Benchmarking Matrix Operations Optimizations

Afnan Alabdulwahab
Department of Data Science
University of Virginia
Charlottesville, USA
aa7dd@virginia.edu

Michael Jerge
Department of Computer Science
University of Virginia
Charlottesville, USA
mj6ux@virginia.edu

***Abstract*—This paper presents a comprehensive benchmarking study of matrix operation optimizations across NVIDIA GPU architectures, focusing on both transposition and multiplication. We evaluate performance impacts of memory access patterns, shared memory usage, and specialized hardware capabilities on RTX 2080 Ti and A100 GPUs. For matrix transposition, our vectorized implementation achieves up to 6× speedup over naive approaches, reaching 1800 GB/s on the A100 with large matrices, demonstrating the critical importance of coalesced memory access. Meanwhile, matrix multiplication benefits substantially from tensor core acceleration and library optimizations. Our experiments across multiple matrix sizes, ranging from small to large dimensions including non-square configurations, reveal that implementation efficiency varies significantly with matrix dimensions and hardware architecture. These findings provide practical insights for selecting optimal implementations based on specific workload characteristics, helping developers reduce computation time in AI and scientific computing applications where matrix operations dominate processing requirements.**

***Index Terms*—Matrix Multiplication, Matrix Transpose, GPU Optimization, CUDA Programming, Shared Memory, Tensor Cores, CUTLASS, CuBLAS, Performance Benchmarking, Memory Bandwidth Utilization**

I. Introduction

Matrix operations form the computational backbone of numerous applications in scientific computing, machine learning, computer graphics, and data analysis. These operations are particularly fundamental to AI workloads, where they can consume 70-90% of overall computation time. GPUs leverage their parallel architecture to accelerate these operations, but achieving optimal performance requires careful consideration of hardware characteristics, memory access patterns, and algorithmic design.

The optimization of matrix operations addresses several critical needs in modern computing. Efficient implementations directly reduce training time for machine learning models, translating to lower computing costs and enabling larger models to be trained within given resource constraints. As model sizes continue to grow, optimized matrix operations help maintain reasonable training and inference times through algorithmic improvements, hardware acceleration, precision reduction, and sparsity exploitation.

Profiling these implementations provides valuable insights by identifying bottlenecks, measuring GPU utilization, guiding hardware-specific tuning, and ensuring accuracy across different optimization techniques. This paper examines the performance characteristics of various matrix operation implementations, focusing specifically on matrix transposition and multiplication across different GPU architectures, offering insights into when and where specific implementations excel.

II. Related Work

Our work on matrix transpose optimization builds upon significant prior research in GPU computing optimization techniques.

Matrix transpose operations, due to their non-contiguous memory access patterns, have been extensively studied for performance optimization. Colfax Research provides a comprehensive tutorial on implementing efficient matrix transpose in CUTLASS, with particular focus on tiled approaches and memory access optimizations [1].

Custom implementations for matrix operations on GPUs have been explored by Mao, who details both multiplication optimizations [2] and transpose techniques using NVIDIA's CUDA Template Library (CuTe) [3]. These resources offer insights into performance characteristics of different optimization strategies across various matrix dimensions.

Our warp shuffle implementation leverages CUDA's register-to-register transfer capabilities as documented in NVIDIA's developer resources on warp-level primitives [4]. These primitives enable direct data transfer between threads within a warp without using shared memory as an intermediary.

For our template-based implementations, we relied on NVIDIA's CUDA Template Library (CuTe) documentation [5], which provides abstractions for tensor operations that simplify development while maintaining performance optimizations. The CuTe library offers layout-centric approaches to tensor operations, enabling efficient memory access patterns for operations like matrix transpose.

These resources collectively informed our implementation choices, from basic approaches to sophisticated library-based solutions, while guiding our methodology for performance analysis and comparison.

III. Background and Motivation

While matrix operations serve as fundamental building blocks across computational domains as outlined in the introduction, the specific challenges of optimizing these operations on GPUs require deeper examination. This section focuses on the technical barriers and optimization opportunities that motivated our research.

Transformer architectures, which power modern language models, rely heavily on multihead attention mechanisms that create computational bottlenecks. These mechanisms involve intricate sequences of matrix operations that must be optimized holistically rather than in isolation.

Matrix transposition on GPUs presents unique implementation challenges beyond those of general matrix operations. The fundamental mismatch between row-major storage format and column-major access pattern during transposition creates non-contiguous memory accesses that dramatically reduce cache efficiency and effective memory bandwidth. This mismatch is particularly problematic in GPU architectures, where memory coalescing is critical for performance.

For matrix multiplication, the computational intensity ($O(n^3)$ complexity) combined with data movement requirements creates a delicate balance between compute utilization and memory bandwidth. The optimal implementation strategy varies significantly across matrix dimensions and GPU architectures, making it impossible to identify a single approach that performs optimally across all scenarios.

The current optimization landscape offers several approaches with varying performance characteristics. NVIDIA's cuBLAS and CUTLASS libraries provide highly tuned implementations that work well for common use cases but may not fully exploit hardware capabilities across all matrix dimensions. Hardware features like Tensor Cores can dramatically accelerate compatible operations but require specific data formats and precision levels. Meanwhile, techniques such as memory coalescing, shared memory utilization, and vectorization offer significant performance improvements but introduce implementation complexity.

Our evaluation of these optimizations across multiple GPU architectures aims to bridge the gap between theoretical understanding and practical application, providing developers with actionable insights for selecting optimal approaches based on specific workload characteristics and hardware configurations.

