Assignment 3 Report: GPU Optimization of the RMM Algorithm

Ricardo Perello, Alexandre Bourgoin
SCIPER: 357241, 339607

May 19, 2025

1 GPU Implementation of RMM

1.1 Algorithm Description

The Reducing Matrix Multiplication (RMM) algorithm computes matrix C as follows:

$$C_{ij} = \sum_k (A_{ik} \cdot B_{kj}) + A_{ik} - B_{kj}$$

Each element of C is computed independently, making the problem well-suited for parallel implementations on GPUs. The focus of our optimization was leveraging GPU memory hierarchy efficiently.

1.2 Parallelization Strategy

The parallelization strategy distributes the computation of matrix C among CUDA thread blocks. Threads within each block handle computations of distinct elements of C.

Work Splitting Strategies Evaluated:

- **Element-wise Threads**: Each thread calculates exactly one element of matrix C.
- **Tile-based Thread Blocks**: Threads within a block collaboratively load tiles of matrices A and B into shared memory and compute multiple elements of C in parallel.

The tile-based approach was chosen due to improved data reuse and reduced global memory latency.

2 Applied Optimizations

We incrementally applied several optimizations and measured their impact:

2.1 Global Memory (Baseline)

This approach directly accesses global memory without optimization. While easy to implement, it suffers from high global memory latency.

2.2 Shared Memory Tiling

We partition matrices into tiles, loading data into shared memory before computations. Shared memory access is significantly faster, reducing latency dramatically.

2.3 Loop Unrolling

We manually unrolled loops in kernel code to improve instruction-level parallelism and reduce branch overhead.

2.4 Pinned Memory

Using pinned (page-locked) memory for host allocations aimed to speed up host-device data transfers. However, it showed mixed results due to overhead from memory management.

3 Experimental Results and Analysis

3.1 CPU vs GPU Comparison

Figure 1 illustrates the dramatic performance advantage GPUs have over CPUs, especially with larger matrices. The GPU scales much better, achieving significant speedups for matrix sizes beyond 1024^3.

3.2 Detailed GPU Performance

Figure 2 shows GPU execution times in detail. Shared memory consistently improves performance, while loop unrolling provides additional incremental improvement. The pinned memory implementation shows mixed results due to increased overhead in smaller and very large sizes.

3.3 Speedup over Global Memory

Figure 3 presents the speedup achieved with different optimizations relative to global memory baseline. Shared memory combined with loop unrolling consistently provides best results, especially at larger scales.

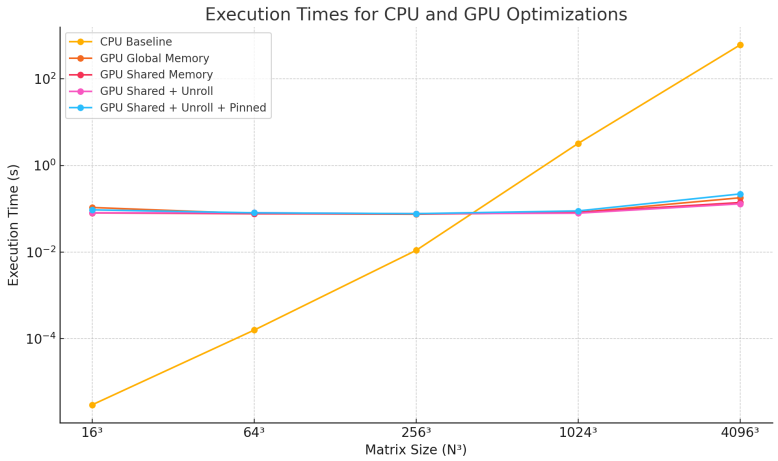

Figure 1: Execution times (log scale) for CPU baseline and GPU versions.

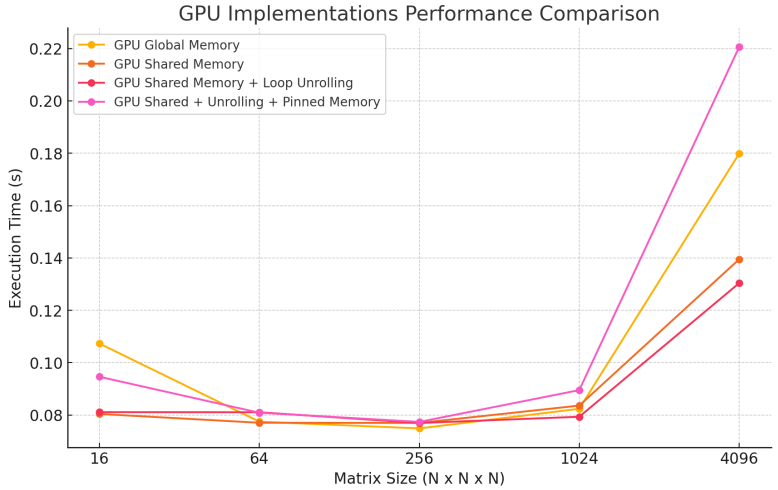

Figure 2: GPU execution times for different optimizations.

4 Discussion of Results

The experimental results highlight key points about the optimization strategies:

- **Global Memory Bottlenecks**: Direct global memory access severely limits performance due to latency and bandwidth constraints.
- **Shared Memory Effectiveness**: Utilizing shared memory achieves up to approximately 40% speedup, highlighting its importance for data locality and reducing memory latency.

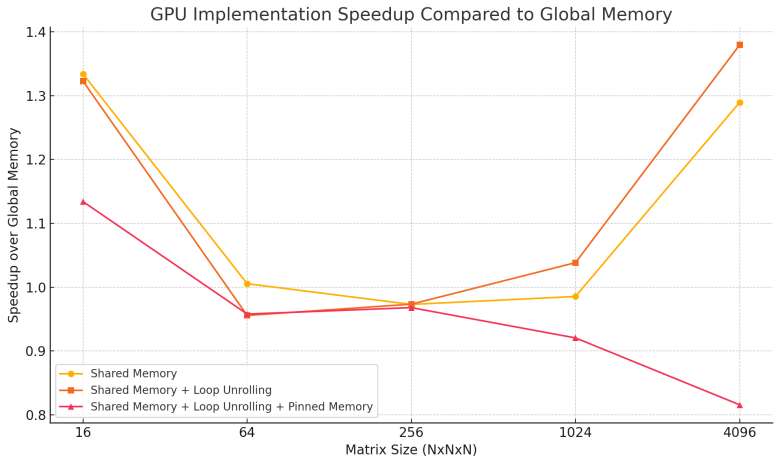

Figure 3: GPU speedups compared to global memory.

- **Loop Unrolling Benefits**: Although loop unrolling is a minor optimization, it incrementally reduces overhead in kernels, contributing positively to overall execution times.
- **Pinned Memory Considerations**: Pinned memory is beneficial for larger transfers (sizes around 1024^3), but overhead negates gains for small or very large matrices (4096^3).

5 Conclusion

We evaluated various GPU optimization techniques for the RMM algorithm, identifying shared memory tiling combined with loop unrolling as the optimal strategy. While pinned memory showed limited effectiveness, the insights provided by this experimentation can guide future performance tuning.

Our experiments underline the importance of memory optimization, data locality, and efficient parallelization strategies to achieve maximum GPU performance.