Benchmarking Matrix Operations Optimizations

CS6501: GPU Architectures

Afnan Alabdulwahab | Michael Jerge

https://github.com/mmjerge/cudatorch

AGENDA

01 Introduction

02 Background & Motivation

03 Experimental Setup

04 Results

05 Artifact Overview

06 Final Report Plan & Contributions

07 Conclusion & Questions

01

Introduction

Core

- Fundamental to AI
- GPUs leverage parallel cores for speed
- 70 - 90% of AI computation time

Why Optimize

- Reduces training time
- Lowers computing cost
- Enables larger models

Optimization Methods

- Algorithmic improvements
- Hardware acceleration
- Precision reduction
- Sparsity exploitation

Profiling

- Finds bottlenecks
- Measures GPU utilization
- Guides hardware-specific tuning
- Ensures accuracy

02

Background & Motivation

Current Limitations

- Multihead attention is a fundamental mechanism in transformer architectures but introduces significant computational bottlenecks
- The mechanism requires multiple intensive matrix operations working in parallel
- Key operations (MatMul, Transpose, Softmax) consume substantial computational resources
- These operations scale poorly with increasing model size and sequence length

Expensive Operations

- **Matrix Multiplication (MatMul):** $O(n^3)$ complexity in naïve implementations; dominates computation time
- **Transpose:** Memory access patterns become non-contiguous, reducing cache efficiency
- **Softmax:** Requires element-wise exponential calculations and normalization across dimensions
- Combined operations create memory bandwidth constraints and synchronization overhead

Current Optimization Landscape

- **cuBLAS:** NVIDIA's optimized BLAS library provides efficient matrix operations but has limitations for specialized patterns in attention
- **CUTLASS:** Template library enabling custom matrix operation implementations
- **Tensor Cores:** Hardware acceleration for specific precision formats (FP16, BF16, INT8)
- Despite these tools, optimizing the entire attention mechanism remains challenging due to the interdependencies between operations

03

Experimental Setup

Matrix Transpose Setup

GPU Architectures: NVIDIA GeForce RTX 2080 Ti, NVIDIA A100-PCIe-40GB, (H100 pending)
Matrix Dimensions: 32×32 (**S**), 1024×1024 (**M**), 8192×8192 (**L**), 1024×2048 (**Non-square**)

Implementations Benchmarked

- **Naive:** Direct element-by-element copy with global memory access
- **Shared Memory:** 32×32 tiles with padding to avoid bank conflicts
- **Swizzled:** Shared memory with index transformations to eliminate conflicts
- **Vectorized:** Using float4 for coalesced memory access (4× wider bandwidth)
- **WarpShuffle:** Direct register-to-register transfers using __shfl_sync

Library-based:

- **cuBLAS:** NVIDIA's optimized SGEAM operation
- **CuTe:** NVIDIA's CUDA Template Library implementations (Naive, Shared, Swizzled)

Each kernel is timed using cudaEventRecord()

Verification

- Element-wise comparison with CPU reference implementation
- Complete verification for small/medium matrices
- Statistical sampling with 100+ random points for large matrices (8192×8192)
- All implementations successfully passed verification tests

Performance Metrics

- Kernel execution time (ms)
- Memory throughput (GB/s)
  - Transpose is memory-bound operation (not compute-bound), with performance limited by memory bandwidth
- Percentage of theoretical bandwidth achieved

Matrix Multiplication Setup

GPU Architectures: NVIDIA GeForce RTX 2080 Ti, NVIDIA A100-PCIe-40GB, (H100 pending)
Matrix Dimensions: 32×32 (**S**), 1024×1024 (**M**), 8192×8192 (**L**), 1024×2048 (**Non-square**)

Implementations Benchmarked

- **Custom Kernels:**
  - Naive Implementation
  - Shared Memory Optimization
  - Tensor Cores Implementation
- **Library-based:**
  - cuBLAS
  - CUTLASS
  - cuSPARSE (Sparse Matrix Multiplication)

Verification

Numerical Accuracy Checking

- Full Verification (Small Matrices)
- Partial Verification (Large Matrices)
  - Random Sampling Technique
  - Configurable Sample Count
- Tolerance Levels
  - Standard Implementations: ε = 1e-2
  - Tensor Core Implementations: ε = 1e-1