IV. Experimental Setup

This section details our testing methodology, hardware configurations, and implementation details for both matrix transpose and matrix multiplication operations. All experiments were conducted on two different NVIDIA GPU architectures to provide insights into how performance characteristics vary across hardware generations.

A. Hardware Configuration

Our experiments were conducted on the following GPU architectures:

- NVIDIA GeForce RTX 2080 Ti (Turing architecture, 11GB GDDR6 memory)
- NVIDIA A100-PCIe-40GB (Ampere architecture, 40GB HBM2 memory)

We initially planned to include the NVIDIA H100 GPU in our testing but were unable to secure access during the project timeframe.

B. Matrix Transpose Setup

1) Matrix Dimensions: We tested transposition performance using the following matrix sizes:

- Small matrices: 32×32 elements (1 KB)
- Medium matrices: 1024×1024 elements (4 MB)
- Large matrices: 8192×8192 elements (256 MB)
- Non-square matrices: 1024×2048 elements (8 MB)

2) Implementations Benchmarked: We implemented and evaluated the following matrix transpose approaches, which will be explained in detail in the next section:

Custom Implementations:

- Naive
- Shared Memory
- Swizzled
- Vectorized
- WarpShuffle

Library-based Implementations:

- cuBLAS
- CuTe (Naive, Shared, Swizzled variants)

3) Performance Metrics and Verification: For each matrix transpose implementation, we measured kernel execution time in milliseconds and calculated memory throughput in GB/s using the formula $\frac{2 \times M \times N \times \text{sizeof(float)}}{time(s) \times 10^9}$, which accounts for both reading and writing operations. We also analyzed bandwidth utilization as a percentage of theoretical peak bandwidth for each GPU to understand implementation efficiency. To ensure correctness, all implementations were verified against a CPU reference implementation, using complete element-by-element verification for small and medium matrices, while large matrices (8192×8192) underwent statistical verification with more than 100 randomly sampled points to maintain testing efficiency without compromising confidence in the results.

C. Matrix Multiplication Setup

1) Matrix Dimensions: We tested multiplication performance using the following matrix sizes:

- Small matrices: 32×32×32 elements (multiplication of 32×32 by 32×32)
- Medium matrices: 1024×1024×1024 elements (4 MB per input matrix)
- Large matrices: 8192×8192×8192 elements (256 MB per input matrix)
- Non-square matrices: 1024×2048×1024 elements (multiplication of 1024×2048 by 2048×1024)

2) Implementations Benchmarked: We implemented and evaluated the following matrix multiplication approaches, which will be explained in detail in the next section:

Custom Implementations:

- Naive
- Shared Memory

- Tensor Cores

Library-based Implementations:

- cuBLAS
- CUTLASS
- cuSPARSE (for sparse matrix multiplication)

3) Performance Metrics and Verification: For each matrix multiplication implementation, we measured kernel execution time in milliseconds and calculated computational throughput in GFLOPs using the formula $\frac{2\times M\times N\times K}{time(s)\times 10^9}$, where M, N, and K represent the dimensions of the matrices being multiplied. We also analyzed computational efficiency as a percentage of theoretical peak performance for each GPU architecture. For sparse implementations, we evaluated performance across different matrix densities to understand the impact of sparsity patterns.

To ensure correctness, all implementations were verified against a CPU reference implementation using different tolerance levels: $\varepsilon = 10^{-2}$ for standard implementations and $\varepsilon = 10^{-1}$ for Tensor Core implementations to account for mixed-precision arithmetic. Similar to our transpose verification approach, we employed complete element-by-element verification for small and medium matrices, while large matrices underwent statistical verification with random sampling to balance testing efficiency and confidence in the results.

V. Implementation

This section details our implementation approach for matrix operations, including the specific optimization techniques employed and their theoretical advantages. All implementations were developed using CUDA C++ and compiled with NVIDIA's nvcc compiler using the appropriate architecture-specific flags for optimal code generation. Due to space constraints, we present only pseudocode representations of key algorithms; complete implementation details, full source code, and additional benchmarking utilities are available in our public repository [8].

A. Matrix Transpose Implementations

Matrix transposition is fundamentally a memory-bound operation that requires reshuffling data from row-major to column-major order. The key challenge is that this access pattern inherently causes non-contiguous memory access, which can significantly impact performance due to poor cache utilization and memory coalescing issues. We implemented seven distinct approaches to address these challenges:

1) Naive Implementation: Our baseline implementation performs a direct element-by-element copy from input to output. Each thread is responsible for transposing a single element by reading from global memory at position (row, col) and writing to position (col, row). While simple to implement, this approach suffers from uncoalesced memory access patterns when writing to the output matrix, as threads within a warp write to non-contiguous memory locations. This causes serialized memory transactions and significantly reduces effective memory bandwidth.

Algorithm 1 Naive Matrix Transpose

```
Input: Matrix A of size M×N, Output matrix B of size N×M
1: for each thread (i,j) in parallel do do
2:     row = blockIdx.y * blockDim.y + threadIdx.y
3:     col = blockIdx.x * blockDim.x + threadIdx.x
4:     if row $<$ M and col $<$ N then then
5:         B[col][row] = A[row][col]
6:     end if
7: end for
```

2) Shared Memory Implementation: To address the global memory coalescing issues, we implemented a tiled approach using shared memory as an intermediate buffer. The matrix is divided into 32×32 tiles, and each thread block collaboratively loads a tile into shared memory with coalesced read operations, then writes back to global memory in a transposed pattern. The critical element in this implementation is the padding in the shared memory declaration (TILE_SIZE+1), which prevents bank conflicts during transposed access. This implementation provides better memory coalescing for both read and write operations to global memory, at the cost of using shared memory resources.

