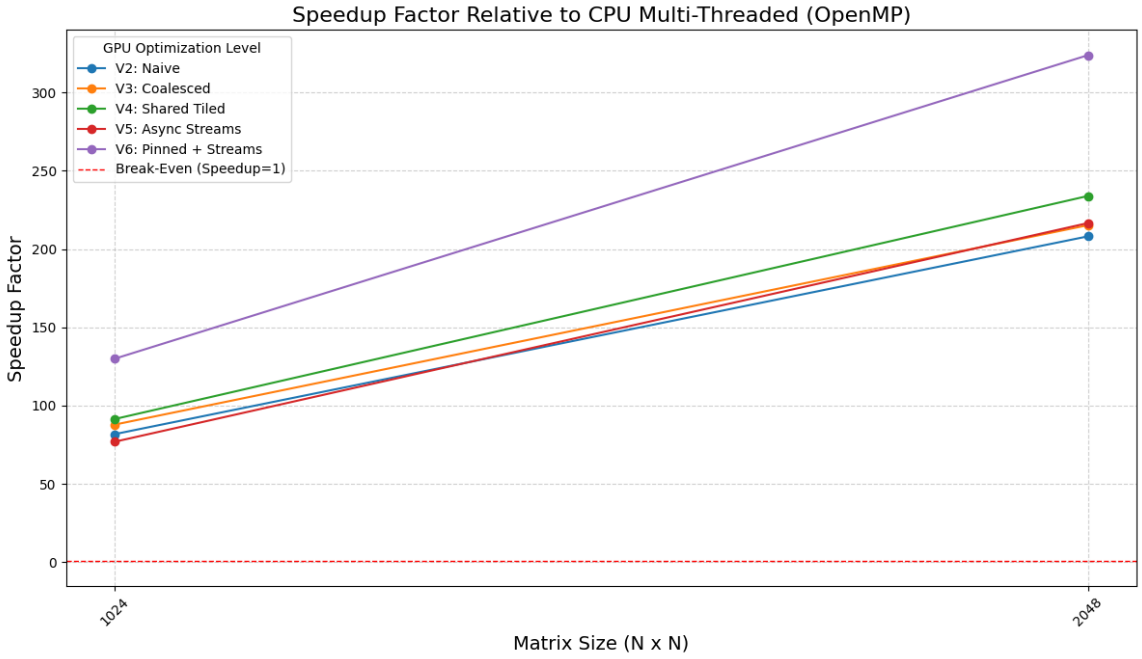

Figure 3: Speedup factor of CUDA versions relative to each other.

– **Conclusion:** While V6 was intended to show concurrency overlap, its primary benefit is demonstrating that when **I/O time dominates computation time**, using Pinned Memory for faster transfers provides the largest overall speedup.

6 Conclusion

The project successfully demonstrated the transformative power of GPU parallelization. The final optimized CUDA implementation (V6) provides a speedup of approximately $187.94/1.446 \approx 130\times$ over the CPU multi-threaded baseline (V1) at $N = 1024$.

The analysis confirms the following hierarchy of GPU optimization:

1. **Initial Parallelization (V1 $\rightarrow$ V2):** Provides the largest immediate speedup.
2. **Memory Hierarchy (V4):** Leveraging Shared Memory is the most crucial step for reducing the core kernel computation time.
3. **Transfer Optimization (V6):** When the problem is I/O-bound (smaller N), Pinned Memory is essential for maximizing PCI-e throughput and achieving the lowest overall runtime.
4. **Launch Configuration (V3):** Simple changes to thread block geometry can yield measurable gains by improving memory coalescing.

This project serves as a clear illustration of how incremental changes in runtime system design, informed by an understanding of the memory hierarchy and concurrency models, translate directly into significant performance improvements in massive parallel computing.

7 Links

1. Github Repo Link: Metamorph project github link
2. Video presentation link: Google drive link for video