Contents:

Machine specs:

- Google cloud with Nvidia T4 GPU as per the instructions in the assignment document appendix.

Part A

1.

- 500:
  - Time: 0.001016 (sec)
  - GFlopsS: 3.779894
  - GBytesS: 45.358725

- 1000
  - Time: 0.002228 (sec)
  - GFlopsS: 3.447004
  - GBytesS: 41.364051

- 2000
  - Time: 0.003757 (sec)
  - GFlopsS: 4.088368
  - GBytesS: 49.060421

```
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$ ./vecadd00 500
Total vector size: 3840000
Time: 0.001016 (sec), GFlopsS: 3.779894, GBytesS: 45.358725
Test PASSED
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$ ./vecadd00 1000
Total vector size: 7680000
Time: 0.002228 (sec), GFlopsS: 3.447004, GBytesS: 41.364051
Test PASSED
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$ ./vecadd00 2000
Total vector size: 15360000
Time: 0.003757 (sec), GFlopsS: 4.088368, GBytesS: 49.060421
Test PASSED
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$
```

For this part, we are performing baseline vector addition with non-coalesced memory access. Since each thread processes a long contiguous slice, we get poorly mapped accesses from the GPU's memory subsystem.

2.

- 500
  - Time: 0.000385 (sec)
  - GFlopsS: 9.972834
  - GBytesS: 119.674011
- 1000
  - Time: 0.000761 (sec)
  - GFlopsS: 10.091558
  - GBytesS: 121.098702

- 2000

- Time: 0.001596 (sec)
- GFlopsS: 9.624217
- GBytesS: 115.490606

In part 2 here we use coalesced memory access instead. Since vector addition is purely memory-bound, any improvement in access regularity translates directly to greater throughput, which is what we see in comparison from part 1 where we see a large increase on throughput.

```
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$ ./vecadd01 500
Total vector size: 3840000
Time: 0.000385 (sec), GFlopsS: 9.972834, GBytesS: 119.674011
Test PASSED
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$ ./vecadd01 1000
Total vector size: 7680000
Time: 0.000761 (sec), GFlopsS: 10.091558, GBytesS: 121.098702
Test PASSED
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$ ./vecadd01 2000
Total vector size: 15360000
Time: 0.001596 (sec), GFlopsS: 9.624217, GBytesS: 115.490606
Test PASSED
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$
```

3.

- 256
    - Time: 0.172199 (sec)
    - nFlops: 137438953472
    - GFlopsS: 798.140203

- 512
    - Time: 1.476909 (sec)
    - nFlops: 1099511627776
    - GFlopsS: 744.468013

- 1024
    - Time: 12.423683 (sec)
    - nFlops: 8796093022208
    - GFlopsS: 708.010102

```
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$ ./matmult00 256
Data dimensions: 4096x4096
Grid Dimensions: 256x256
Block Dimensions: 16x16
Footprint Dimensions: 16x16
Time: 0.172199 (sec), nFlops: 137438953472, GFlopsS: 798.140203
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$ ./matmult00 512
Data dimensions: 8192x8192
Grid Dimensions: 512x512
Block Dimensions: 16x16
Footprint Dimensions: 16x16
Time: 1.476909 (sec), nFlops: 1099511627776, GFlopsS: 744.468013


TEST FAILED: number of errors:  4321886, max rel error: 0.000122
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$ ./matmult00 1024
Data dimensions: 16384x16384
Grid Dimensions: 1024x1024
Block Dimensions: 16x16
Footprint Dimensions: 16x16
Time: 12.423683 (sec), nFlops: 8796093022208, GFlopsS: 708.010102


TEST FAILED: number of errors:  138340737, max rel error: 0.000244
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$
```