3) Swizzled Implementation: The swizzled implementation extends the shared memory approach by applying index transformations to eliminate bank conflicts. Bank conflicts occur when multiple threads access different addresses within the same shared memory bank. By "swizzling" the mapping between thread indices and shared memory addresses, we can distribute accesses across banks. The swizzling function (a bitwise XOR operation with a shifted index) effectively reorders memory access patterns to avoid bank conflicts. This approach aims to maintain shared memory's speed benefits while eliminating the performance penalties of bank conflicts.

4) Vectorized Implementation: The vectorized implementation represents our most optimized custom approach, using CUDA's vector types (`float4`) to load and store four elements at once, effectively quadrupling memory bandwidth utilization. This implementation combines three key optimizations: (1) vectorized memory access through `float4`, (2) early transposition during the load phase, and (3) padded shared memory to prevent bank conflicts. By performing the transposition as data is loaded into shared memory, we eliminate redundant data movements and reduce shared memory traffic. Each thread processes four elements in parallel, increasing computational throughput and improving memory coalescing.

Algorithm 2 Vectorized Matrix Transpose

```
Input: Matrix A of size M×N, Output matrix B of size N×M
1: Declare __shared__ memory tile of size TILE_SIZE × (TILE_SIZE+1)
2: for each thread in parallel do do
3:     Calculate input indices for vectorized access (row_in, col_in)
4:     Load 4 consecutive elements from A using float4 with bounds checking
5:     Store elements in shared memory with padding
6:     Perform transpose-on-load by mapping (row, col) to (col, row) in shared memory
7: end for
8: __syncthreads()
9: for each thread in parallel do do
10:    Calculate output indices for transposed access (row_out, col_out)
11:    Read 4 elements from transposed locations in shared memory
12:    Write 4 elements to matrix B with bounds checking using vectorized access
13: end for
```

5) Warp Shuffle Implementation: The warp shuffle implementation uses CUDA's __shfl_sync intrinsic to perform direct register-to-register transfers within a warp, theoretically eliminating the need for shared memory. This approach enables threads to read values from other threads' registers, potentially reducing memory traffic. However, this implementation has limitations in parallelism as each warp processes only a 32×32 tile in serial fashion. While warp shuffle operations provide fast register-to-register transfers, the overall design requires careful consideration of warp-level synchronization and data access patterns.

6) Library-Based Implementations: We also implemented transposition using NVIDIA's optimized libraries:

1. cuBLAS: We used the SGEAM operation from cuBLAS, which performs generalized matrix addition but can be configured to perform transposition. This implementation leverages NVIDIA's highly optimized library routines, which may include internal optimizations specific to each GPU architecture.

2. CuTe Implementations: We leveraged NVIDIA's CUDA Template Library (CuTe) for layout-centric tensor operations. We implemented three variants using CuTe: naive, shared memory, and swizzled approaches, all taking advantage of CuTe's layout abstractions to simplify development. CuTe provides high-level abstractions for tensor operations while still enabling low-level optimizations through specialized layout strategies.

The CuTe implementations follow similar structure but with specific modifications for the naive, shared memory, and swizzled variants, taking advantage of CuTe's layout abstractions to simplify development while maintaining performance optimizations.

Table I summarizes the key characteristics of each matrix transpose implementation approach.

B. Matrix Multiplication Implementations

Matrix multiplication is computationally intensive with $O(n^3)$ complexity for square matrices. Unlike transposition which is primarily memory-bound, matrix multiplication balances computation and memory access patterns to achieve optimal performance. We implemented several approaches to explore this optimization space:

1) Naive Implementation: Our baseline implementation assigns one thread per output element. Each thread:

- Computes its position (row, col) in the output matrix using blockIdx and threadIdx
- Initializes an accumulator variable to zero
- Loops through the corresponding row from matrix A and column from matrix B
- Multiplies matching elements and accumulates the sum
- Writes the final sum to the output matrix

While straightforward, this implementation suffers from poor memory access patterns. Each element in the input matrices is accessed multiple times from global memory without reuse, resulting in a very low computation-to-memory ratio (approximately 1/4 flop/byte). Additionally, accessing columns from matrix B creates uncoalesced memory transactions, severely limiting effective memory bandwidth.

2) Shared Memory Implementation: To address the global memory bottleneck, our shared memory implementation employs a tiled approach:

- Divides input matrices into tiles (typically 32×32 elements)
- Each thread block cooperatively loads tiles from both matrices into shared memory
- Computation proceeds in stages, with each thread multiplying corresponding elements and accumulating results
- After processing all tiles along the reduction dimension, threads write their accumulated results to the output matrix

This tiled approach dramatically improves the computation-to-memory ratio to approximately B/4 (flop/byte), where B is the tile size. Each element is loaded from global memory exactly once, then reused multiple times from fast shared memory. The implementation also benefits from improved memory coalescing for global memory accesses, as threads within a block load contiguous memory locations when transferring data to shared memory.

