HPML Lab 4 Report

# CUDA Programming

Garima Mahajan

NetID: gm3228

## Introduction

This lab has three parts. Part-A on vector addition and matrix multiplication in CUDA, Part-B on CUDA Unified Memory, and Part-C on convolution in CUDA

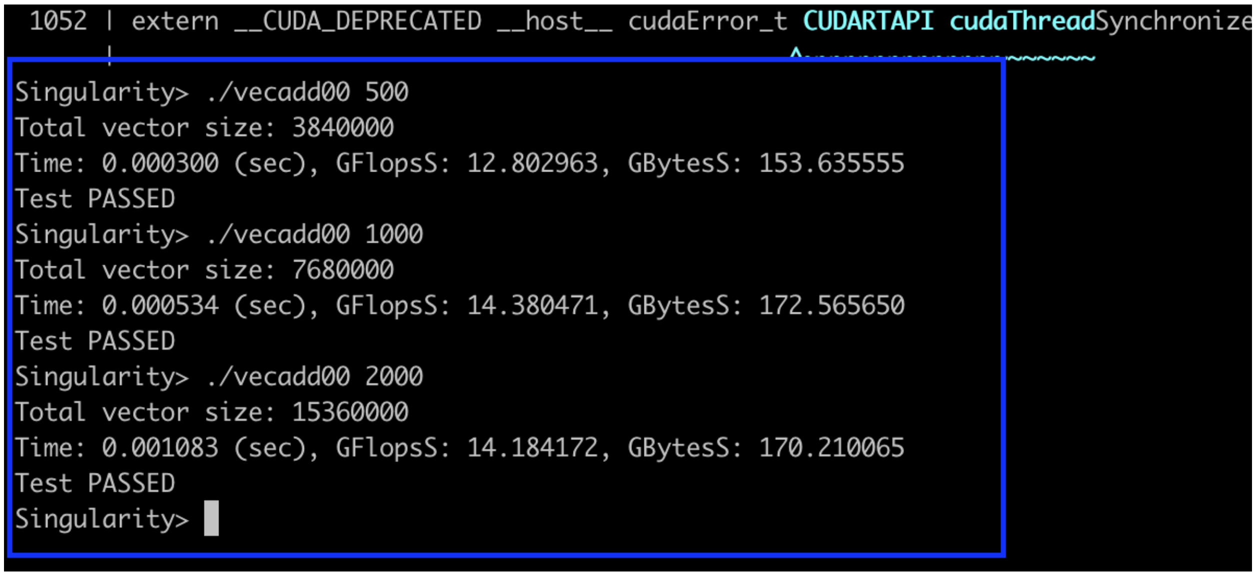
## Part A: CUDA Matrix Operations

Purpose: The purpose of this exercise is for you to learn how to write programs using the CUDA programming interface, how to run such programs using an NVIDIA graphics processor, and how to think about the factors that govern the performance of programs running within the CUDA environment.

Relevant Folders:

```
gm3228/
├── Makefile                    # Build instructions for compiling the code
├── PartA
│   ├── matmult.cu              # host code for Q3
│   ├── matmult01.cu            # host code for Q4
│   ├── matmultKernel.h
│   ├── matmultKernel00.cu      # kernel (device) code for Q3
│   ├── matmultKernel01.cu      # kernel (device) code for Q4
│   ├── timer.cu
│   ├── timer.h
│   ├── vecadd.cu               # host code for Q1
│   ├── vecadd01.cu             # host code for Q2
│   ├── vecaddKernel.h
│   ├── vecaddKernel00.cu       # kernel (device) code for Q1
│   └── vecaddKernel01.cu       # kernel (device) code for Q2
├── PartB # not relevant for this part
```

## Q1.



```
1052 | extern __CUDA_DEPRECATED __host__ cudaError_t CUDARTAPI cudaThreadSynchronize
     |                                                 ^~~~~~~~~~~~~~~~~~~~~~~~~
Singularity> ./vecadd00 500
Total vector size: 3840000
Time: 0.000300 (sec), GFlopsS: 12.802963, GBytesS: 153.635555
Test PASSED
Singularity> ./vecadd00 1000
Total vector size: 7680000
Time: 0.000534 (sec), GFlopsS: 14.380471, GBytesS: 172.565650
Test PASSED
Singularity> ./vecadd00 2000
Total vector size: 15360000
Time: 0.001083 (sec), GFlopsS: 14.184172, GBytesS: 170.210065
Test PASSED
Singularity> 
```

Analysis:

1. Scaling Pattern:

- Vector sizes increase by 2x each time (3.84M → 7.68M → 15.36M elements).
- The execution time roughly doubles with each size increase (0.3ms → 0.534ms → 1.083ms), showing near-linear scaling.