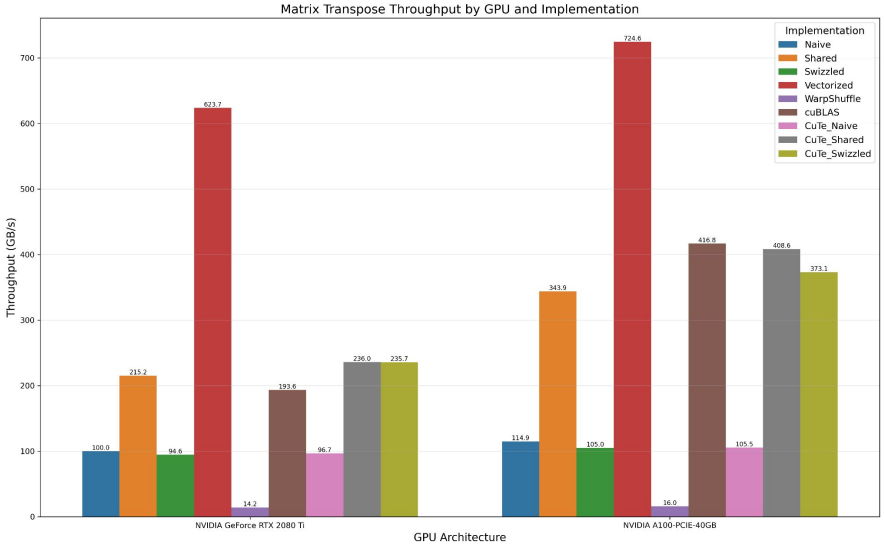
Performance Metrics

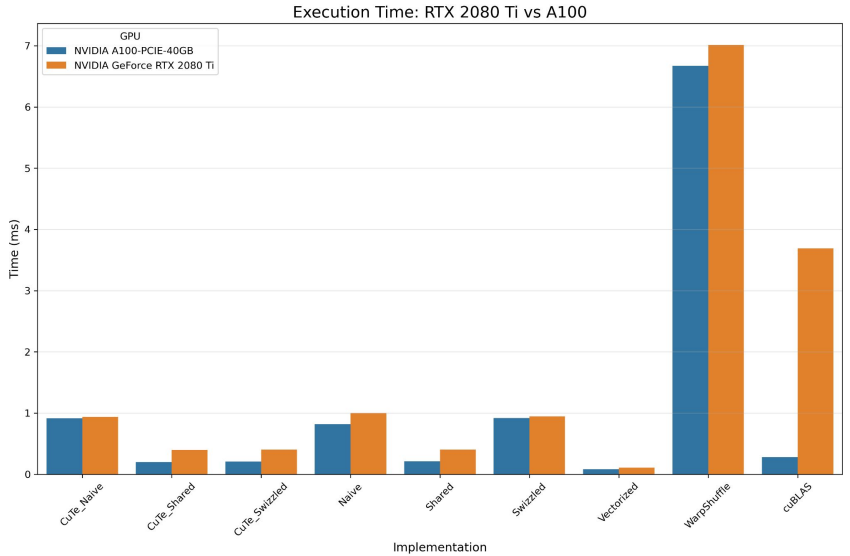
- Execution Time (milliseconds)
- Throughput (GFLOPs)
- Computational Efficiency
- Memory Bandwidth Utilization
- Density Impact (for Sparse Matrices)

04

Results

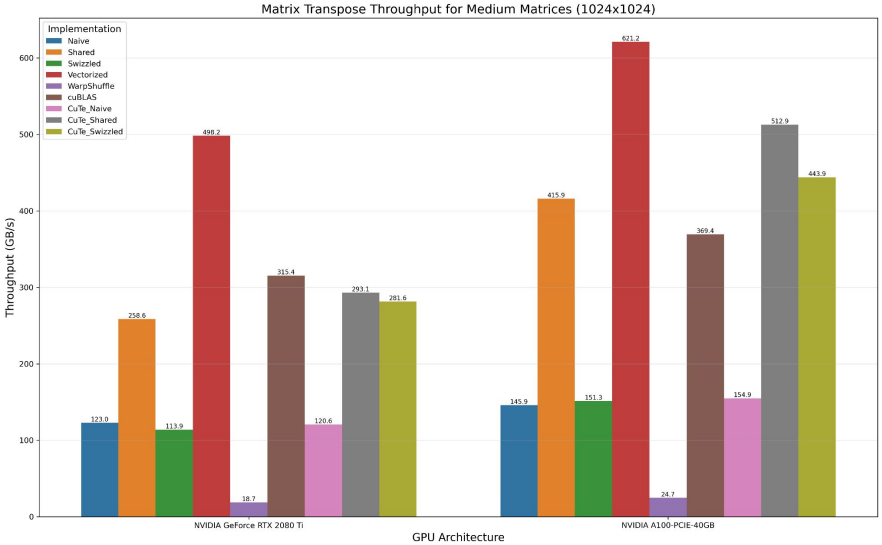
Matrix Transpose Performance: Implementation Comparison & Execution Time

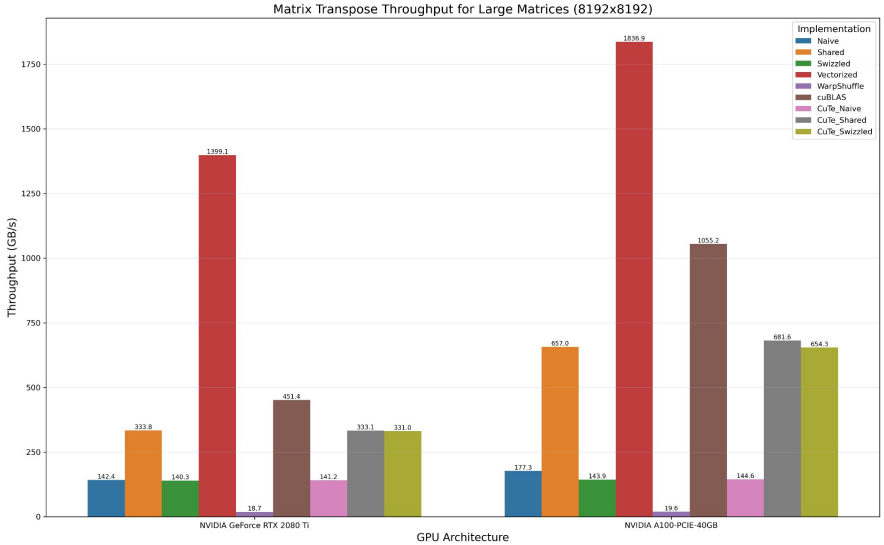




- Vectorized implementation dominates performance (620-720 GB/s), achieving ~6× speedup over naive approach
- A100 outperforms RTX 2080 Ti by 15-50% across most implementations
- WarpShuffle underperforms expectations despite direct register transfers, likely due to warp synchronization overhead and reduced parallelism compared to other approaches