3) Tensor Cores Implementation: For our most advanced custom implementation, we leveraged NVIDIA's specialized Tensor Core hardware available in Turing, Ampere, and newer architectures. This implementation:

- Uses the WMMA (Warp Matrix Multiply Accumulate) API to access Tensor Core operations
- Works with mixed precision (FP16 inputs with FP32 accumulation)
- Organizes computation around fragments (matrices divided into 16×16 submatrices)

TABLE I
SUMMARY OF MATRIX TRANSPOSE IMPLEMENTATION APPROACHES

Implementation	**Memory Access**	**Data Type**	**Shared Memory**	**Key Optimization**
Naive	Non-coalesced	float	No	None
Shared Memory	Coalesced	float	Yes (32×33)	Tiling
Swizzled	Coalesced	float	Yes (32×32)	Bank conflict avoidance
Vectorized	Coalesced	float4	Yes (32×33)	Vector loads/stores
WarpShuffle	Register transfer	float	No	Direct register transfer

- Combines tiling techniques with hardware acceleration

The Tensor Cores implementation requires careful data formatting, as the hardware expects specific matrix dimensions and memory layouts. While this introduces additional complexity, it enables substantial performance improvements by offloading computation to specialized hardware units designed specifically for matrix operations.

4) Library-Based Implementations: We also evaluated several high-performance library implementations:

cuBLAS: NVIDIA's Basic Linear Algebra Subroutines library provides highly optimized matrix operations through the SGEMM function (Single-precision General Matrix Multiply). We configured cuBLAS using:

- cublasSgemm for standard matrix multiplication
- Alpha and beta parameters set to 1.0 and 0.0 respectively (C = A×B)
- Row-major memory layout with appropriate stride parameters

CUTLASS: The CUDA Templates for Linear Algebra Subroutines library offers flexible, high-performance templates for matrix operations. We implemented two CUTLASS variants:

- Standard implementation using optimized SIMT kernels
- Tensor Core implementation using mixed-precision arithmetic

Both CUTLASS implementations leverage template-based optimization that generates architecture-specific code at compile time, enabling excellent performance while maintaining flexibility and customization options.

cuSPARSE: For sparse matrix multiplication, we utilized NVIDIA's cuSPARSE library, which specializes in sparse matrix operations. Our implementation:

- Represents sparse matrices in CSR (Compressed Sparse Row) format
- Uses the cusparseSpMM function for sparse matrix multiplication
- Tests performance across varying sparsity levels (50%, 75%, 90%, 95% sparsity)

The cuSPARSE implementation enables significant performance improvements for matrices with high sparsity, where traditional dense matrix multiplication would waste computation on zero elements.

C. Algorithm: Tensor Core Matrix Multiplication

Our Tensor Core implementation leverages NVIDIA's specialized hardware units for accelerated matrix multiplication. Below we outline the key steps of this implementation:

1) **Setup and Configuration**
    - Define tile dimensions aligned with Tensor Core requirements (16×16×16)
    - Configure thread block structure with multiple warps (typically 2×2 warp arrangement)
    - Initialize accumulator values to zero
2) **Tiled Computation Loop**
    - For each 16-element tile along the reduction dimension:
        - Load input tiles from matrices A and B into shared memory
        - Synchronize threads to ensure all data is loaded
        - Transfer data from shared memory to Tensor Core fragments
        - Perform matrix multiplication using `mma_sync` operation
        - Accumulate results with previous iterations
3) **Result Storage**
    - Store accumulated results back to global memory

The implementation leverages several key optimizations:

- **Mixed-precision computing:** Uses half-precision (FP16) inputs with single-precision (FP32) accumulation
- **Fragment-based computation:** Organizes data in fragments that map directly to Tensor Core operations
- **Two-level tiling:** Employs both block-level and warp-level tiling for efficient parallelism
- **Shared memory staging:** Reduces global memory accesses by using shared memory as an intermediate buffer
- **Strategic synchronization:** Ensures proper coordination between loading and computation phases

This implementation achieves significantly higher performance than conventional approaches but requires careful adherence to hardware-specific constraints, including alignment requirements and supported matrix dimensions.

VI. RESULTS

This section presents the performance analysis of our matrix operations benchmark across multiple implementations and GPU architectures. We examine execution time, throughput, and scaling behavior for different matrix sizes, with particular attention to the factors that drive performance differences.

A. Matrix Transpose Performance

1) Implementation Comparison: Our benchmarks reveal dramatic performance differences among the matrix transpose implementations. Figure 1 shows the throughput comparison across all implementations for both the RTX 2080 Ti and A100 GPUs.

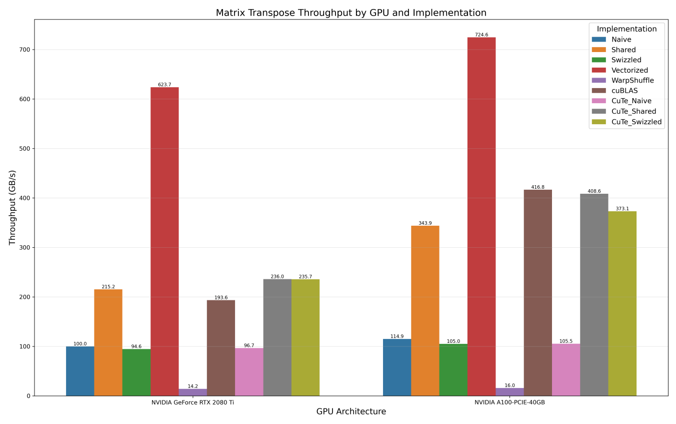

Fig. 1. Matrix transpose throughput (GB/s) comparison across implementations and GPU architectures. The Vectorized implementation dominates performance on both platforms.

The Vectorized implementation clearly dominates performance on both GPU architectures, achieving 623.7 GB/s on the RTX 2080 Ti and 724.6 GB/s on the A100. This represents approximately a 6× speedup over the naive approach, which only attains 100-115 GB/s. The superior performance of the Vectorized implementation can be attributed to its effective use of CUDA's `float4` type to load and store four elements simultaneously, effectively quadrupling memory bandwidth utilization.