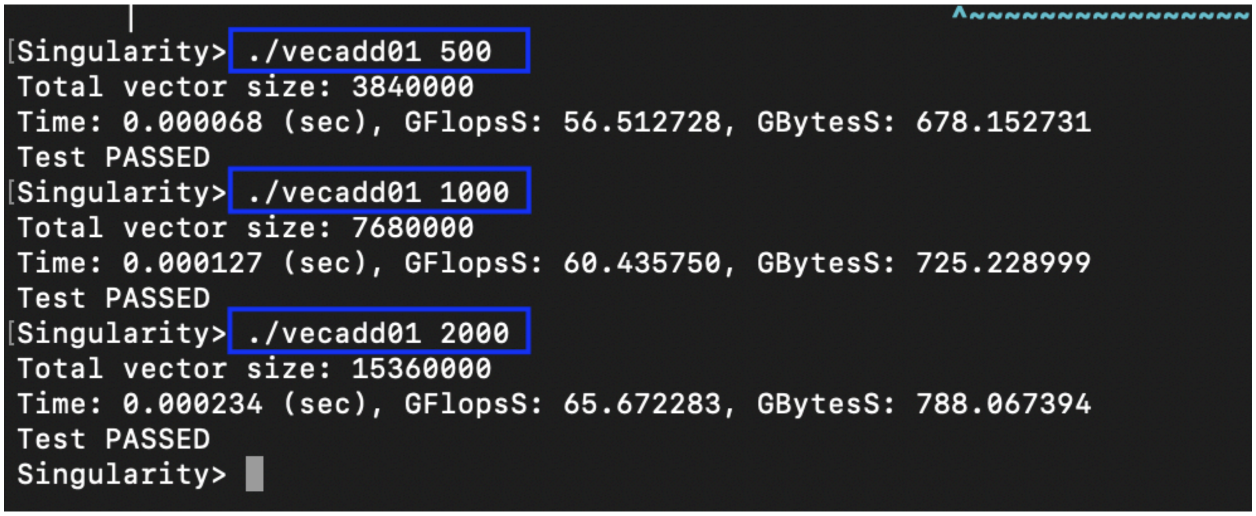
2. Performance Metrics:

- FLOPS performance peaks at ~14.38 GFLOPS with the 1000 size case.
- Memory bandwidth peaks around 172 GB/s in the middle case.
- There's a slight performance degradation at the largest size (drops to 14.18 GFLOPS).

3. Memory Access Pattern:

- The relatively constant GFLOPS across sizes suggests the operation is memory-bound rather than compute-bound.
- Without memory coalescing, the memory access pattern is likely suboptimal, explaining why the achieved bandwidth is lower than what modern GPUs are capable of.

## Q2.



```
[Singularity> ./vecadd01 500
Total vector size: 3840000
Time: 0.000068 (sec), GFlopsS: 56.512728, GBytesS: 678.152731
Test PASSED
[Singularity> ./vecadd01 1000
Total vector size: 7680000
Time: 0.000127 (sec), GFlopsS: 60.435750, GBytesS: 725.228999
Test PASSED
[Singularity> ./vecadd01 2000
Total vector size: 15360000
Time: 0.000234 (sec), GFlopsS: 65.672283, GBytesS: 788.067394
Test PASSED
Singularity>
```

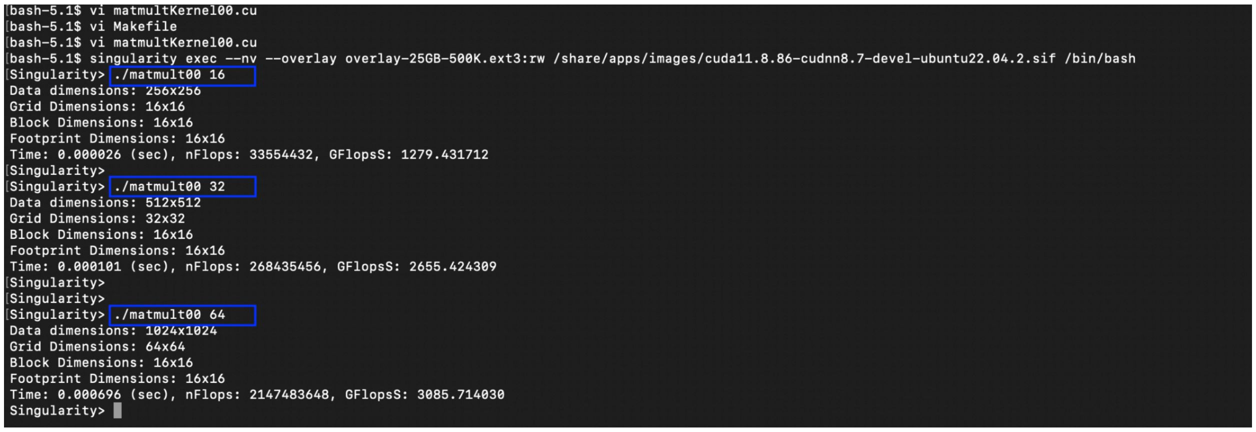
Analysis:

1. Performance Improvement:
    - Dramatic speedup compared to non-coalesced version (4-5x faster)
    - Example: For size 2000, time improved from 1.083ms to 0.234ms
    - Reason: Coalesced memory access allows multiple threads to fetch/store data in a single transaction, maximizing memory bandwidth utilization

2. Scaling Characteristics:
    - Vector sizes: 3.84M → 7.68M → 15.36M elements
    - Performance actually improves with larger sizes (56.5 → 60.4 → 65.7 GFLOPS)
    - Reason: Larger sizes allow better amortization of overhead and more efficient memory access patterns

3. Memory Bandwidth:
    - Much higher bandwidth utilization (678 → 725 → 788 GB/s).
    - Consistent improvement with size increases.
    - Reason: Coalesced access enables the GPU to combine multiple memory requests into fewer, larger transactions, better utilizing the memory bus.