4

- Footprint size: 32
  - 256
    - Time: 0.726687 (sec)
    - nFlops: 1099511627776
    - GFlopsS: 1513.047153
  - 512
    - Time: 6.010936 (sec)
    - nFlops: 8796093022208
    - GFlopsS: 1463.348302
  - 1024
    - Time: 60.007587 (sec)
    - nFlops: 70368744177664
    - GFlopsS: 1172.664120

```
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$ ./matmult01 256
Data dimensions: 8192x8192
Grid Dimensions: 256x256
Block Dimensions: 16x16
Footprint Dimensions: 32x32
Time: 0.726687 (sec), nFlops: 1099511627776, GFlopsS: 1513.047153


TEST FAILED: number of errors:  4321886, max rel error: 0.000122
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$ ./matmult01 512
Data dimensions: 16384x16384
Grid Dimensions: 512x512
Block Dimensions: 16x16
Footprint Dimensions: 32x32
Time: 6.010936 (sec), nFlops: 8796093022208, GFlopsS: 1463.348302


TEST FAILED: number of errors:  138340737, max rel error: 0.000244
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$ ./matmult01 1024
Data dimensions: 32768x32768
Grid Dimensions: 1024x1024
Block Dimensions: 16x16
Footprint Dimensions: 32x32
Time: 60.007587 (sec), nFlops: 70368744177664, GFlopsS: 1172.664120


TEST FAILED: number of errors:  849488718, max rel error: 0.000488
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$
```

From our results in 3 and 4, we can see a few things based on effect size and block size. As N increases, we see a slight decrease in GFLOPS. One reason for this could be that larger matrices increase global memory traffic. The fixed shared-memory tile then becomes less

effective relative to total work. We also might see diminishing cache performances with larger N and we see the Kernel becomes more DRAM-bandwidth bound.

For block size, when we set the footprint size to 16, each tile will load in 512 bytes from A to B in each iteration. However when we increase this to 32, we get each thread computing 4 outputs instead of just one. We see the GFLOPS essentially double as the arithmetic intensity increases.

5.

1. Coalesced memory access drastically improves performance
    a. In the vector-add experiment, switching from non-coalesced (vecadd00) to coalesced (vecadd01) memory access improved throughput by ~2.2–2.5×.
    b. If a kernel is bandwidth-bound, fixing memory access patterns often gives the largest speedup.
2. Increase the amount of work per thread to improve arithmetic intensity
    a. In matmult00, each thread computes one C element. While in matmult01, each thread computes four C elements using a larger tile (2×2 outputs/thread). This reduces the ratio of expensive global memory loads to useful computation.Resulting in matmult01 achieved ~1.7–2× higher GFlops/s across N = 256, 512, 1024.

3. Larger tiling (shared memory footprints) increases data reuse
    a. matmult00 uses 16×16 shared-memory tiles while matmult01 uses 32×32 tiles. We saw that larger tiles mean each element loaded into shared memory is reused more times which improves data locality and reduces pressure on global memory bandwidth.

4. Occupancy enables performance; memory efficiency and arithmetic intensity *deliver* it.

5. Most real CUDA kernels are bandwidth-bound; design for memory efficiency first.

Part B:

1. Q1
    a. K = 1 million
        i. N = 1000000 elements
        ii. CPU add time: 3.52771 ms

    b. K = 5 million
        i. N = 5000000 elements
        ii. CPU add time: 18.3888 ms

    c. \K = 10 million
        i. N = 10000000 elements
        ii. CPU add time: 36.1951 ms

    d. K = 50 million
        i. N = 50000000 elements
        ii. CPU add time: 176.674 ms

    e. K = 100 million
        i. CPU add time: 353.106 ms

```
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$ ./qB1 1
K = 1 million
N = 1000000 elements
CPU add time: 3.52771 ms
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$ ./qB1 5
K = 5 million
N = 5000000 elements
CPU add time: 18.3888 ms
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$ ./qB1 10
K = 10 million
N = 10000000 elements
CPU add time: 36.1951 ms
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$ ./qB1 50
K = 50 million
N = 50000000 elements
CPU add time: 176.674 ms
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$ ./qB1 100
K = 100 million
N = 100000000 elements
CPU add time: 353.106 ms
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$
```

2. - Without unified memory
    a. K = 1
       K = 1 million, N = 1000000 elements

       Scenario 1: 1 block, 1 thread
       GPU time (including memcpy) for scenario 1: 213.366 ms

       Scenario 2: 1 block, 256 threads
       GPU time (including memcpy) for scenario 2: 4.10298 ms

       Scenario 3: 3907 blocks, 256 threads per block (total threads ≈ N)
       GPU time (including memcpy) for scenario 3: 1.89882 ms

    b. K = 5 million, N = 5000000 elements

       Scenario 1: 1 block, 1 thread
       GPU time (including memcpy) for scenario 1: 637.023 ms