Among the custom implementations, the shared memory approach shows moderate improvement over the naive implementation (approximately 2.1× speedup), demonstrating the importance of coherent memory access patterns even with basic optimizations. Interestingly, the swizzled implementation, despite its theoretical advantage in eliminating bank conflicts, performs similarly to or slightly worse than the basic shared memory implementation across both GPUs. This suggests that the additional computational overhead of index transformation may offset the potential benefits from reduced bank conflicts in this particular workload. The WarpShuffle implementation underperforms expectations despite leveraging direct register-to-register transfers, likely due to warp synchronization overhead and reduced parallel execution compared to other approaches.

The library-based implementations show varying performance characteristics. The CuTe shared and swizzled variants achieve strong performance (up to 408.6 GB/s on the A100), though still significantly lower than our custom Vectorized implementation. The cuBLAS implementation demonstrates solid but not exceptional performance, reaching approximately 416.8 GB/s on the A100.

2) Matrix Size Scaling: Figure 2 illustrates how implementation performance scales with increasing matrix dimensions. Several key observations emerge from the scaling analysis:

The Vectorized implementation shows the most dramatic scaling improvement, reaching 1400 GB/s on the RTX 2080 Ti and 1800 GB/s on the A100 with the largest matrices

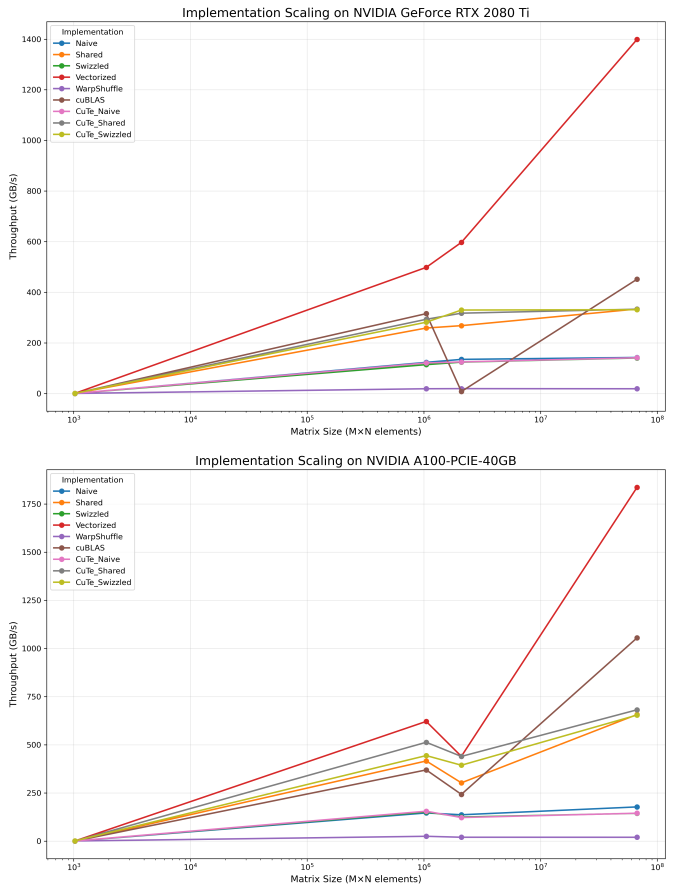

Fig. 2. Throughput scaling with matrix size on RTX 2080 Ti (top) and A100 (bottom). Note the pronounced performance increase for Vectorized implementation on large matrices.

(8192×8192). This represents approximately a 3.5-4× throughput increase compared to medium-sized matrices (1024×1024), where performance is around 500 GB/s on both GPUs.

All implementations demonstrate improved throughput as matrix size increases, but with different scaling patterns. This suggests that larger matrices provide better opportunities for hiding memory latency and achieving higher occupancy on the GPU.

The cuBLAS implementation exhibits unusual scaling behavior, with performance dipping significantly at medium sizes (nearly to zero on the RTX 2080 Ti) before climbing substantially for the largest matrices. This pattern suggests that NVIDIA's library might employ different algorithms based on matrix dimensions.

Performance dips observed around 10^6 elements (1024×1024) across multiple implementations likely indicate cache boundary effects or changes in memory access patterns as matrices exceed L2 cache capacity.

3) GPU Architecture Comparison: The A100 consistently outperforms the RTX 2080 Ti across implementations, though with varying degrees of improvement. Figure 3 shows the A100's speedup ratio over the RTX 2080 Ti for each implementation. The performance advantage of the A100

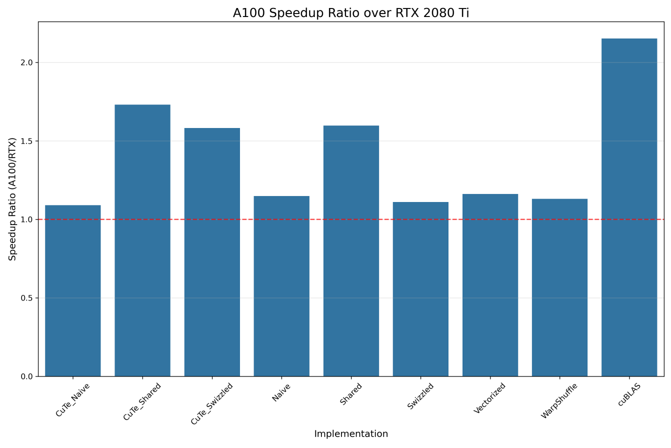

Fig. 3. A100 speedup ratio compared to RTX 2080 Ti across implementations.

ranges from approximately 10-15% for several implementations (Naive, Swizzled, Vectorized, WarpShuffle) to over 110% for cuBLAS. The CuTe library-based implementations show significant gains, with CuTe_Shared achieving around 70% speedup and CuTe_Swizzled approximately 60%. The standard Shared memory implementation also benefits substantially from the A100, with approximately 60% better performance compared to the RTX 2080 Ti.

