GPU Memory Hierarchy & Performance Optimization

Edge AI NPU Engine: GEMV with ReLU Activation

Week 3 Assignment Report

GPU Programming using CUDA and Triton (WiDS'25)
IIT Bombay, Department of Electrical Engineering

January 4, 2026

Abstract

This report presents a comprehensive analysis of GPU memory hierarchy and optimization techniques through the implementation of Matrix-Vector Multiplication (GEMV) with ReLU activation. We implement and benchmark three key components: (1) coalesced vs. non-coalesced global memory access patterns, (2) shared memory optimization techniques, and (3) CPU baseline for performance comparison. Our experimental results on a 4096×4096 matrix indicate that for this specific problem size and hardware configuration, kernel launch overhead and cache efficiency masked the theoretical benefits of coalescing and shared memory. The highly optimized CPU baseline (NumPy/BLAS) significantly outperformed the custom GPU kernels, achieving nearly 19.84 GFLOPS compared to $4.27 - 4.52$ GFLOPS on the GPU, highlighting the dominance of optimized CPU libraries for intermediate matrix sizes.

1 Task 1: Global Memory Access Patterns

1.1 Coalesced Access Kernel

The coalesced kernel implements a standard memory access pattern where consecutive threads access consecutive rows.

Results:

Kernel	**Time (ms)**	**Throughput (GFLOPS)**
Coalesced	7.424	4.52

Table 1: Coalesced Kernel Performance

1.2 Non-Coalesced Access Kernel

The non-coalesced kernel implements strided memory access (stride=4) to demonstrate memory divergence effects.

Results:

Kernel	**Time (ms)**	**Throughput (GFLOPS)**
Non-Coalesced	7.531	4.46

Table 2: Non-Coalesced Kernel Performance

1.3 Analysis

Performance Comparison:

$$\text{Performance Ratio} = \frac{4.52}{4.46} = 1.01\times \tag{1}$$

The coalesced kernel achieves similar throughput to the non-coalesced version in this specific test run. . Theoretical expectations suggest a larger gap; the observed similarity indicates that the specific GPU cache architecture masked the latency penalty, or that kernel launch overhead dominated the execution time, rendering the memory optimization negligible for this matrix size.

2 Task 2: Shared Memory Optimization

2.1 Baseline Kernel (Global Memory Only)

Kernel using only global memory for all data access (Coalesced).

Performance:

Implementation	**Time (ms)**	**GFLOPS**
Global Baseline	7.424	4.52

Table 3: Global Memory Baseline

2.2 Optimized Kernel (Shared Memory)

Kernel loading input vector into shared memory in tiles and reusing across threads.

Performance:

Implementation	**Time (ms)**	**GFLOPS**
Shared Memory	7.862	4.27

Table 4: Shared Memory Optimization

2.3 Speedup Analysis

Speedup Calculation:

$$\text{Speedup} = \frac{\text{Baseline Time}}{\text{Shared Mem Time}} = \frac{7.424}{7.862} = 0.94\times \tag{2}$$

Key Insights and Bank Conflict Analysis:

- **Performance Overhead:** The Shared Memory implementation showed slightly higher latency (0.94× speedup) compared to the global baseline. This suggests that for this specific problem size, the overhead of synchronization (`__syncthreads`) and tile loading outweighed the benefits of L1/Shared Memory caching.
- **Bank Conflicts:** The implementation effectively **avoids bank conflicts**.
  - *Load Phase:* Threads write to shared memory with a unit stride (`s_x[threadIdx.x]`). Since shared memory is organized into 32 banks (4-byte words), consecutive threads map to consecutive banks, ensuring conflict-free writing.

– *Compute Phase:* Threads read the same value (`s_x[j]`) simultaneously. This triggers the **broadcast mechanism**, where a single value is read from one bank and broadcast to all requesting threads in the warp, preventing serialization penalties.

3 Task 3: CPU Baseline

3.1 Implementation

NumPy vectorized implementation using optimized BLAS routines for matrix-vector multiplication.

3.2 Performance Across Problem Sizes

Problem Size	Time (ms)	GFLOPS
256×256	0.061	2.14
1024×1024	0.081	25.99
4096×4096	1.692	19.84
8192×8192	6.807	19.72

Table 5: CPU Baseline Performance (NumPy)

3.3 Key Findings

- CPU achieves exceptional performance: 19.84 GFLOPS for the target size (4096×4096).
- The CPU implementation is highly optimized via BLAS (AVX/SIMD instructions), efficiently handling intermediate matrix sizes.

