Performance Analysis of GPU Optimization Strategies for Histogram Computation and Matrix Multiplication

Jaka Škerjanc, Erik Pahor

I. Introduction

This report presents an analysis of two GPU performance optimization tasks: histogram computation and matrix multiplication using TensorCores. The homework focuses on understanding the impact of different implementation strategies on GPU performance metrics and execution efficiency. The histogram computation task compares naive and privatized implementations, while the matrix multiplication task evaluates the benefits of TensorCore acceleration against traditional GPU computation methods.

II. Performance Metrics Explained

To understand performance differences, we analyze key metrics obtained using the GEM5 simulator:

- **Mean Load Latency (cycles)** – Average number of cycles needed to load data. Higher latency often indicates memory contention.
- **Vector ALU Instructions** – Number of arithmetic vector instructions executed. Indicates compute workload.
- **Shared Memory Reads/Writes** – Accesses to shared memory, useful for measuring local data sharing efficiency.
- **Bank Accesses** – Number of accesses to shared memory banks. Excessive access can cause bank conflicts and slowdowns.
- **Total Cycles** – Total execution time of the kernel in GPU clock cycles. Lower values are better.
- **Vector Per Cycle (VPC)** – Number of vector instructions executed per cycle. Indicates GPU utilization efficiency.

These metrics provide insights into memory bottlenecks, compute intensity, and overall kernel efficiency.

III. Implementation Approach

Task 1: Histogram Computation

The implementation approach for the histogram computation involved two distinct strategies:

Naive Implementation

- Direct updates to global memory
- Single histogram shared across all threads
- No privatization strategy

Optimized Implementation

- Thread group privatization
- Local histogram copies per thread group
- Final merge step to combine private histograms

Task 2: Matrix Multiplication

The matrix multiplication implementation utilized two approaches:

Naive Implementation (mm_naive)

- Traditional GPU matrix multiplication
- Basic block-based approach
- Standard CUDA kernel implementation

TensorCore Implementation (mm_block_tc)

- WMMA PTX API utilization
- Block-based matrix multiplication
- TensorCore acceleration
- Configurable block sizes matching warp dimensions

IV. Performance Results

A. Histogram Computation

The performance analysis was conducted with varying compute units (2, 4, and 8) and focused on the following metrics:

Table I
Histogram Performance Metrics

Metric	Naive			Optimized		
	8CU	4CU	2CU	8CU	4CU	2CU
Load Latency (cycles)	24.2M	12.4M	6.2M	4.1M	1.6M	0.8M
Vector ALU Instructions	5.6K	11.1K	22.5K	7.2K	14.3K	28.7K
Shared Memory Reads	0	0	0	512	1.0K	2.0K
Shared Memory Writes	0	0	0	512	1.0K	2.0K
Bank Accesses	0	0	0	98.3K	196.6K	393.2K
Total Cycles	791.8K	802.9K	827.2K	174.5K	176.2K	227.6K
Vector Per Cycle	0.993	1.928	3.803	7.136	14.131	21.887

B. Matrix Multiplication

Performance comparison was conducted across different matrix sizes:

Table II
Matrix Multiplication Performance Comparison

Matrix Size	TensorCore (ms)	Without TensorCore (ms)	Speedup
256×256	0.016	0.224	14.38×
512×512	0.040	1.543	38.38×
1024×1024	0.327	11.724	35.88×
2048×2048	1.884	80.922	42.96×
4096×4096	15.329	881.823	57.53×

V. Analysis

Task 1: Histogram Computation

Memory Contention and Latency: The naive implementation has significantly higher load latency, caused by multiple threads accessing the same global memory address concurrently. The optimized version, through privatization, reduces this contention, which is reflected in a drop from 24.2M to 4.1M cycles for 8 compute units.

Shared Memory Utilization: In the naive implementation, shared memory is unused. The optimized version introduces local histograms in shared memory, visible through increased reads, writes, and bank accesses. While this introduces more shared memory traffic, it is still far faster than using global memory due to lower access latency.

Vector Per Cycle and Total Cycles: The VPC metric shows significant improvement in the optimized version, indi-

cating better GPU utilization. Correspondingly, total execution time is greatly reduced—up to 4.5× faster in the 8CU setup.

Task 2: Matrix Multiplication

Scaling with Matrix Size: The performance benefit of TensorCores increases with matrix size. This is due to the hardware acceleration features becoming more effective with larger data blocks.

Execution Time Reduction: TensorCore-based kernels show 14× to 57× speedup over the naive CUDA version. This is due to the parallelism and matrix math capabilities embedded in TensorCore hardware.

Block Size Matching and Utilization: Configuring the block sizes to match warp dimensions (typically 32 threads) ensures maximal TensorCore utilization and minimizes idle cycles.

VI. Conclusion

The study demonstrates several key findings:

- **Memory Access Optimization:**
  - Proper memory access patterns (privatization and shared memory usage) significantly reduce contention and latency.
- **Hardware Utilization:**
  - TensorCore acceleration dramatically improves performance, especially for large matrix workloads.

Overall, GPU performance is highly sensitive to how memory and hardware resources are managed. By reducing contention and leveraging specialized hardware, significant speedups can be achieved.