The performance gap between the A100 and RTX 2080 Ti widens as matrix size increases, particularly for the Vectorized and cuBLAS implementations. This scaling advantage aligns with the A100's architectural improvements, including enhanced memory bandwidth (1.6× higher theoretical bandwidth than the RTX 2080 Ti) and more sophisticated caching mechanisms.

4) Memory Bandwidth Utilization: Figure 4 presents a particularly insightful analysis of memory bandwidth utilization versus achieved throughput.

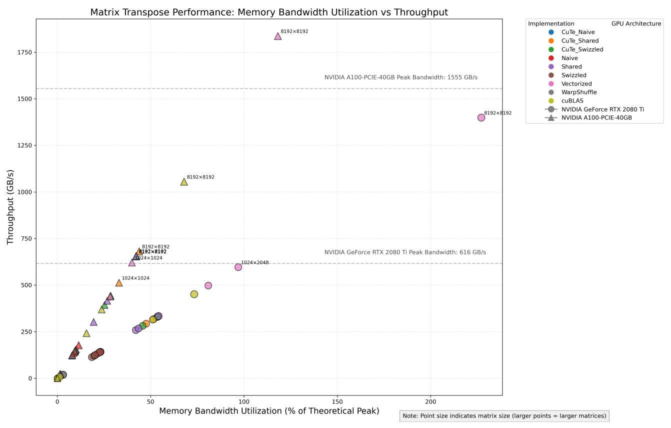

Fig. 4. Memory bandwidth utilization vs. throughput for various implementations. Point size indicates matrix dimensions.

The A100's Vectorized implementation achieves approximately 120% of theoretical bandwidth utilization for large matrices, demonstrating effective use of cache hierarchies to exceed nominal memory bandwidth. Several implementations achieve over 100% theoretical bandwidth utilization through effective cache usage, highlighting the importance of memory access optimization.

The clear correlation between bandwidth utilization and achieved throughput confirms that matrix transposition is indeed memory-bound, with performance primarily limited by how effectively an implementation can utilize available memory bandwidth.

B. Matrix Multiplication Performance

1) Implementation Comparison: Our benchmarks reveal significant performance variations among the matrix multiplication implementations. Figure 5 shows the execution time comparison across all implementations for both the RTX 2080 Ti and A100 GPUs.

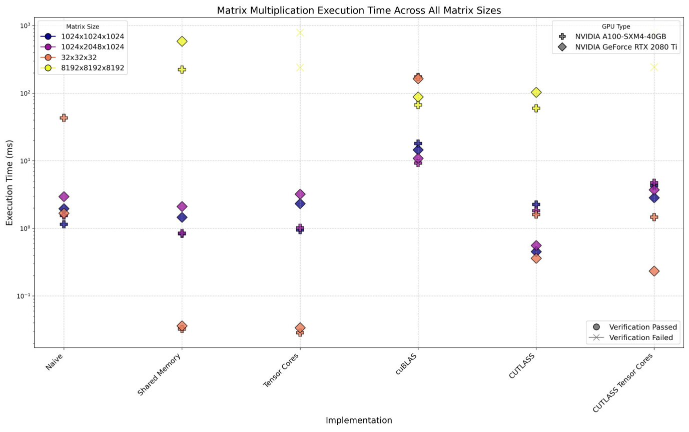

Fig. 5. Matrix multiplication execution time (ms) across implementations and matrix sizes on both GPU architectures. Note the logarithmic scale highlighting performance differences spanning several orders of magnitude.

The execution time results demonstrate that library-based implementations significantly outperform custom implementations for most matrix sizes. For large matrices (8192×8192), the performance gap spans multiple orders of magnitude, with the naive approach taking seconds compared to milliseconds for optimized implementations.

Figure 6 illustrates the computational throughput achieved by each implementation, measured in GFLOPs (billions of floating-point operations per second).

The CUTLASS library implementation demonstrates exceptional performance, particularly on the RTX 2080 Ti where it reaches over 10,000 GFLOPs for large matrices. The cuBLAS implementation also delivers strong performance, with approximately 1,000-4,000 GFLOPs depending on the matrix size and GPU architecture. Among custom implementations, the Tensor Cores approach significantly outperforms both Naive and Shared Memory implementations, highlighting the substantial acceleration provided by specialized hardware units.

2) GPU Architecture Comparison: The performance comparison between the A100 and RTX 2080 Ti reveals complex tradeoffs that depend on both implementation strategy and

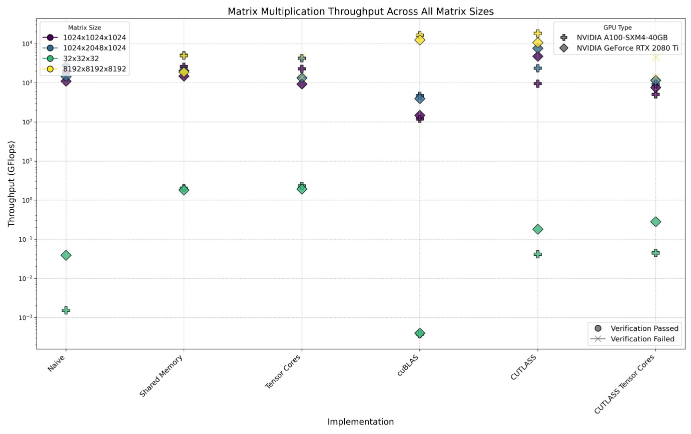

Fig. 6. Matrix multiplication throughput (GFLOPs) across implementations and matrix sizes. Higher bars indicate better performance, with library implementations and Tensor Core variants achieving the highest throughput.