       Scenario 2: 1 block, 256 threads
       GPU time (including memcpy) for scenario 2: 15.9903 ms

       Scenario 3: 19532 blocks, 256 threads per block (total threads ≈ N)
       GPU time (including memcpy) for scenario 3: 8.83175 ms

    c. K = 10 million, N = 10000000 elements

       Scenario 1: 1 block, 1 thread
       GPU time (including memcpy) for scenario 1: 1254.32 ms

       Scenario 2: 1 block, 256 threads
       GPU time (including memcpy) for scenario 2: 31.5532 ms

       Scenario 3: 39063 blocks, 256 threads per block (total threads ≈ N)
       GPU time (including memcpy) for scenario 3: 17.5787 ms

    d. K = 50 million, N = 50000000 elements

       Scenario 1: 1 block, 1 thread
       ./qB2GPU time (including memcpy) for scenario 1: 5808.06 ms

       Scenario 2: 1 block, 256 threads
       GPU time (including memcpy) for scenario 2: 155.824 ms

       Scenario 3: 195313 blocks, 256 threads per block (total threads ≈ N)
       GPU time (including memcpy) for scenario 3: 86.0508 ms

e. K = 100 million, N = 100000000 elements

Scenario 1: 1 block, 1 thread
GPU time (including memcpy) for scenario 1: 11567.9 ms

Scenario 2: 1 block, 256 threads
GPU time (including memcpy) for scenario 2: 312.366 ms

Scenario 3: 390625 blocks, 256 threads per block (total threads ≈ N)
GPU time (including memcpy) for scenario 3: 178.119 ms

```
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$ ./qB2 1
K = 1 million, N = 1000000 elements

Scenario 1: 1 block, 1 thread
GPU time (including memcpy) for scenario 1: 213.366 ms

Scenario 2: 1 block, 256 threads
GPU time (including memcpy) for scenario 2: 4.10298 ms

Scenario 3: 3907 blocks, 256 threads per block (total threads ≈ N)
GPU time (including memcpy) for scenario 3: 1.89882 ms
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$ ./qB2 5
K = 5 million, N = 5000000 elements

Scenario 1: 1 block, 1 thread
GPU time (including memcpy) for scenario 1: 637.023 ms

Scenario 2: 1 block, 256 threads
GPU time (including memcpy) for scenario 2: 15.9903 ms

Scenario 3: 19532 blocks, 256 threads per block (total threads ≈ N)
GPU time (including memcpy) for scenario 3: 8.83175 ms
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$ ./qB2 10
K = 10 million, N = 10000000 elements

Scenario 1: 1 block, 1 thread
GPU time (including memcpy) for scenario 1: 1254.32 ms

Scenario 2: 1 block, 256 threads
GPU time (including memcpy) for scenario 2: 31.5532 ms

Scenario 3: 39063 blocks, 256 threads per block (total threads ≈ N)
GPU time (including memcpy) for scenario 3: 17.5787 ms
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$ ./qB2 50
K = 50 million, N = 50000000 elements

Scenario 1: 1 block, 1 thread
./qB2GPU time (including memcpy) for scenario 1: 5808.06 ms

Scenario 2: 1 block, 256 threads
GPU time (including memcpy) for scenario 2: 155.824 ms

Scenario 3: 195313 blocks, 256 threads per block (total threads ≈ N)
GPU time (including memcpy) for scenario 3: 86.0508 ms
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$ ./qB2 100
K = 100 million, N = 100000000 elements

Scenario 1: 1 block, 1 thread
GPU time (including memcpy) for scenario 1: 11567.9 ms

Scenario 2: 1 block, 256 threads
GPU time (including memcpy) for scenario 2: 312.366 ms

Scenario 3: 390625 blocks, 256 threads per block (total threads ≈ N)
GPU time (including memcpy) for scenario 3: 178.119 ms
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$
```

3. - With unified memory
    a. K = 1 million, N = 1000000 elements

        Scenario 1: 1 block, 1 thread
        Unified Memory GPU time (kernel + migrations) for scenario 1: 211.736 ms

        Scenario 2: 1 block, 256 threads
        Unified Memory GPU time (kernel + migrations) for scenario 2: 2.67944 ms

