Week 3 Assignment: GPU Memory Optimization

Krishna Kukreja

January 2, 2026

1 Introduction

In this assignment, we explored the impact of GPU memory hierarchy on kernel performance. Specifically, we investigated:

1. The performance difference between coalesced and non-coalesced global memory access.
2. The speedup achieved by using Shared Memory tiling for Matrix Multiplication.
3. Establishing a CPU baseline for the final course project (Edge AI NPU Simulation).

All experiments were conducted on Google Colab using an **NVIDIA T4 GPU**.

2 Task 1: Global Memory Access Patterns

2.1 Experiment Description

We implemented two kernels to process a large array ($N = 16$ million float elements):

- **Coalesced Kernel:** Threads access consecutive memory addresses (idx).
- **Non-Coalesced Kernel:** Threads access memory with a stride of 32 ($idx \times 32$).

The goal was to measure the "Effective Bandwidth" and understand the penalty of strided access.

2.2 Results

Table 1: Memory Coalescing Analysis (N=16,777,216 elements)

Access Pattern	Stride	Time (ms)	Bandwidth (GB/s)
Coalesced	1	0.583	**230.10**
Non-Coalesced	32	0.565	7.43

2.3 Analysis

The results demonstrate a massive discrepancy in effective bandwidth.

- The Coalesced kernel achieves ≈ 230 GB/s, which is close to the theoretical peak of the T4 GPU.
- The Non-Coalesced kernel drops to just 7.43 GB/s. Even though the execution time appears similar, the strided kernel processes $32\times$ less data per transaction.
- **Conclusion:** Accessing global memory in a strided pattern wastes approximately 97% of the available memory bandwidth because the GPU loads full 128-byte cache lines but only utilizes 4 bytes from each.

3 Task 2: Shared Memory Optimization

3.1 Experiment Description

We implemented Matrix Multiplication ($C = A \times B$) for matrix size $N = 2048$ using two approaches:

- **Naive Implementation:** Relies entirely on Global Memory reads for every dot product operation.
- **Tiled Implementation:** Loads 32×32 blocks of data into Shared Memory to reuse data and reduce global memory traffic.

3.2 Results

Table 2: Matrix Multiplication Performance (N=2048)

Implementation	**Execution Time (ms)**	**Speedup**
Naive (Global Memory)	53.718	1.0x
Tiled (Shared Memory)	41.965	**1.28x**

3.3 Analysis

The Shared Memory implementation provided a 1.28x speedup. By loading data into shared memory, we reduced the latency of repeated accesses to the matrix elements. While the speedup is positive, it could likely be increased further by optimizing the thread-to-work ratio or adjusting the tile size to better match the T4 architecture.

4 Task 3: Project CPU Baseline

4.1 Workload Description

For the final project, I have selected the Edge AI NPU Simulation. The core bottleneck is a Matrix-Vector Multiplication (GEMV) representing a linear layer in a neural network.

- **Batch Size (M):** 1 (Edge Inference scenario)
- **Input Features (K):** 4096
- **Output Features (N):** 4096

4.2 Baseline Measurement

We measured the runtime of this operation on the CPU (using NumPy) to serve as a baseline for future GPU optimization.

Table 3: CPU Baseline Results

Configuration	**Value**
Input Dimensions	1×4096
Weight Matrix	4096×4096
CPU Runtime	**4.5674 ms**

This baseline of ≈ 4.5 ms gives us a clear target to beat in Weeks 4-6 using CUDA.

5 Conclusion

This week's experiments confirmed that memory bandwidth is often the primary bottleneck in GPU programming. We observed that ensuring coalesced access is critical for utilizing the GPU's full potential (230 GB/s vs 7 GB/s), and that using Shared Memory can provide immediate speedups (1.28x) for memory-intensive workloads like Matrix Multiplication.