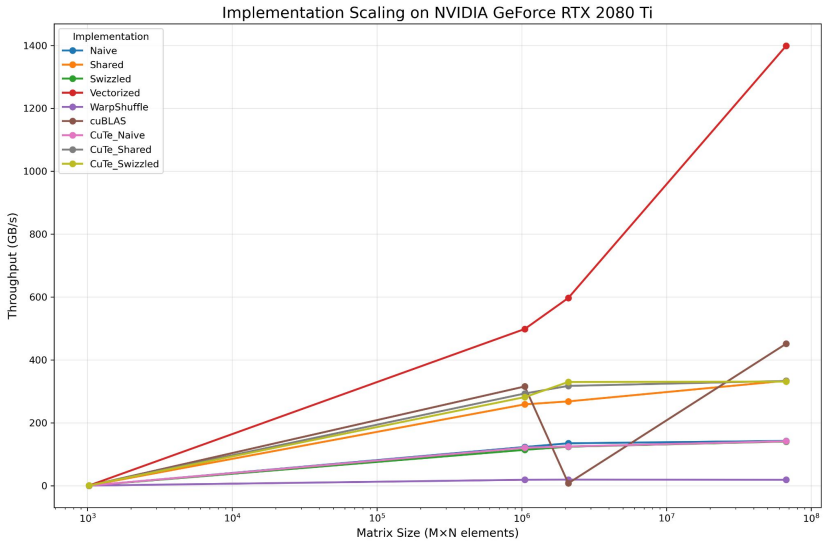
Performance Scaling: Medium vs Large Matrices

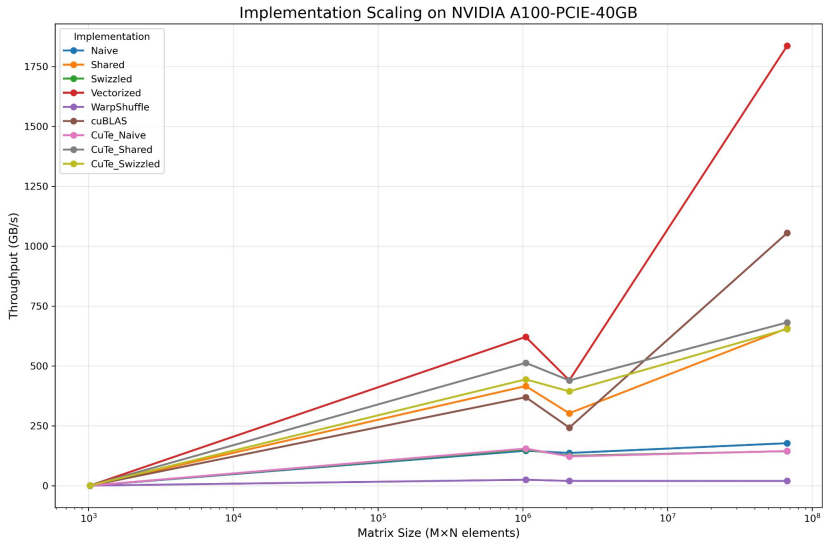




- A100 advantage widens at larger matrix sizes (up to 1800 GB/s for Vectorized implementation)
- Vectorized implementation shows the most dramatic scaling improvement (~3× from medium to large)
- cuBLAS performance becomes more competitive with CUDA Template Library (CuTe) variants at larger sizes

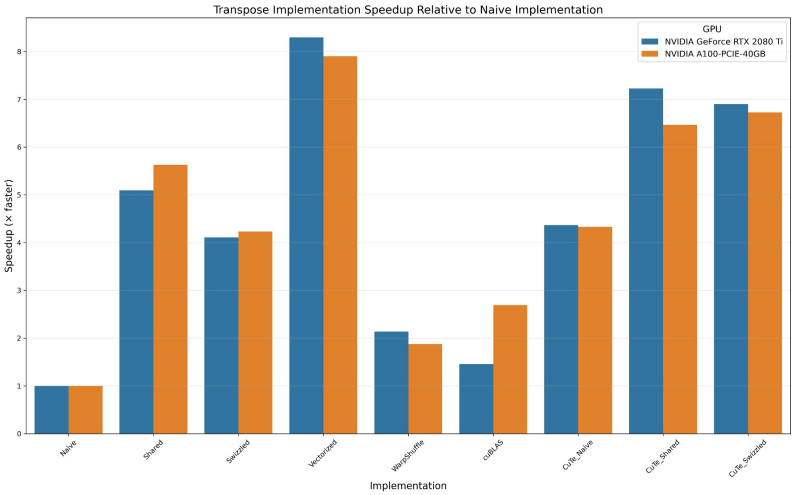
Implementation Scaling Behavior: RTX 2080 Ti vs A100