        Scenario 3: 3907 blocks, 256 threads per block (total threads ≈ N)
        Unified Memory GPU time (kernel + migrations) for scenario 3: 0.263739 ms

    b. K = 5 million, N = 5000000 elements

        Scenario 1: 1 block, 1 thread
        Unified Memory GPU time (kernel + migrations) for scenario 1: 617.301 ms

        Scenario 2: 1 block, 256 threads
        Unified Memory GPU time (kernel + migrations) for scenario 2: 7.82158 ms

        Scenario 3: 19532 blocks, 256 threads per block (total threads ≈ N)
        Unified Memory GPU time (kernel + migrations) for scenario 3: 0.558965 ms

    c. K = 10 million, N = 10000000 elements

        Scenario 1: 1 block, 1 thread
        Unified Memory GPU time (kernel + migrations) for scenario 1: 1034.3 ms

        Scenario 2: 1 block, 256 threads
        Unified Memory GPU time (kernel + migrations) for scenario 2: 15.2769 ms

        Scenario 3: 39063 blocks, 256 threads per block (total threads ≈ N)
        Unified Memory GPU time (kernel + migrations) for scenario 3: 1.05063 ms

    d. K = 50 million, N = 50000000 elements

        Scenario 1: 1 block, 1 thread
        Unified Memory GPU time (kernel + migrations) for scenario 1: 4900.1 ms

        Scenario 2: 1 block, 256 threads
        Unified Memory GPU time (kernel + migrations) for scenario 2: 74.3238 ms

Scenario 3: 195313 blocks, 256 threads per block (total threads ≈ N)
Unified Memory GPU time (kernel + migrations) for scenario 3: 4.86015 ms

e. K = 100 million, N = 100000000 elements

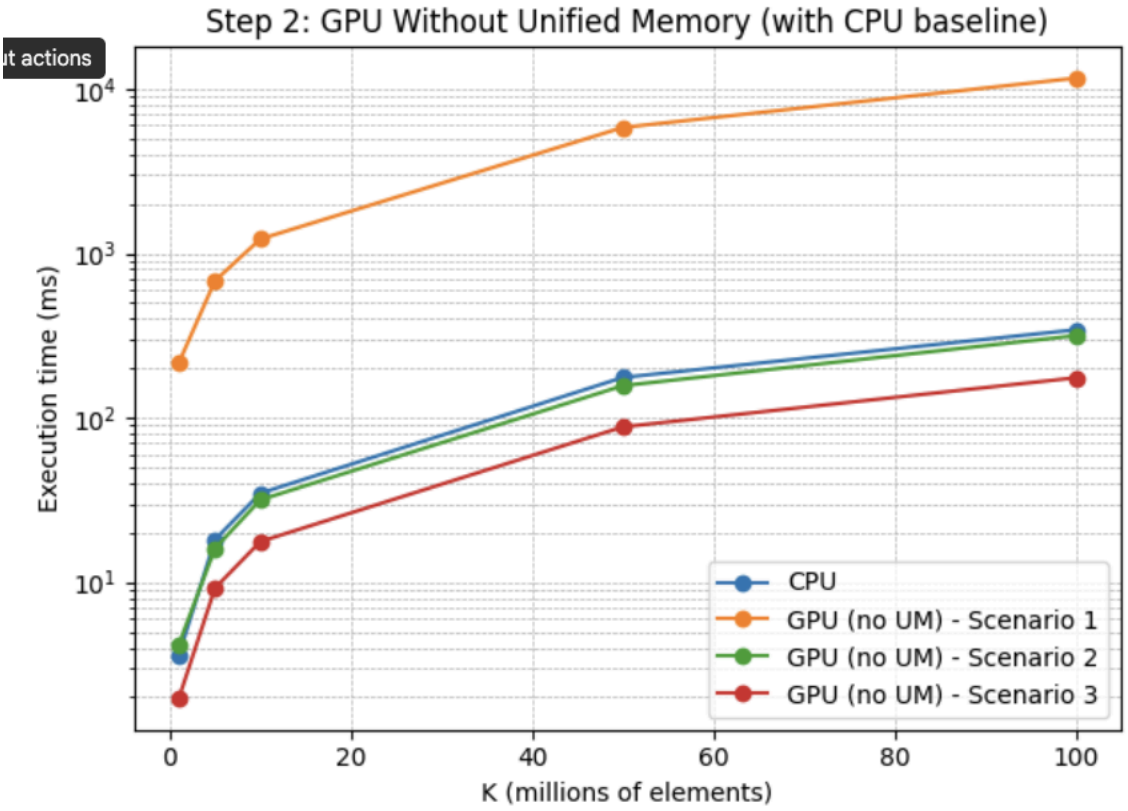
Scenario 1: 1 block, 1 thread
Unified Memory GPU time (kernel + migrations) for scenario 1: 9707.35 ms