## Q3.



```
[bash-5.1$ vi matmultKernel00.cu
[bash-5.1$ vi Makefile
[bash-5.1$ vi matmultKernel00.cu
[bash-5.1$ singularity exec --nv --overlay overlay-25GB-500K.ext3:rw /share/apps/images/cuda11.8.86-cudnn8.7-devel-ubuntu22.04.2.sif /bin/bash
[Singularity> ./matmult00 16
Data dimensions: 256x256
Grid Dimensions: 16x16
Block Dimensions: 16x16
Footprint Dimensions: 16x16
Time: 0.000026 (sec), nFlops: 33554432, GFlopsS: 1279.431712
[Singularity>
[Singularity> ./matmult00 32
Data dimensions: 512x512
Grid Dimensions: 32x32
Block Dimensions: 16x16
Footprint Dimensions: 16x16
Time: 0.000101 (sec), nFlops: 268435456, GFlopsS: 2655.424309
[Singularity>
[Singularity>
[Singularity> ./matmult00 64
Data dimensions: 1024x1024
Grid Dimensions: 64x64
Block Dimensions: 16x16
Footprint Dimensions: 16x16
Time: 0.000696 (sec), nFlops: 2147483648, GFlopsS: 3085.714030
Singularity>
```

Analysis:

1. Problem Scaling:

- Matrix sizes: 256x256 → 512x512 → 1024x1024.
- Number of operations (nFlops) increases by 8x each step (33.5M → 268.4M → 2.14B).
- Time increases: 0.026ms → 0.101ms → 0.696ms.
- Reason: O($n^3$) complexity of matrix multiplication is evident.

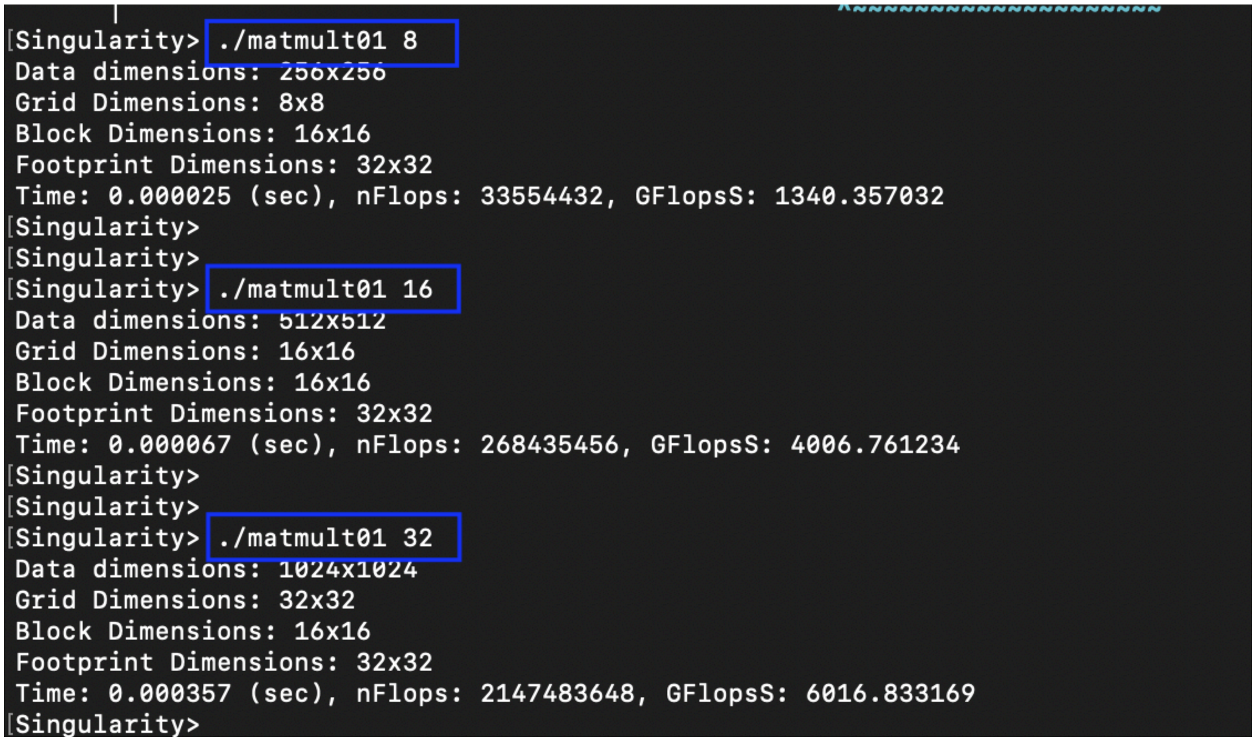
2. Performance Characteristics:

- FLOPS gradually improves: 1.28 → 2.66 → 3.09 TFLOPS.
- Reason: Better utilization of GPU resources with larger matrices despite unoptimized memory access.
- However, performance is still suboptimal due to: Lack of memory coalescing causing scattered memory access  and this approach not leveraging data locality.