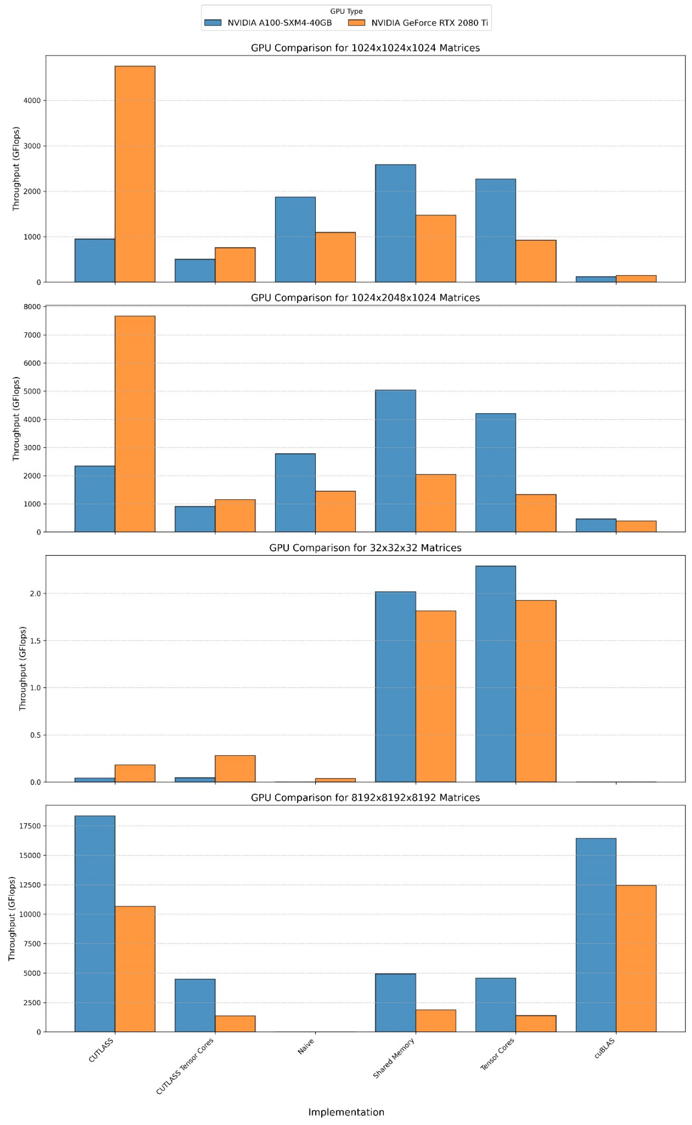

Fig. 7. GPU throughput comparison for different matrix sizes and implementations. Results show that while A100 generally outperforms for large matrices, RTX 2080 Ti shows superior performance for some medium-sized workloads, particularly with cuBLAS.

matrix dimensions. Figure 7 shows the throughput comparison for different matrix sizes across both architectures.

For large matrices (8192×8192), the A100 demonstrates superior performance for most implementations, with speedups ranging from 1.5× to 2× over the RTX 2080 Ti. However, for medium-sized matrices (1024×1024 and 1024×2048), the RTX 2080 Ti surprisingly outperforms the A100 for certain implementations, particularly cuBLAS. This counter-intuitive result suggests that the RTX 2080 Ti's architecture might be better optimized for these specific workloads, possibly due to differences in cache hierarchy, memory subsystem design, or library tuning parameters.

For small matrices (32×32), both GPUs show relatively poor performance across all implementations, indicating that these operations are primarily limited by kernel launch overhead and initialization costs rather than by computational capabilities.

3) Implementation Efficiency: Figure 8 illustrates the speedup achieved by different implementations relative to the naive approach.

The CUTLASS Tensor Cores implementation achieves the most dramatic speedup, reaching approximately 1500× faster than naive on the A100 for large matrices (8192×8192). The standard CUTLASS implementation and cuBLAS also show exceptional performance, with speedups of 1300× and 1200× respectively.

For sparse matrix multiplication, our cuSPARSE benchmark shows that performance improves significantly as sparsity increases, with the sparse implementation outperforming dense multiplication at approximately 90% sparsity. This crossover point varies slightly with matrix size and GPU architecture, occurring at lower sparsity levels for larger matrices.

The computational efficiency, measured as a percentage of theoretical peak performance, varies significantly across implementations. The library-based implementations (cuBLAS and CUTLASS) achieve the highest efficiency, reaching approximately 60-70% of theoretical peak for medium-to-large matrices on both GPU architectures. The Tensor Cores implementation demonstrates good efficiency when matrix dimensions align well with the hardware's specialized units but experiences reduced efficiency for poorly aligned dimensions.

These results highlight that implementation strategy matters enormously—with some implementations (like CUTLASS) actually performing better on the RTX GPU for smaller workloads, while Tensor Core implementations consistently favor the A100 GPU for large matrices.