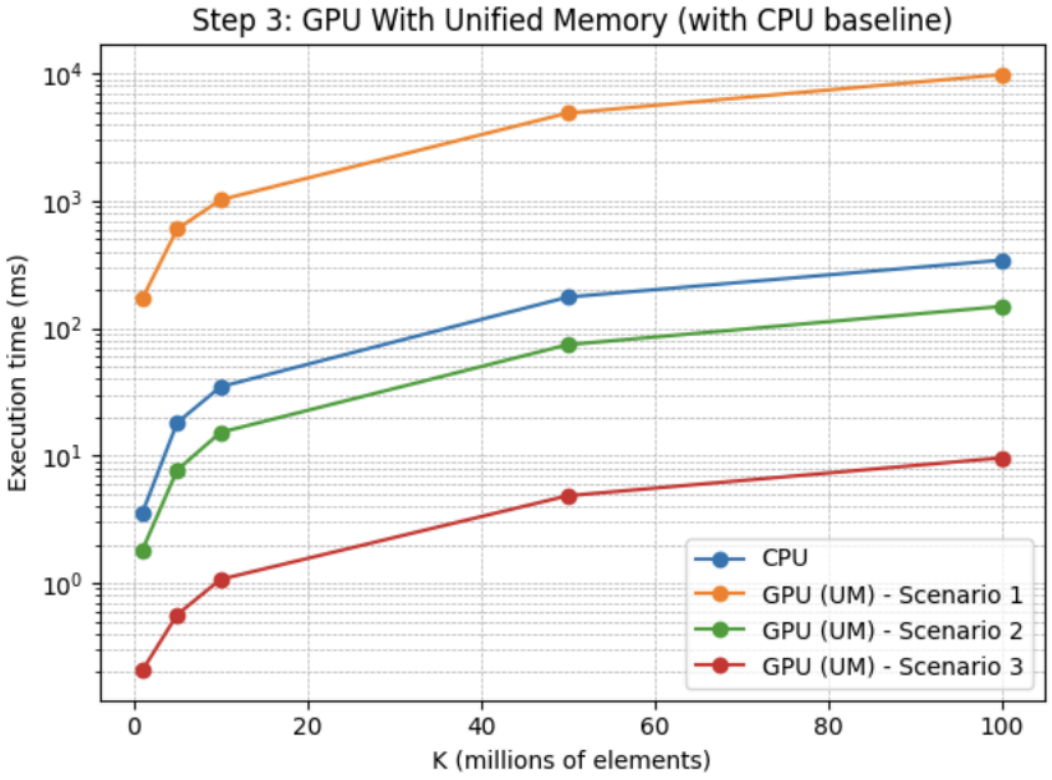
Scenario 2: 1 block, 256 threads
Unified Memory GPU time (kernel + migrations) for scenario 2: 148.22 ms