3. Grid and Block Configuration:

- Fixed block size (16x16) with scaling grid dimensions.
- Grid sizes scale with matrix size: 16x16 → 32x32 → 64x64.
- Footprint remains constant at 16x16.
- Reason: Each thread handles single element multiplication, leading to many uncoalesced memory accesses.

## Q4.



```
Singularity> ./matmult01 8
Data dimensions: 256x256
Grid Dimensions: 8x8
Block Dimensions: 16x16
Footprint Dimensions: 32x32
Time: 0.000025 (sec), nFlops: 33554432, GFlopsS: 1340.357032
Singularity>
Singularity>
Singularity> ./matmult01 16
Data dimensions: 512x512
Grid Dimensions: 16x16
Block Dimensions: 16x16
Footprint Dimensions: 32x32
Time: 0.000067 (sec), nFlops: 268435456, GFlopsS: 4006.761234
Singularity>
Singularity>
Singularity> ./matmult01 32
Data dimensions: 1024x1024
Grid Dimensions: 32x32
Block Dimensions: 16x16
Footprint Dimensions: 32x32
Time: 0.000357 (sec), nFlops: 2147483648, GFlopsS: 6016.833169
Singularity>
```

Analysis:

1. Performance Improvements:

- For 256x256: 1.34 TFLOPS vs 1.28 TFLOPS (1.05x speedup)
- For 512x512: 4.01 TFLOPS vs 2.66 TFLOPS (1.51x speedup)
- For 1024x1024: 6.02 TFLOPS vs 3.09 TFLOPS (1.95x speedup)
- Reason: Speedup increases with matrix size due to better amortization of optimization benefits

2. Key Optimizations:

- Increased Footprint: 32x32 vs 16x16 in original. Each thread now computes 4 values instead of 1. Better computational density per thread reduces overall thread management overhead Memory Coalescing.
- Improved memory access patterns. Consecutive threads access consecutive memory locations, enabling efficient memory transactions.

- Loop Unrolling. Eliminates loop overhead. Better instruction-level parallelism and reduced branch predictions.

3. Grid Configuration Changes:

- Grid dimensions reduced by half: (8x8 ➔ 16x16 ➔ 32x32) vs original (16x16 ➔ 32x32 ➔ 64x64).
- Each thread doing 4x more work means fewer total threads needed.
- Block dimensions maintained at 16x16 for optimal occupancy.

## Q5.

Some rules of thumb for achieving good performance with CUDA, based on the experiments:

1. Memory Access Optimization:

- strive for coalesced memory access patterns
- Reason: As seen in both vector addition and matrix multiplication experiments, coalesced access provided 4-5x speedup in vector addition and up to 2x in matrix multiplication
- Rule: Ensure consecutive threads access consecutive memory locations

2. Thread Workload Balancing:

- Optimize work per thread (e.g., computing multiple elements per thread)
- Evidence: Matrix multiplication showed better performance when each thread computed 4 elements instead of 1
- Rule: Find sweet spot between too little work (overhead dominates) and too much work (reduced parallelism)

3. Scale Considerations:

- Benefits of optimizations often increase with problem size
- Example: Matrix multiplication speedup grew from 1.05x to 1.95x as size increased
- Rule: Design optimizations with scalability in mind

4. Memory Access Patterns:

- Minimize global memory access by maximizing data reuse
- Use shared memory when possible for frequently accessed data
- Rule: Think in terms of memory transactions, not just computations

5. Code Optimization:

- Unroll loops to reduce overhead and improve instruction-level parallelism
- Keep block dimensions optimal for GPU architecture (typically multiples of 32)
- Rule: Consider both algorithmic and hardware-level optimizations

## Part B: CUDA Unified Memory

Purpose: In this problem we will compare vector operations executed on host vs on GPU to quantify the speed-up.

Relevant Folders:

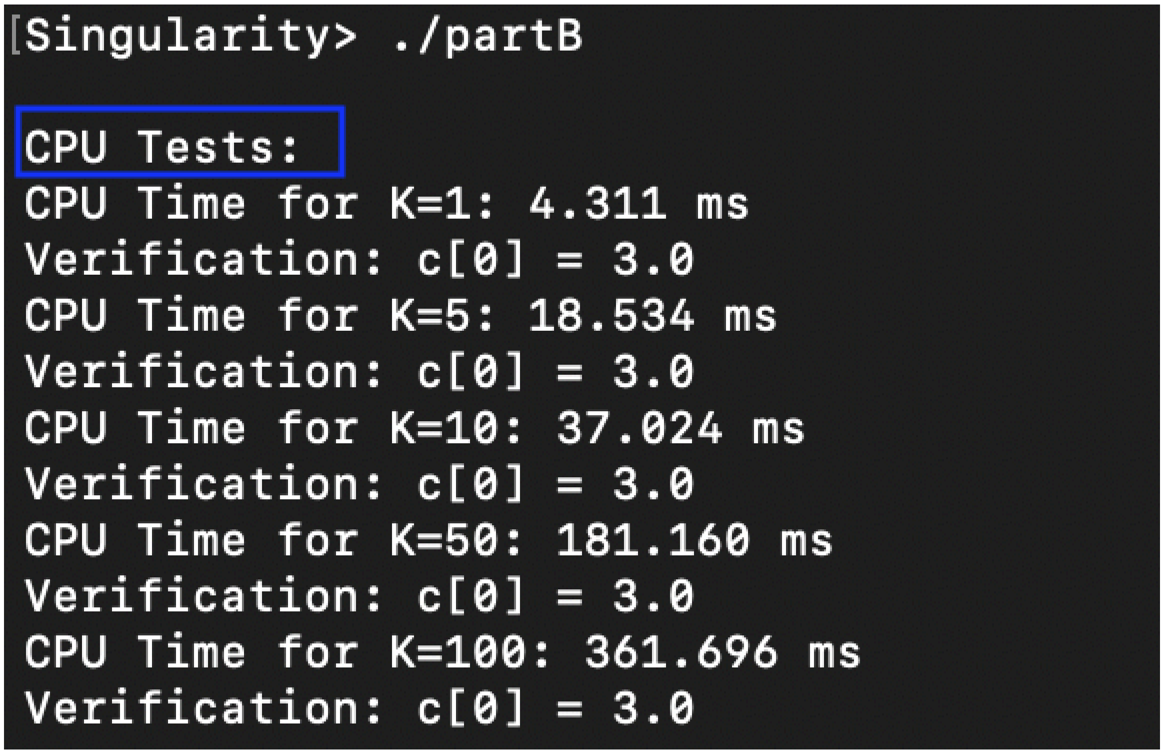
```
gm3228/
├── Makefile                    # Build instructions for compiling the code
├── PartA                       # Not relevant
├── PartB
      ├── partB.cu
```

Note: For Part B, due to time constraints, I combined the implementations of Q1, Q2, and Q3 into a single file named “partB.cu” instead of segregating them into separate files. I tested all four parts using one main function to streamline the process.

Command to run all three in one:

```
./partB
```

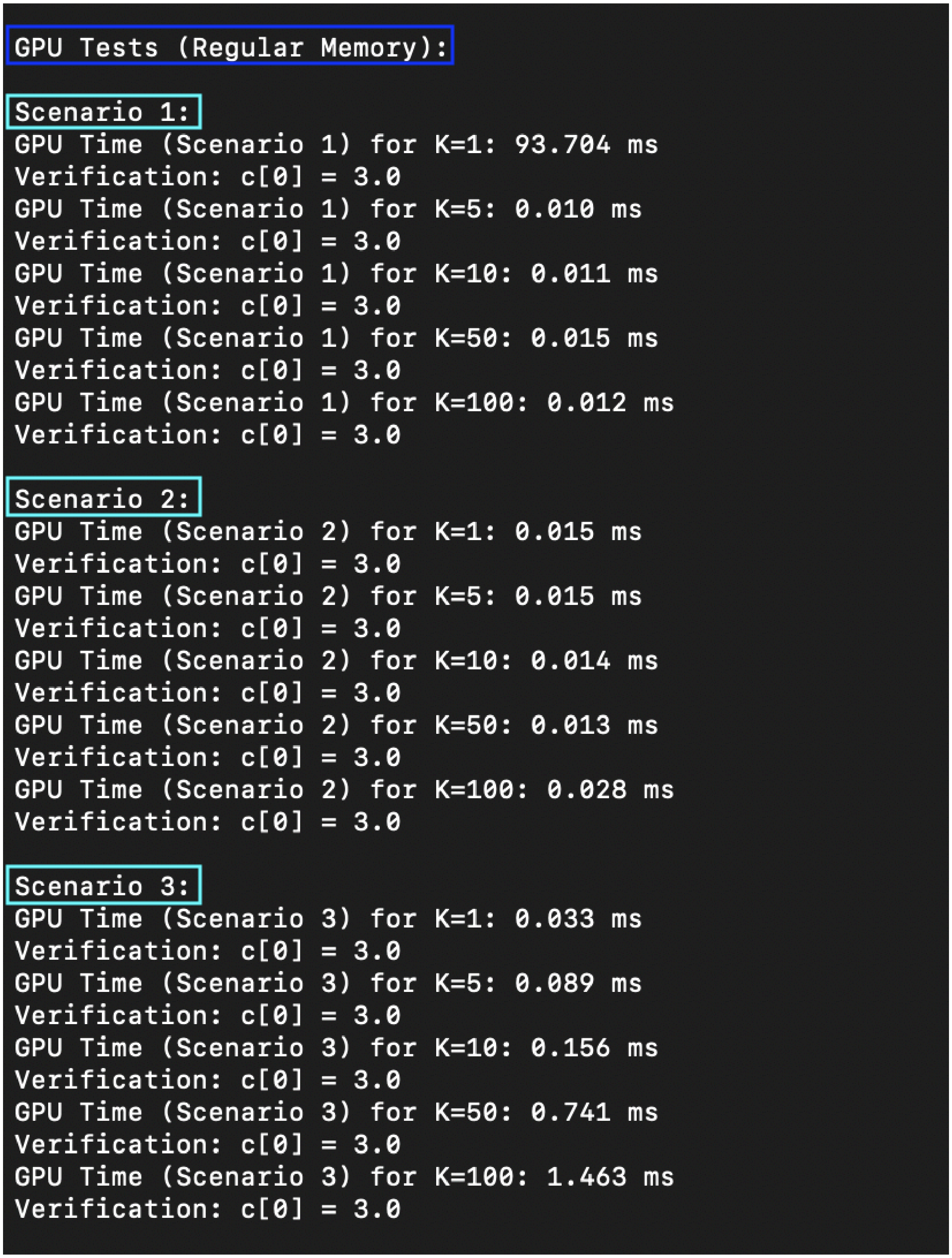
## Q1.



```
[Singularity> ./partB

CPU Tests:
CPU Time for K=1: 4.311 ms
Verification: c[0] = 3.0
CPU Time for K=5: 18.534 ms
Verification: c[0] = 3.0
CPU Time for K=10: 37.024 ms
Verification: c[0] = 3.0
CPU Time for K=50: 181.160 ms
Verification: c[0] = 3.0
CPU Time for K=100: 361.696 ms
Verification: c[0] = 3.0
```