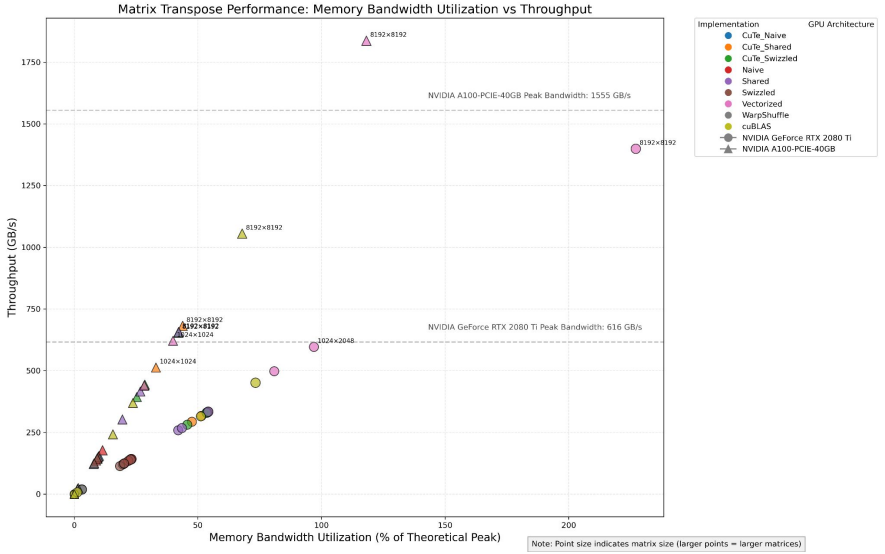




- Vectorized implementation demonstrates superior scaling, reaching 1400 GB/s (RTX 2080 Ti) and 1800 GB/s (A100) with largest matrices
- All implementations show improved throughput as matrix size increases, but with different scaling patterns
- The cuBLAS implementation dips at medium sizes and then climbs significantly for the largest matrices.
    - This suggests NVIDIA's library might use different algorithms based on matrix size.
- The poor performance of WarpShuffle shows that register-based transfers aren't efficient for matrix transpose, likely due to thread synchronization overhead
- The performance dips around 10^6 elements (1024×1024) might indicate cache boundary effects or memory access pattern changes

Implementation Efficiency: Speedup over Naive & Bandwidth Utilization





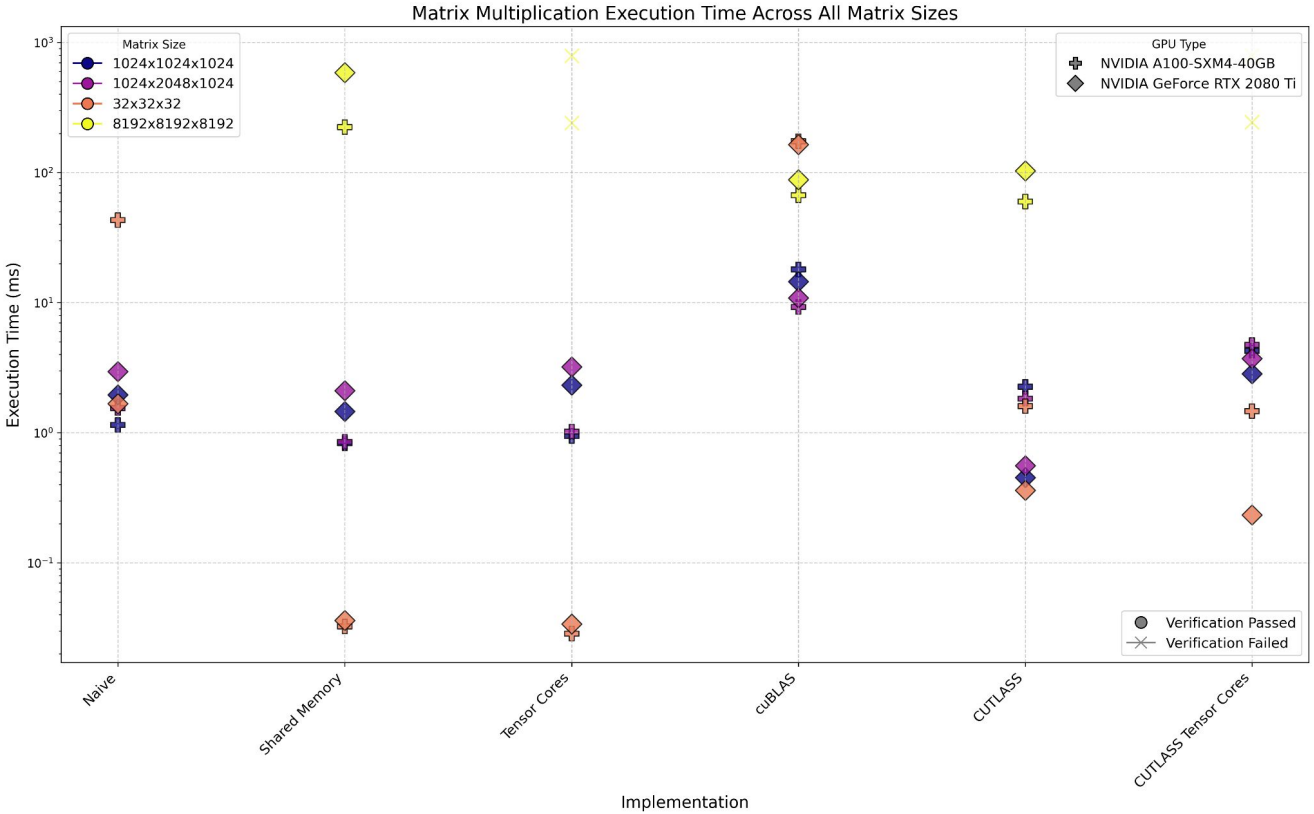
Speedup Findings:

- The Vectorized implementation achieves the most dramatic speedup: ~8.3× faster than naive on RTX 2080 Ti and ~7.9× on A100
- CuTe_Shared and CuTe_Swizzled both show excellent performance (7.2× and 6.9× speedups on RTX 2080 Ti)

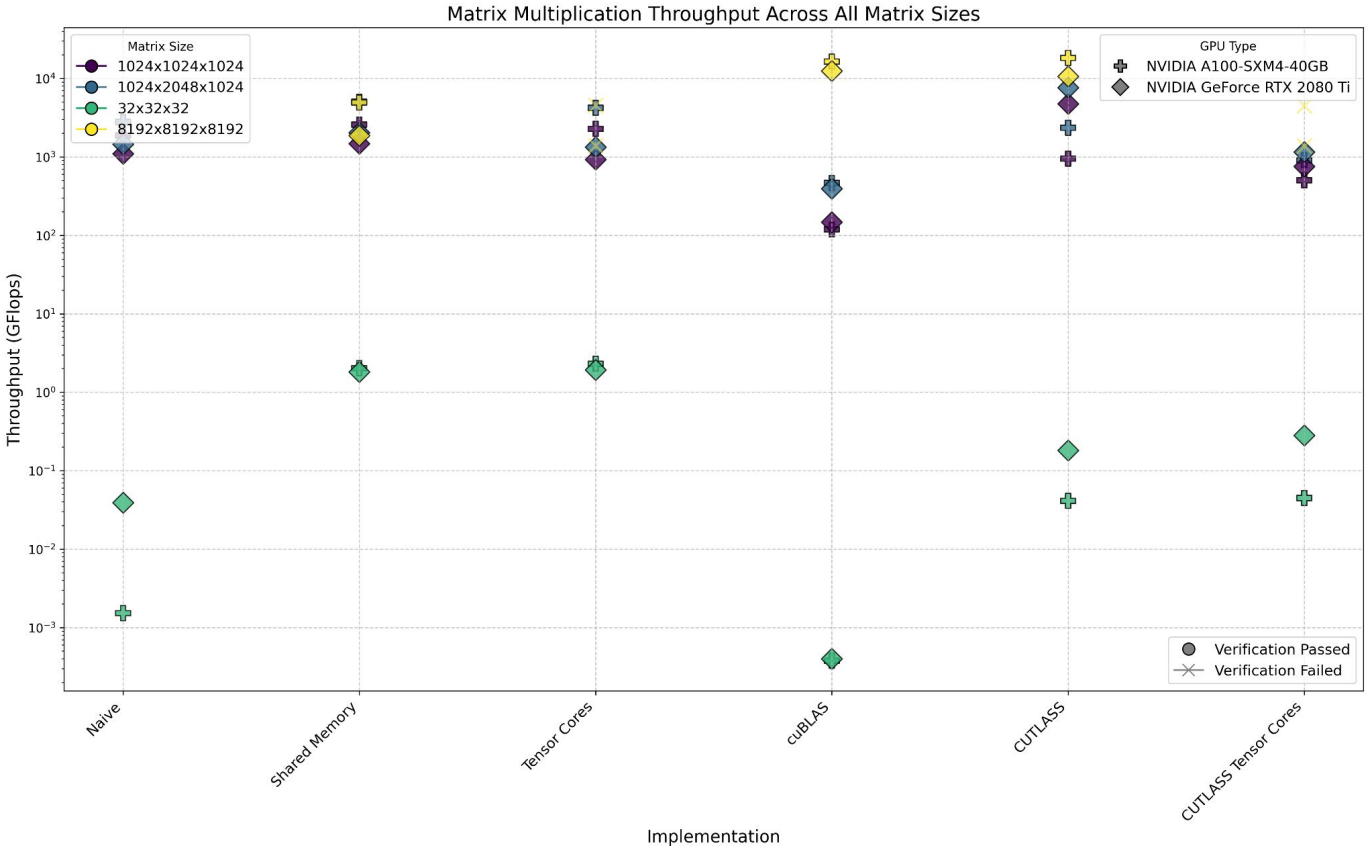
Bandwidth Utilization:

- The A100 Vectorized implementation reaches ~120% of theoretical bandwidth utilization for large matrices
- cuBLAS (yellow points) shows relatively good bandwidth utilization
- Some implementations achieve >100% theoretical bandwidth utilization through effective cache usage
- Clear correlation between bandwidth utilization and achieved throughput
- Memory access pattern optimizations directly translate to performance improvements