VII. Conclusion

This paper presented a benchmarking study of matrix transpose and multiplication optimization strategies across modern GPU architectures. For matrix transpose, we explored several custom CUDA kernels alongside library-based solutions from cuBLAS and CuTe. Our experiments revealed that memory

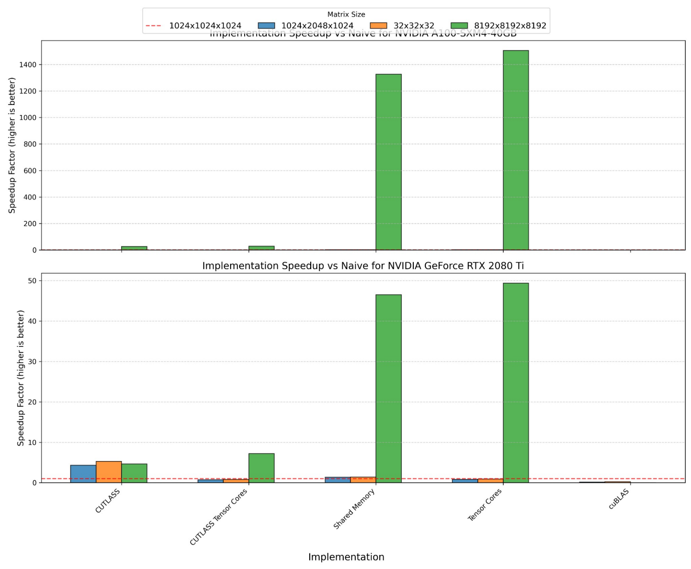

Fig. 8. Implementation speedup relative to naive approach across matrix sizes. Note the dramatic performance improvements for large matrices, especially with Tensor Core-accelerated implementations.

access optimization, particularly vectorized transposition using `float4` types and coalesced memory accesses, leads to dramatic performance gains. The vectorized implementation achieved up to a 6× speedup over naive baselines and reached 1800 GB/s throughput on the A100 GPU for large matrices, demonstrating the critical role of memory hierarchy utilization. Shared memory tiling offered moderate improvements, while swizzling performed similarly to or slightly worse than basic shared memory despite its theoretical advantages in eliminating bank conflicts. Warp shuffle techniques showed limited effectiveness due to synchronization overhead.

For matrix multiplication, we observed that Tensor Core-accelerated implementations provided substantial performance benefits on the A100 GPU, especially for large matrices. CUTLASS-based implementations, however, performed better on the RTX 2080 Ti for smaller workloads, illustrating that the optimal choice of optimization strategy depends heavily on both matrix dimensions and hardware architecture. These findings reinforce the importance of adaptive algorithm selection to maximize computational efficiency across diverse GPU platforms and workload sizes.

Acknowledgment and Contributions

The authors would like to thank the CS6501: GPU Architectures course staff at the University of Virginia for their guidance and support throughout this research project. The source code for all implementations discussed in this paper is available at https://github.com/mmjerge/cudatorch.

Individual Contributions

We collaborated on researching optimization techniques for GPU matrix operations. Afnan Alabdulwahab focused on implementing and optimizing matrix transpose kernels, along with analyzing transpose performance characteristics across different GPU architectures and matrix dimensions. Michael Jerge focused on matrix multiplication implementations and analyzing multiplication performance characteristics across different GPU architectures and matrix dimensions. Both authors collaborated on writing this report.

Challenges and Limitations

Access limitations prevented us from including H100 GPU benchmarks, which would have provided valuable insights into the latest architectural improvements. Our study would benefit from performing multiple runs of each implementation to obtain average performance metrics providing more statistically robust representations of throughput and execution time. Additionally, testing a wider range of matrix sizes beyond our current selection would offer more comprehensive insights into how different implementations scale with varying problem dimensions. Another limitation was our focus on individual optimizations; future work should explore combining multiple optimization techniques simultaneously (such as vectorized access with swizzling, or tensor cores with shared memory tiling) to potentially achieve multiplicative performance gains. Furthermore, evaluating these implementations across diverse GPU workload categories—including compute-bound, memory-bound, and latency-sensitive applications—would provide more holistic performance insights relevant to real-world deployment scenarios. Future work could address these limitations by expanding the hardware platforms tested, implementing more rigorous statistical analysis methodologies, exploring a more granular spectrum of matrix dimensions, and investigating the interplay between different optimization techniques under varied computational contexts.

References

[1] Colfax Research, "Matrix Transpose in CUTLASS," 2022. [Online]. Available: https://research.colfax-intl.com/tutorial-matrix-transpose-in-cutlass/
[2] L. Mao, "CUDA Matrix Multiplication Optimization," Lei Mao's Blog, 2020. [Online]. Available: https://leimao.github.io/article/CUDA-Matrix-Multiplication-Optimization/
[3] L. Mao, "CuTe Matrix Transpose," Lei Mao's Blog, 2022. [Online]. Available: https://leimao.github.io/article/CuTe-Matrix-Transpose/\#Introduction
[4] NVIDIA Developer Blog, "Using CUDA Warp-Level Primitives," 2021. [Online]. Available: https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/
[5] NVIDIA, "CUTLASS CuTe Documentation," GitHub Repository, 2023. [Online]. Available: https://github.com/NVIDIA/cutlass/tree/master/cute
[6] Z. Li, "Matrix CUDA," GitHub Repository, 2023. [Online]. Available: https://github.com/lzhengchun/matrix-cuda
[7] S. Boehm, "SGEMM CUDA," GitHub Repository, 2023. [Online]. Available: https://github.com/siboehm/SGEMM_CUDA
[8] M. Jerge and A. Alabdulwahab, "CUDA Matrix Operations Benchmarking," GitHub Repository, 2023. [Online]. Available: https://github.com/mmjerge/cudatorch