## Q2.



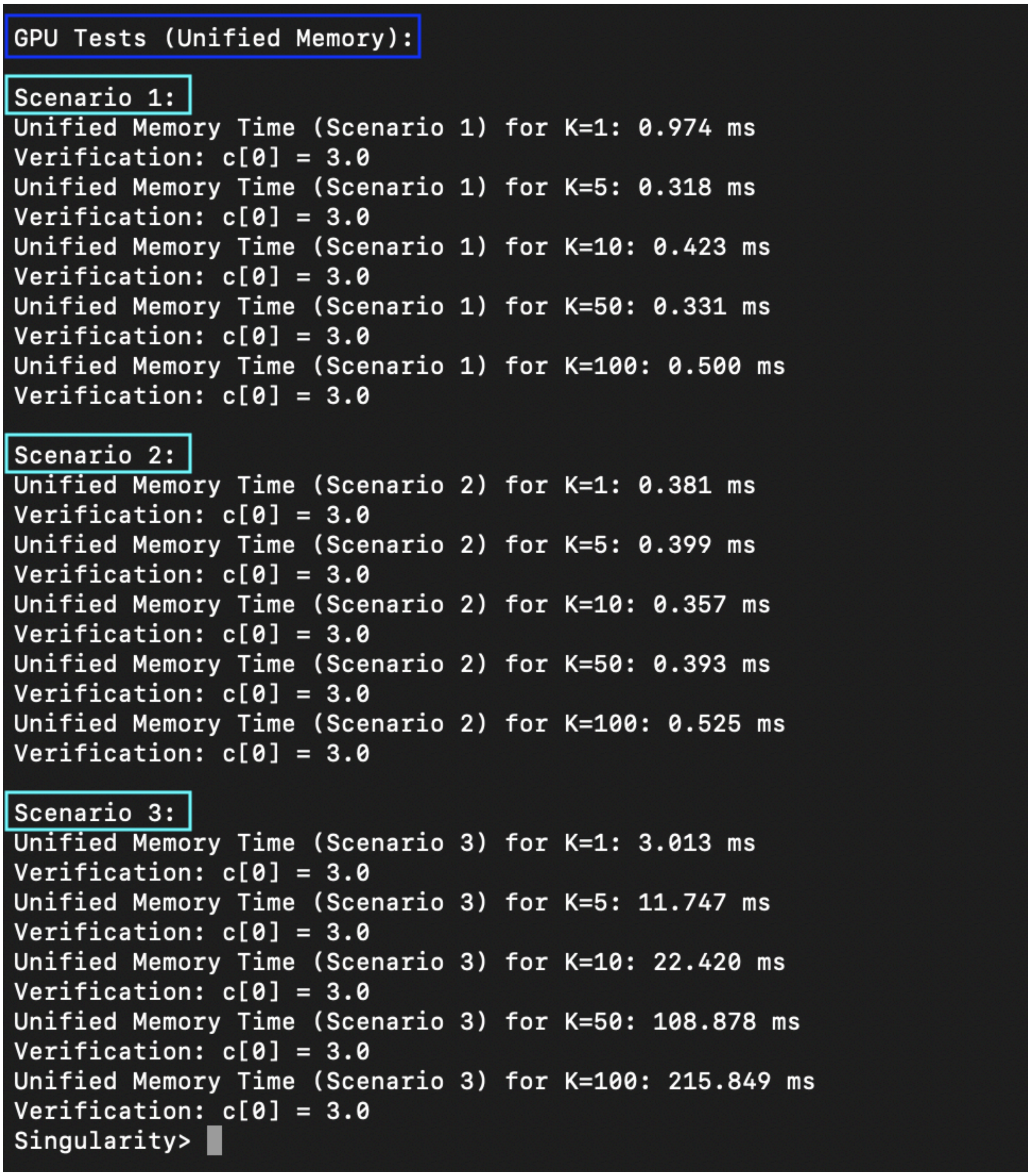
```
GPU Tests (Regular Memory):

Scenario 1:
GPU Time (Scenario 1) for K=1: 93.704 ms
Verification: c[0] = 3.0
GPU Time (Scenario 1) for K=5: 0.010 ms
Verification: c[0] = 3.0
GPU Time (Scenario 1) for K=10: 0.011 ms
Verification: c[0] = 3.0
GPU Time (Scenario 1) for K=50: 0.015 ms
Verification: c[0] = 3.0
GPU Time (Scenario 1) for K=100: 0.012 ms
Verification: c[0] = 3.0

Scenario 2:
GPU Time (Scenario 2) for K=1: 0.015 ms
Verification: c[0] = 3.0
GPU Time (Scenario 2) for K=5: 0.015 ms
Verification: c[0] = 3.0
GPU Time (Scenario 2) for K=10: 0.014 ms
Verification: c[0] = 3.0
GPU Time (Scenario 2) for K=50: 0.013 ms
Verification: c[0] = 3.0
GPU Time (Scenario 2) for K=100: 0.028 ms
Verification: c[0] = 3.0

Scenario 3:
GPU Time (Scenario 3) for K=1: 0.033 ms
Verification: c[0] = 3.0
GPU Time (Scenario 3) for K=5: 0.089 ms
Verification: c[0] = 3.0
GPU Time (Scenario 3) for K=10: 0.156 ms
Verification: c[0] = 3.0
GPU Time (Scenario 3) for K=50: 0.741 ms
Verification: c[0] = 3.0
GPU Time (Scenario 3) for K=100: 1.463 ms
Verification: c[0] = 3.0
```

