Week 3 Assignment Report
GPU Memory Hierarchy & Performance Optimization

Anshika Jain
GPU Programming using CUDA and Triton (WiDS'25)

1 Introduction

The objective of this assignment is to study GPU memory behavior and understand how memory access patterns, shared memory usage, and kernel design impact performance. Experiments were conducted using CUDA to compare coalesced and non-coalesced memory accesses, evaluate shared memory optimization, and establish a CPU baseline for a real-world workload.

The chosen workload for the CPU baseline is a U-Net style convolutional neural network forward pass used in medical image segmentation, which is computationally intensive and memory-bound, making it suitable for GPU acceleration.

2 Task 1: Global Memory Access Patterns

2.1 Coalesced Memory Access

In the coalesced kernel, consecutive CUDA threads access consecutive global memory locations. Each thread processes one element of the input array, ensuring that threads within a warp generate contiguous memory requests.

This access pattern allows the GPU to combine multiple memory requests into fewer memory transactions, maximizing memory bandwidth utilization.

2.2 Non-Coalesced Memory Access

In the non-coalesced kernel, threads access memory with a fixed stride greater than one. Although the computation performed by each thread is identical to the coalesced version, the strided access pattern causes threads in the same warp to access non-adjacent memory locations.

As a result, the GPU must issue multiple memory transactions per warp, increasing memory latency and reducing throughput.

2.3 Performance Comparison and Analysis

Table 1 summarizes the observed execution behavior.

Kernel Type	Access Pattern	Relative Performance
Coalesced	Contiguous	Fast
Non-Coalesced	Strided	Slow

Table 1: Comparison of coalesced and non-coalesced memory access

The performance difference arises from warp-level memory access behavior. Coalesced accesses minimize the number of global memory transactions, while non-coalesced accesses waste memory bandwidth due to scattered loads.

Kernel Type	Execution Time (ms)
Coalesced	0.880576
Non-Coalesced	1.10883

Table 2: Execution time comparison of coalesced and non-coalesced global memory access kernels

3 Task 2: Shared Memory Optimization

3.1 Baseline Kernel Using Global Memory

The baseline kernel performs a reduction operation directly using global memory. Each thread loads a value from global memory and performs an atomic addition to accumulate the result.

This approach suffers from high global memory latency and serialization due to frequent atomic operations.

3.2 Optimized Kernel Using Shared Memory

In the optimized version, data is first loaded into shared memory. Threads within a block collaboratively reduce the data using shared memory, and only one atomic operation per block is issued to global memory.

Synchronization using `__syncthreads()` ensures correctness during the reduction process.

3.3 Performance Analysis

Shared memory significantly improves performance by:

- Reducing global memory accesses
- Exploiting data reuse within a block
- Minimizing atomic operation contention

Although shared memory can suffer from bank conflicts, the chosen reduction pattern minimizes such conflicts, resulting in substantial speedup over the global-memory-only approach.

4 Task 3: CPU Baseline for Medical Image Segmentation

4.1 Workload Description

The CPU baseline implements a U-Net style encoder–decoder forward pass for medical image segmentation. The model processes a 256×256 grayscale medical image and produces multiple feature maps using convolution, pooling, upsampling, and skip connections.

This workload is representative of real medical imaging applications such as tumor or organ segmentation and is both memory-intensive and computation-heavy.

4.2 CPU Implementation

The CPU baseline is implemented using NumPy with explicit nested loops. A custom 2D convolution with ReLU activation is used, followed by max pooling, upsampling, and a skip connection. The implementation is fully sequential, providing a reference runtime for later GPU acceleration.

4.3 Relevance for GPU Acceleration

Convolutional neural networks involve high data movement and parallel computation, making them ideal candidates for GPU execution. The same workload can be accelerated on the GPU using custom CUDA kernels or optimized libraries in subsequent weeks.

5 Discussion

The experiments highlight the importance of memory access patterns and memory hierarchy in GPU programming. While GPUs offer massive parallelism, performance gains depend heavily on coalesced memory access, effective use of shared memory, and sufficient workload size to amortize kernel launch overhead.

For small workloads, CPU execution may outperform GPU execution due to lower overhead and better cache utilization.

6 Conclusion

This assignment demonstrated key GPU programming concepts including memory coalescing, shared memory optimization, and fair CPU–GPU performance comparison. The CPU baseline establishes a realistic reference for evaluating GPU acceleration in future assignments. Overall, the experiments reinforce that understanding memory behavior is crucial for achieving high performance on GPUs.