Linnéuniversitetet
Kalmar Växjö

Assignment 3

Optimizing CUDA Matrix Multiplication

Författare: Razan Kngo, Rula Nayf, Ferzend Shekhow
Termin: VT25
Ämne: Parallel and Distributed Computing
Kurskod: 4DT906

1. Introduction

In this assignment, we implemented and optimized a matrix multiplication kernel in CUDA. Starting from the straightforward three-loop implementation (Problem 1), we explored a series of performance improvements: privatization, memory coalescing, thread coarsening, tiled shared memory, and combinations of them. The goal was to investigate how each optimization affects runtime for a fixed matrix size and draw conclusions about their relative impact.

2. Problem Definition and Baseline

Problem Statement: Compute the product **C = A × B** for two NxN matrices using CUDA.
Baseline Implementation: The standard three-nested-loops version, with each thread computing one element of C. This version serves as our performance baseline.

3. Optimization Strategies

We applied the following techniques, each designed to address specific GPU performance:

1. **Privatization:** Each thread allocates private registers for intermediate sums to reduce global memory accesses.

2. **Memory Coalescing:** Arrange global memory loads so that consecutive threads access consecutive memory addresses.

3. **Thread Coarsening:** Each thread computes multiple output elements (by strip-mining the output space), reducing overhead from thread launches and control divergence.

4. **Tiled Shared Memory:** Load blocks (tiles) of A and B into shared memory to exploit data reuse and reduce redundant global memory transactions.

5. **Combined Tiling + Coarsening:** Integrate tiled shared memory loading with thread coarsening to maximize locality and work per thread.

The CUDA file A3.cu implements each variant and measures runtime using CUDA events.

4. Experimental Setup

- **Hardware:**
  - NVIDIA GPU Quadro P4000
  - Compute Capability: 6.1
  - Total Global Memory: 8106 MB
  - Shared Memory per Block: 49152 bytes
  - Registers per Block: 65536
  - Warp Size: 32
  - Max Threads per Block: 1024
  - Max Threads Dim: [1024, 1024, 64]
  - Max Grid Size: [2147483647, 65535, 65535]
  - Clock Rate: 1480 MHz
  - MultiProcessor Count: 14
- **Matrix Size:** N=1024.
- **Tile Width:** 16.
- **Thread-Block Size:** 16×16 threads.
- **Coarsening Factors Tested:** 1, 2, 4, and 8.

Results were collected by running each kernel variant multiple times and averaging the measured GPU execution time. The raw timings are recorded in Experiments.md.

5. Results

Table 1. Execution times in seconds for various optimizations at coarsening width = 1, tile width = 16.

Variant	Time
Privatization Only	0.0166185
Memory Coalescing	0.0403925
Thread Coarsening	0.0165212
Tiled Shared Memory	0.0218448
Tiled + Coarsening	0.0208169

Table 2. Execution times at coarsening factors 2, 4, 8.

Coarsening	Privatization	Coalescing	Coarsening Only	Tiled SM	Combined
2	0.0166175	0.0378378	0.0188396	0.0203909	0.0193956
4	0.0166093	0.0404664	0224572	0.0219044	0.0184074
8	0.0166195	0.0405831	0.0346839	0.0218358	0.0188979

Table 3. Influence of tile width for coarsening factor = 1 at width = 32.

Variant	Time (s)
Privatization Only	0.0166963
Memory Coalescing	0.0580710
Thread Coarsening	0.0147374
Tiled Shared Memory	0.0190966
Tiled + Coarsening	0.0191201

6. Discussion and Interpretation

- **Privatization** runs steadily at about 0.0166 s, no matter the coarsening level. This shows that using registers for temporary values is fast and doesn't slow things down.
- **Memory Coalescing** by itself is slow. Even though reading data in order helps, the performance stays low because the same data is read many times without being reused.
- **Thread Coarsening** works best when the width is 1×32, reaching around 0.0147 s. This is because each thread does more work, which reduces the time spent starting and managing threads.

- **Tiled Shared Memory** improves over coalescing, confirming shared tiles effectively reuse data but incur synchronization overhead.
- **Combined Tiling + Coarsening.** The tiling + coarsening version is the most balanced, usually running in ~0.018–0.019 s. It may not always be the fastest, but it's more reliable than the others because it reuses data well and gives each thread more work.
- **Tile Width Variation.** Changing the tile width from 16 to 32 doesn't affect tiling much, but it makes coalescing-only much slower at larger widths, probably because it overloads the cache.

7. Conclusion

This assignment showed how different CUDA optimization strategies can greatly improve the speed of matrix multiplication. By starting with a simple version and adding one optimization at a time like privatization, memory coalescing, thread coarsening, and shared memory tiling we were able to see how each technique affects performance.

Privatization gave some of the best speedups, especially when using registers to hold temporary values. This made the code fast and efficient without much extra effort. Thread coarsening also helped a lot by giving more work to each thread, which reduced the time wasted on launching and managing too many threads.

The most reliable results came from combining tiling with coarsening. This method reused data effectively and balanced the workload well across the GPU. It wasn't always the absolute fastest, but it performed well across different settings, making it a strong overall choice.