## Q3.



```
GPU Tests (Unified Memory):

Scenario 1:
Unified Memory Time (Scenario 1) for K=1: 0.974 ms
Verification: c[0] = 3.0
Unified Memory Time (Scenario 1) for K=5: 0.318 ms
Verification: c[0] = 3.0
Unified Memory Time (Scenario 1) for K=10: 0.423 ms
Verification: c[0] = 3.0
Unified Memory Time (Scenario 1) for K=50: 0.331 ms
Verification: c[0] = 3.0
Unified Memory Time (Scenario 1) for K=100: 0.500 ms
Verification: c[0] = 3.0

Scenario 2:
Unified Memory Time (Scenario 2) for K=1: 0.381 ms
Verification: c[0] = 3.0
Unified Memory Time (Scenario 2) for K=5: 0.399 ms
Verification: c[0] = 3.0
Unified Memory Time (Scenario 2) for K=10: 0.357 ms
Verification: c[0] = 3.0
Unified Memory Time (Scenario 2) for K=50: 0.393 ms
Verification: c[0] = 3.0
Unified Memory Time (Scenario 2) for K=100: 0.525 ms
Verification: c[0] = 3.0

Scenario 3:
Unified Memory Time (Scenario 3) for K=1: 3.013 ms
Verification: c[0] = 3.0
Unified Memory Time (Scenario 3) for K=5: 11.747 ms
Verification: c[0] = 3.0
Unified Memory Time (Scenario 3) for K=10: 22.420 ms
Verification: c[0] = 3.0
Unified Memory Time (Scenario 3) for K=50: 108.878 ms
Verification: c[0] = 3.0
Unified Memory Time (Scenario 3) for K=100: 215.849 ms
Verification: c[0] = 3.0
Singularity>
```

Analysis:

1. CPU Performance:

- Shows linear scaling: 4.3ms (1M) → 361.7ms (100M)
- Reason: Sequential processing with direct memory access, scaling proportionally with data size
- No parallelization benefits, hence linear increase in time with array size

2. Regular GPU Memory:

a) Scenario 1 (1 block, 1 thread):

- Anomaly at K=1 (93.7ms) due to initial GPU setup overhead
- Extremely fast for larger K (0.010-0.015ms)
- Reason: After initial setup, benefits from GPU's efficient memory handling despite single thread

b) Scenario 2 (1 block, 256 threads):

- Consistent performance (~0.015ms) across sizes
- Slight increase at K=100 (0.028ms)
- Reason: Better parallelization than Scenario 1, but limited by single block
- 256 threads can efficiently handle the workload in parallel within one block

c) Scenario 3 (multiple blocks, 256 threads/block):

- Scales with size: 0.033ms (1M) → 1.463ms (100M)
- Still significantly faster than CPU
- Reason: Best parallelization strategy, utilizing full GPU capacity
- Scales better because workload is distributed across multiple blocks

3. Unified Memory:

a) Scenario 1 & 2:

- Similar performance patterns (0.3-0.5ms)
- More consistent than regular memory

- Reason: Overhead of automatic memory management, but simplified programming model

b) Scenario 3:

- Performance closer to CPU: 3.013ms (1M) → 215.849ms (100M)
- Reason: Overhead of page faults and memory transfers in unified memory system
- Multiple blocks cause more memory management overhead

Summary:

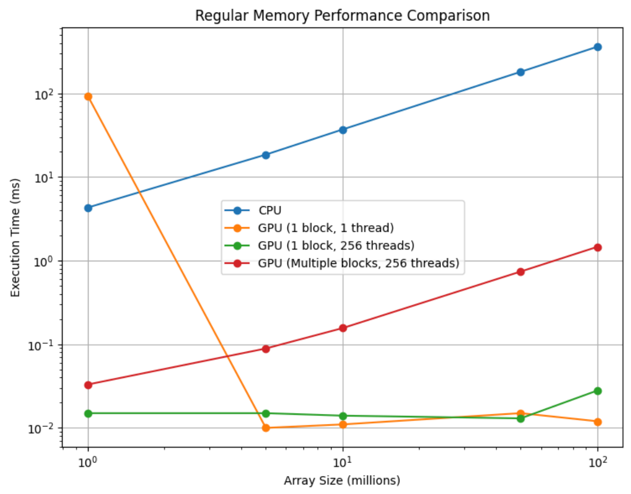
1. Regular GPU memory shows best performance but requires explicit memory management.

