Assignment 3: CPU vs GPU Performance Analysis with Memory-Centric Workloads

Sruthi Manepalli

1 Overview

This assignment investigates how memory behavior affects performance on GPUs and how this differs from CPU execution. While GPUs provide massive parallelism, their performance is often limited by memory access patterns rather than computation. Through a series of experiments, this assignment studies global memory access patterns and shared memory optimization.

2 Task 1: Global Memory Access Patterns

The objective of this task is to study the impact of memory coalescing on GPU performance. Two CUDA kernels were implemented that perform the same computation but differ only in how global memory is accessed.

2.1 Task 1.1: Coalesced Access Kernel

In the coalesced kernel, consecutive threads access consecutive memory locations. A one-dimensional grid and block layout was used, and bounds checking was implemented to ensure correctness. This access pattern allows memory requests from threads within a warp to be combined into a small number of global memory transactions, maximizing bandwidth utilization.

2.2 Task 1.2: Non-Coalesced Access Kernel

In the non-coalesced kernel, the computation performed by each thread remains identical to the coalesced version. However, memory accesses are intentionally strided such that consecutive threads access non-adjacent memory locations. The number of threads, arithmetic operations, and memory operations per thread were kept identical to ensure a fair comparison.

2.3 Task 1.3: Results

Table 1 summarizes the measured execution times for both kernels.

Table 1: Task 1: Coalesced vs Non-Coalesced Global Memory Access

Kernel Type	Execution Time (ms)	Relative Performance
Coalesced Access	*0.365443*	Faster
Non-Coalesced Access	*4.7251*	Slower

2.4 Task 1.4: Discussion and Justification

The coalesced kernel consistently executes faster than the non-coalesced kernel. This is because, in the coalesced case, threads within a warp access contiguous memory addresses, allowing the GPU to service these requests using fewer global memory transactions. In contrast, the strided access pattern in the non-coalesced kernel prevents efficient coalescing, resulting in multiple memory transactions per warp and increased memory latency.

Since the computation and number of memory operations are identical in both kernels, the observed performance difference can be attributed solely to memory access patterns. This experiment clearly demonstrates that memory coalescing is a critical factor in achieving high GPU performance.

3 Task 2: Shared Memory Optimization

This task explores how shared memory can be used to reduce global memory traffic for computations that exhibit data reuse.

3.1 Task 2.1: Baseline Kernel Using Global Memory

The baseline kernel computes a simple sliding window operation:

$$y[i] = x[i-1] + x[i] + x[i+1]$$

Each thread independently loads the required input values directly from global memory. Because adjacent threads access overlapping input elements, the same global memory locations are repeatedly accessed, resulting in redundant global memory traffic.

3.2 Task 2.2: Optimized Kernel Using Shared Memory

In the optimized version, each block first loads its required input data into shared memory. After synchronization using `__syncthreads()`, threads perform the sliding window computation using shared memory instead of global memory. This reduces the number of global memory accesses and exploits fast on-chip shared memory for data reuse.

3.3 Task 2.3: Results

Table 2 summarizes the execution times for the global memory and shared memory kernels.

Table 2: Task 2: Global Memory vs Shared Memory Implementation

Kernel Type	Execution Time (ms)	Speedup
Global Memory Kernel	*16.8835*	1.0×
Shared Memory Kernel	*0.28144*	59.9897×

3.4 Task 2.4: Discussion and Justification

The shared memory implementation achieves lower execution time compared to the global memory baseline. This improvement occurs because shared memory has significantly lower latency than global memory and allows input data to be reused efficiently within a thread block.

In the global memory kernel, each thread independently loads overlapping data from global memory, resulting in redundant memory accesses. In contrast, the shared memory kernel loads each input element only once per block and reuses it across multiple threads. This reduction in global memory traffic leads to improved performance.

4 Task 3: CPU Baseline for 2D Convolution

As required, a CPU baseline was implemented for the 2D convolution workload introduced in Assignment 1. The convolution was computed using a direct spatial approach with nested loops and a fixed 3×3 kernel.

4.1 CPU Implementation and Timing

Boundary pixels were excluded to avoid out-of-bounds accesses. The execution time was measured using Python's `time.perf_counter()` function.

For an input image of size 2048×2048, the measured CPU execution time was:

CPU 2D convolution runtime: *30.864006089000043* seconds

4.2 CPU vs GPU Discussion

The CPU executes the convolution sequentially with limited parallelism and relies on cache hierarchies to exploit data locality. In contrast, the GPU launches thousands of threads concurrently, enabling parallel computation of output elements. However, GPU performance is strongly dependent on memory access patterns, as demonstrated in Tasks 1 and 2.

5 Conclusion

This assignment demonstrates that GPU performance is highly sensitive to memory behavior. Coalesced global memory access significantly improves performance, while shared memory further reduces global memory traffic by exploiting data reuse. The CPU baseline highlights the limitations of sequential execution for memory-intensive workloads such as convolution. Together, these experiments provide a clear understanding of why memory-aware programming is essential for achieving high performance on GPUs.