Matrix Multiplication Execution Time

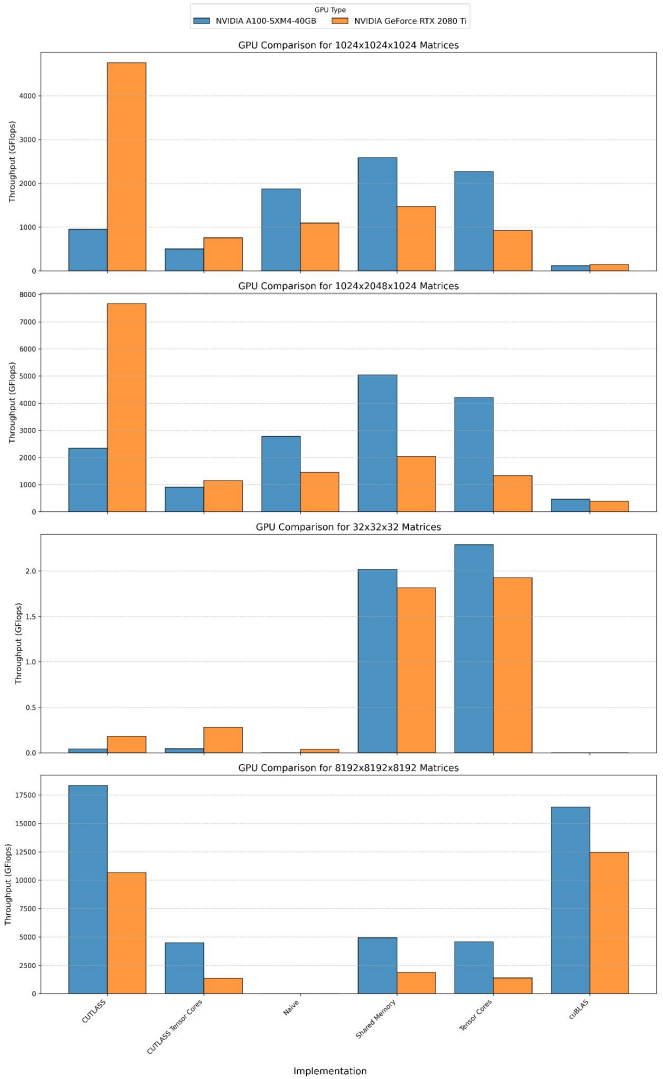


Matrix Multiplication Throughput



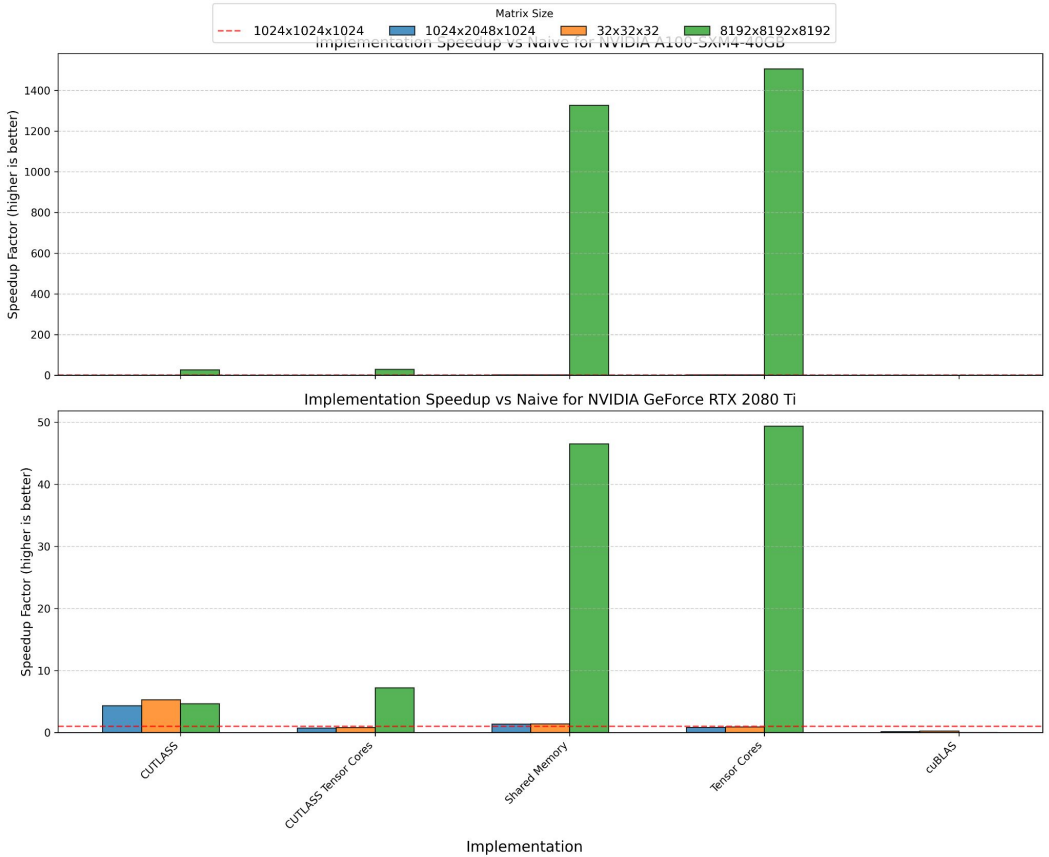
GPU Comparison for Matrix Multiplication

- Detailed throughput comparison between NVIDIA A100 and RTX 2080 Ti
- Multiple matrix sizes and multiplication implementations
- Performance measured in Gigaflops (GFLOPs)



Implementation Speedup vs Naive Approach

- Compares different matrix multiplication implementations
- Speedup calculated relative to naive implementation
- Two GPU architectures: A100 and RTX 2080 Ti