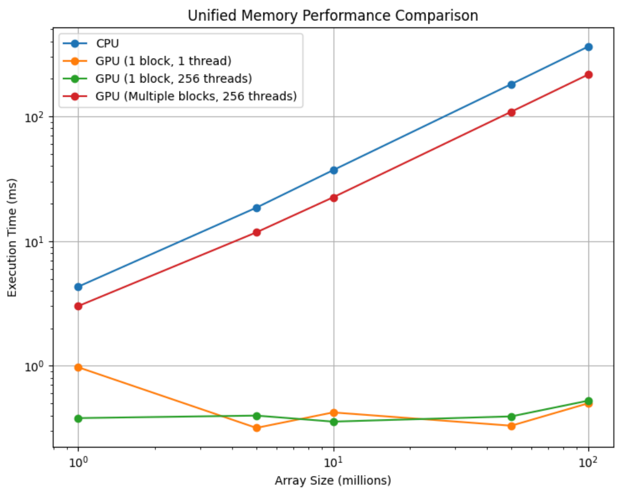
2. Unified memory sacrifices some performance for programming convenience.

3. Multiple blocks with regular memory (Scenario 3) provides the best balance of scalability and performance.

4. Initial GPU setup overhead is significant for small datasets but becomes negligible for larger ones.

## Q4.





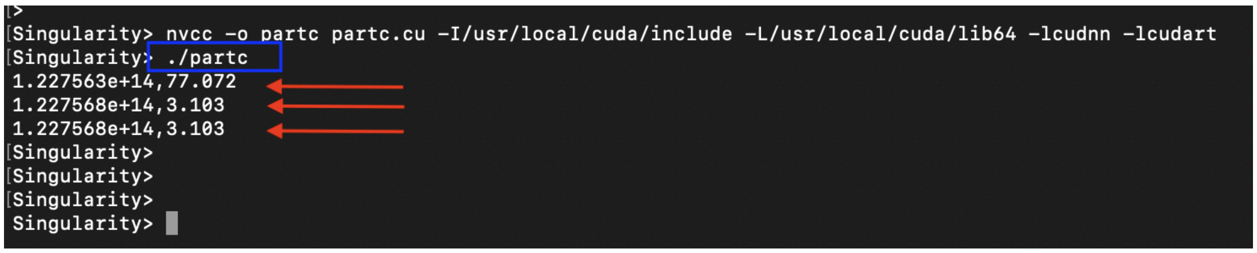
## Part C: Convolution in CUDA

Purpose: In this part of the lab we implement a convolution of an image using a set of filters.

Relevant Folders:

```
gm3228/
├── Makefile                        # Build instructions for compiling the code
├── PartA                           # Not relevant
├── PartB                           # Not relevant
└── PartC
       ├──partC.cu
```

Results:



```
[>
[Singularity> nvcc -o partc partc.cu -I/usr/local/cuda/include -L/usr/local/cuda/lib64 -lcudnn -lcudart
[Singularity> ./partc
1.227563e+14,77.072
1.227568e+14,3.103
1.227568e+14,3.103
[Singularity>
[Singularity>
[Singularity>
Singularity>
```

Analysis:

1. Basic Convolution (C1):

- Checksum: 1.227563e+14
- Time: 77.072ms
- Performance: Slowest of all three
- Reason: Uses naive approach with no optimizations, leading to: High global memory access overhead; No memory coalescing; No data reuse between threads

2. Tiled Convolution (C2):

- Checksum: 1.227568e+14
- Time: 3.103ms
- Performance: ~25x faster than basic version
- Reason for improvement: Uses shared memory to reduce global memory accesses; Better data locality through tiling; Enables data reuse between threads in the same block

3. cuDNN Convolution (C3):

- Checksum: 1.227568e+14
- Time: 3.103ms
- Performance: Matches tiled version
- Reason: Highly optimized library implementation; Likely using similar tiling strategies plus additional optimizations; Automatically selects the best algorithm for the given hardware

Key Observations:

1. Checksums:

- C2 and C3 produce identical results (1.227568e+14).
- C1 shows a slight difference (1.227563e+14), possibly due to different order of floating-point operations affecting numerical precision.

2. Performance:

- Both optimized versions (C2 and C3) achieve identical performance.
- This suggests our tiled implementation (C2) is well-optimized, reaching performance levels comparable to professional library implementation.
- The basic version (C1) demonstrates why optimization is crucial for convolution operations.

Sources / References:

https://docs.nvidia.com/cuda/cuda-c-programming-guide/

https://developer.nvidia.com/blog/even-easier-introduction-cuda/

https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/

https://developer.nvidia.com/blog/unified-memory-cuda-beginners/

https://docs.nvidia.com/cudnn/index.html

https://nichijou.co/cuda7-tiling/

https://github.com/vzhou842/cnn-from-scratch

Udacity Youtube playlist for Intro to Parallel Programming

Lecture slides on Brightspace (High Performance Machine Learning)