4 GPU vs. CPU Comparison

4.1 Performance Speedup

For 4096×4096 matrix:

$$\text{GPU Speedup (vs Optimized Shared Mem)} = \frac{\text{CPU Time}}{\text{GPU Time}} = \frac{1.692 \text{ ms}}{7.862 \text{ ms}} = 0.22\times \quad (3)$$

Platform	Implementation	Time (ms)	GFLOPS	vs. CPU
CPU	NumPy	1.692	19.84	1.00×
GPU	Coalesced	7.424	4.52	0.23×
GPU	Shared Mem	7.862	4.27	0.22×

Table 6: GPU vs. CPU Performance Summary

4.2 Analysis

- In this experiment, the CPU Baseline significantly outperformed the custom CUDA kernels (4.5× faster).

- This indicates that for $N = 4096$, the overhead of data transfer and kernel launch prevented the GPU from reaching competitive throughput against an optimized CPU backend.
- To see GPU acceleration, much larger problem sizes or highly optimized libraries (cuBLAS) would be required.

5 Computational Intensity

5.1 GEMV Operations

Matrix-vector multiplication with ReLU activation:

$$y[i] = \max\left(\sum_{j=0}^{N-1} A[i,j] \cdot x[j], 0\right) \tag{4}$$

Data Movement (4096×4096):

- Matrix A: $4096 \times 4096 \times 4$ bytes = 64 MB
- Total: $\approx 64MB$ (matrix-bound)

5.2 Computational Intensity Calculation

$$I = \frac{\text{Operations}}{\text{Data Bytes}} = \frac{2 \times 4096 \times 4096}{(4096 \times 4096 + 4096) \times 4} \approx 0.5 \text{ FLOP/Byte} \tag{5}$$

Implication: GEMV is **memory-bound**.

6 Memory Access Pattern Impact

6.1 Coalescing Efficiency

Access Pattern	Address Span	Cache Lines	Transactions
Coalesced	128 B	1	1
Strided (stride=4)	512 B	4	4+

Table 7: Memory Coalescing Impact on Transactions

6.2 Observed Impact

$$\text{Throughput Ratio} = 1.01 \times \text{ (from 4.52 to 4.46 GFLOPS)} \tag{6}$$

7 Shared Memory Benefits

7.1 Actual Speedup

$$\text{Measured Speedup} = 0.94\times \tag{7}$$

7.2 Optimality Gap

$$\text{Efficiency} = \frac{\text{Actual Speedup}}{\text{Theoretical Max}} = \frac{0.94}{4096} \approx 0.02\% \tag{8}$$

8 Performance Summary

8.1 Key Metrics

Metric	Value	Unit	Status
Coalescing Ratio	1.01	×	Observed
Shared Memory Speedup	0.94	×	Observed
GPU vs. CPU Speedup	0.22	×	Observed
Computational Intensity	0.5	FLOP/Byte	Memory-Bound
CPU Throughput	19.84	GFLOPS	Baseline
GPU Coalesced	4.52	GFLOPS	0.23× vs CPU
GPU Optimized	4.27	GFLOPS	0.22× vs CPU

Table 8: Performance Metrics Summary

9 Design Principles for GPU Kernels

9.1 Principle 1: Maximize Memory Coalescing

Structure thread indexing to access consecutive memory addresses to minimize transaction overhead. **Note:** While not dominant in this small-scale test, coalescing remains a critical best practice for larger workloads.

9.2 Principle 2: Exploit Data Locality

Use shared memory to cache frequently reused data. **Note:** Overhead must be amortized over sufficient computation; otherwise, synchronization costs may degrade performance as observed (0.94×).

9.3 Principle 3: Minimize Overhead

Ensure that the computational work per thread is sufficient to hide the latency of synchronization barriers and kernel launches.

10 Conclusion

10.1 Key Findings

1. **CPU Baseline**: Achieved highest performance (19.84 GFLOPS) for this problem size.
2. **GPU Kernels**: The custom GPU implementations (4.27 - 4.52 GFLOPS) were hindered by overhead and did not outperform the optimized CPU BLAS implementation.
3. **Optimization**: Shared memory did not provide a speedup in this specific instance (0.94×), suggesting that synchronization costs outweighed caching benefits or that the problem size was too small to saturate the GPU.

10.2 Implications

Future work should focus on increasing the problem size (N), optimizing the kernel launch parameters, or utilizing CUDA streams to hide latency. The results highlight that GPU acceleration is not automatic and requires careful tuning relative to the overheads.