05

Artifact Overview

Repository Structure

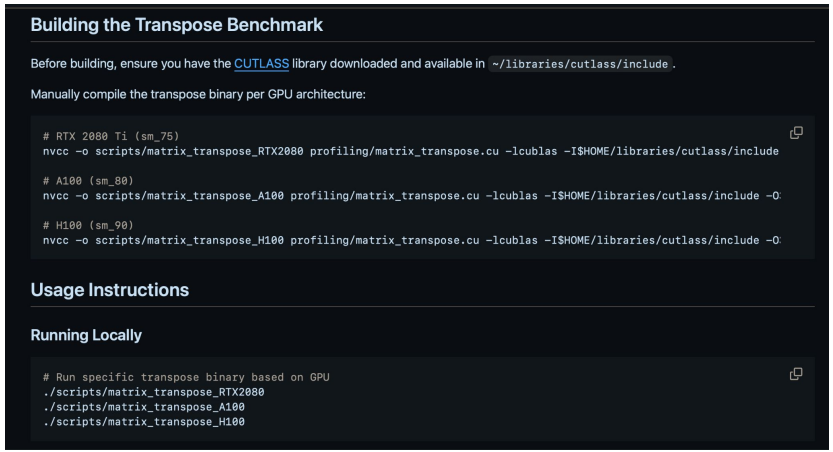
GitHub Repo: https://github.com/mmjerge/cudatorch

Source Files:

- matrix_transpose.cu: Contains all transpose implementations
- matrix_multiplication.cu: Contains all multiplication implementations

Benchmarking:

- Cross-GPU architecture testing (RTX 2080 Ti, A100, H100)
- SLURM job scripts for running tests on multiple GPUs
- Automated verification against CPU reference implementation
- Performance metrics (throughput, execution time) exported to .csv files
- Visualization scripts for generating performance charts



Building the Transpose Benchmark

Before building, ensure you have the CUTLASS library downloaded and available in `~/libraries/cutlass/include`.

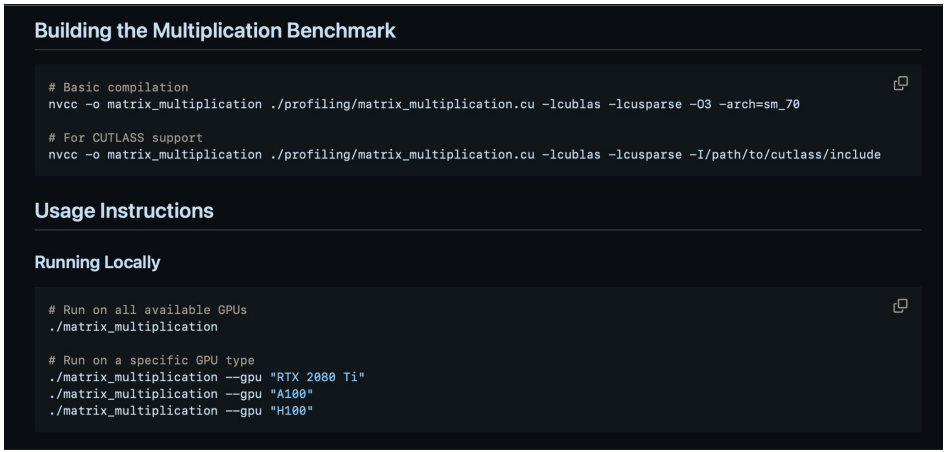
Manually compile the transpose binary per GPU architecture:

```
# RTX 2080 Ti (sm_75)
nvcc -o scripts/matrix_transpose_RTX2080 profiling/matrix_transpose.cu -lcublas -I$HOME/libraries/cutlass/include

# A100 (sm_80)
nvcc -o scripts/matrix_transpose_A100 profiling/matrix_transpose.cu -lcublas -I$HOME/libraries/cutlass/include -O

# H100 (sm_90)
nvcc -o scripts/matrix_transpose_H100 profiling/matrix_transpose.cu -lcublas -I$HOME/libraries/cutlass/include -O
```

Usage Instructions

Running Locally

```
# Run specific transpose binary based on GPU
./scripts/matrix_transpose_RTX2080
./scripts/matrix_transpose_A100
./scripts/matrix_transpose_H100
```



Building the Multiplication Benchmark

```
# Basic compilation
nvcc -o matrix_multiplication ./profiling/matrix_multiplication.cu -lcublas -lcusparse -O3 -arch=sm_70

# For CUTLASS support
nvcc -o matrix_multiplication ./profiling/matrix_multiplication.cu -lcublas -lcusparse -I/path/to/cutlass/include
```

Usage Instructions

Running Locally

```
# Run on all available GPUs
./matrix_multiplication

# Run on a specific GPU type
./matrix_multiplication --gpu "RTX 2080 Ti"
./matrix_multiplication --gpu "A100"
./matrix_multiplication --gpu "H100"
```

Naive Implementations


```
// =============== KERNEL IMPLEMENTATIONS ===============

__global__ void matrixMulNaive(float *a, float *b, float *c, int m, int n, int k) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < m && col < n) {
        float sum = 0.0f;
        for (int i = 0; i < k; i++) {
            sum += a[row * k + i] * b[i * n + col];
        }
        c[row * n + col] = sum;
    }
}
```

```
__global__ void matrixTransposeNaive(float *input, float *output, int m, int n) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < m && col < n) {
        // Input: row-major layout of m×n matrix
        // Output: row-major layout of n×m matrix (the transpose)
        output[col * m + row] = input[row * n + col];
    }
}
```

Transpose Key Implementation Highlight
Vectorized Transpose


```cpp
__global__ void matrixTransposeVectorized4(float *input, float *output, int m, int n) {
    const int TILE_SIZE = 32;
    __shared__ float tile[TILE_SIZE][TILE_SIZE + 1];

    int bx = blockIdx.x, by = blockIdx.y;
    int tx = threadIdx.x, ty = threadIdx.y;

    // Compute input positions
    int row_in = by * TILE_SIZE + ty;           // row index of input matrix
    int col_in = bx * TILE_SIZE * 4 + tx * 4;   // each thread handles 4 elements (vectorized)

    float4 data;

    // load input elements into float4 with bounds checking:
    // 4 contiguous elements from a row in global memory (vectorized access)
    data.x = (row_in < m && col_in + 0 < n) ? input[row_in * n + col_in + 0] : 0.0f;
    data.y = (row_in < m && col_in + 1 < n) ? input[row_in * n + col_in + 1] : 0.0f;
    data.z = (row_in < m && col_in + 2 < n) ? input[row_in * n + col_in + 2] : 0.0f;
    data.w = (row_in < m && col_in + 3 < n) ? input[row_in * n + col_in + 3] : 0.0f;

    // Transpose-on-load: Store in shared memory, effectively transposing during the load
    // The key optimization: swap row and column indices when storing
    // This transforms the data layout from row-major to column-major in shared memory
    tile[ty][tx * 4 + 0] = data.x;
    tile[ty][tx * 4 + 1] = data.y;
    tile[ty][tx * 4 + 2] = data.z;
    tile[ty][tx * 4 + 3] = data.w;

    __syncthreads();

    // Compute output positions (now transposed)
    // Swap block indices to achieve matrix transposition
    int row_out = bx * TILE_SIZE * 4 + tx * 4;
    int col_out = by * TILE_SIZE + ty;

    // If within bounds, write transposed elements back to global memory
    // Each thread writes 4 values down a column — resulting in coalesced writes after the transpose
    if (col_out < m) {
        if (row_out + 0 < n) output[(row_out + 0) * m + col_out] = tile[ty][tx * 4 + 0];
        if (row_out + 1 < n) output[(row_out + 1) * m + col_out] = tile[ty][tx * 4 + 1];
        if (row_out + 2 < n) output[(row_out + 2) * m + col_out] = tile[ty][tx * 4 + 2];
        if (row_out + 3 < n) output[(row_out + 3) * m + col_out] = tile[ty][tx * 4 + 3];
    }
}
```

- Uses float4 to load/store 4 elements at once, effectively quadrupling memory bandwidth
- Performs transposition during initial data loading, eliminating redundant data movements and reducing shared memory traffic
- Combines coalesced global memory access with padded shared memory layout to prevent bank conflicts

Advanced Matrix Multiplication Libraries Example


```
// Initialize cuBLAS handle
cublasHandle_t handle;
cublasCreate(&handle);

// Matrix multiplication parameters
int m = rows_of_A;    // Rows of matrix A
int n = cols_of_B;    // Columns of matrix B
int k = cols_of_A;    // Columns of A / Rows of B

float alpha = 1.0f;   // Scaling factor for A*B
float beta = 0.0f;    // Scaling factor for existing C matrix

// Perform matrix multiplication C = α * (A * B) + β * C
cublasSgemm(handle,
            CUBLAS_OP_N, CUBLAS_OP_N,  // No transpose
            n, m, k,                   // Matrix dimensions
            &alpha,                    // Scaling factor
            d_B, n,                    // Matrix B, leading dimension
            d_A, k,                    // Matrix A, leading dimension
            &beta,                     // Beta
            d_C, n);                   // Result matrix, leading dimension

// Cleanup
cublasDestroy(handle);
```

- Utilizes NVIDIA's optimized BLAS library for GPU-accelerated matrix multiplication
- Performs $C = \alpha(A * B) + \beta(C)$
- Key components:
  1. Create cuBLAS handle
  2. Specify matrix dimensions (m, n, k)
  3. Set scaling factors (α, β)
  4. Call cublasSgemm() to compute matrix product
  5. Destroy handle to release resources

06

Final Plan & Contributions

Plan for the Final Report

1. Complete final benchmark runs:
    - Perform multiple runs (100+) for better statistical representation
    - Attempt benchmarking on one additional GPU architecture (H100 if available)
    - Possibly test with larger matrix sizes
2. Analyze results, finalize figures, and finish drafting the report.

Individual Contributions

We collaborated on researching optimization techniques

- Afnan: Implemented and benchmarked matrix transpose kernels
- Michael: Implemented and benchmarked matrix multiplication kernels

For the final phase, we will collaborate on running remaining benchmarks and divide the report writing equally, with each focusing on our respective matrix operations.

07

Conclusion & Questions

Key Takeaways:

The dramatic performance differences between implementations (up to 8× speedup) demonstrate that optimizing memory access patterns through techniques like vectorization and coalescing is the primary driver of transpose performance.
Implementation strategy matters enormously - with some implementations (like CUTLASS) actually performing better on the RTX GPU for smaller workloads, but Tensor Core implementations consistently favoring the A100 GPU for large matrices.

Questions

- What do our results suggest about the real-world applicability of each implementation? Would certain implementations be better suited for specific application domains or matrix shapes?
- How might specialized matrix multiplication algorithms revolve to better address the growing performance gap between different GPU architectures at scale?

THANK YOU

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