Scenario 3: 390625 blocks, 256 threads per block (total threads ≈ N)
Unified Memory GPU time (kernel + migrations) for scenario 3: 9.55671 ms

```
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$ ./qB3 1
K = 1 million, N = 1000000 elements

Scenario 1: 1 block, 1 thread
Unified Memory GPU time (kernel + migrations) for scenario 1: 211.736 ms

Scenario 2: 1 block, 256 threads
Unified Memory GPU time (kernel + migrations) for scenario 2: 2.67944 ms

Scenario 3: 3907 blocks, 256 threads per block (total threads ≈ N)
Unified Memory GPU time (kernel + migrations) for scenario 3: 0.263739 ms
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$ ./qB3 5
K = 5 million, N = 5000000 elements

Scenario 1: 1 block, 1 thread
Unified Memory GPU time (kernel + migrations) for scenario 1: 617.301 ms

Scenario 2: 1 block, 256 threads
Unified Memory GPU time (kernel + migrations) for scenario 2: 7.82158 ms

Scenario 3: 19532 blocks, 256 threads per block (total threads ≈ N)
Unified Memory GPU time (kernel + migrations) for scenario 3: 0.558965 ms
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$ ./qB3 10
K = 10 million, N = 10000000 elements

Scenario 1: 1 block, 1 thread
Unified Memory GPU time (kernel + migrations) for scenario 1: 1034.3 ms

Scenario 2: 1 block, 256 threads
Unified Memory GPU time (kernel + migrations) for scenario 2: 15.2769 ms

Scenario 3: 39063 blocks, 256 threads per block (total threads ≈ N)
Unified Memory GPU time (kernel + migrations) for scenario 3: 1.05063 ms
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$ ./qB3 50
K = 50 million, N = 50000000 elements

Scenario 1: 1 block, 1 thread
Unified Memory GPU time (kernel + migrations) for scenario 1: 4900.1 ms

Scenario 2: 1 block, 256 threads
Unified Memory GPU time (kernel + migrations) for scenario 2: 74.3238 ms

Scenario 3: 195313 blocks, 256 threads per block (total threads ≈ N)
Unified Memory GPU time (kernel + migrations) for scenario 3: 4.86015 ms
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$ ./qB3 100
K = 100 million, N = 100000000 elements

Scenario 1: 1 block, 1 thread
Unified Memory GPU time (kernel + migrations) for scenario 1: 9707.35 ms

Scenario 2: 1 block, 256 threads
Unified Memory GPU time (kernel + migrations) for scenario 2: 148.22 ms

Scenario 3: 390625 blocks, 256 threads per block (total threads ≈ N)
Unified Memory GPU time (kernel + migrations) for scenario 3: 9.55671 ms
(base) benjamincyna@hpml-assignemt3-cuda:~/CUDA1$
```

4.


Step 2: GPU Without Unified Memory (with CPU baseline)
Execution time (ms)
K (millions of elements)
CPU
GPU (no UM) - Scenario 1
GPU (no UM) - Scenario 2
GPU (no UM) - Scenario 3


Step 3: GPU With Unified Memory (with CPU baseline)
Execution time (ms)
K (millions of elements)
CPU
GPU (UM) - Scenario 1
GPU (UM) - Scenario 2
GPU (UM) - Scenario 3

Part B Analysis:

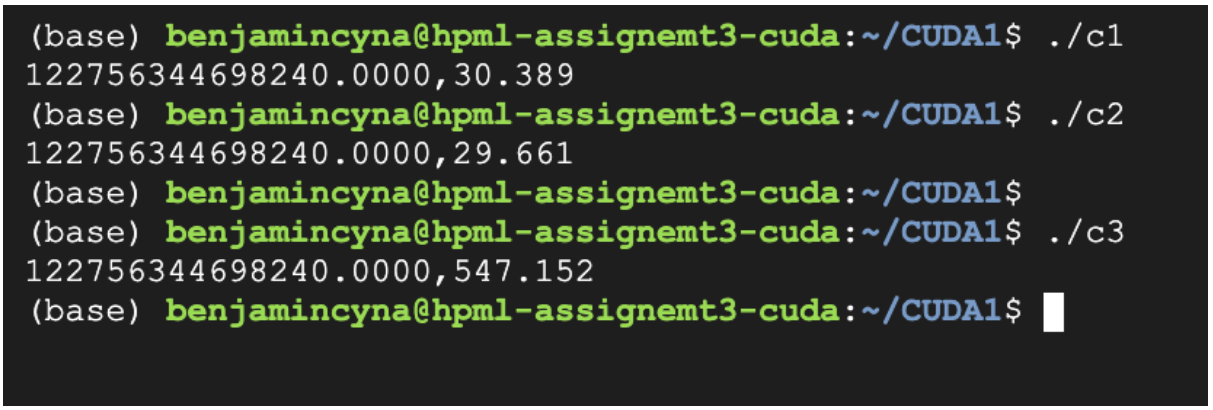
For our part B, we compare the performance of the CPU vs GPU, cudaMalloc() vs cudaMallocManaged() which allocates data in unified memory and plotted those comparisons. From the results, we can see that unified memory is faster than explicitly memcpy for large K. This could be because the unified memory does on-demand page migration, rather than full bulk copies. Unified Memory also does not copy back from GPU to CPU unless the host accesses the memory, which can account for the speedup. We can also see that scenario 3 is consistently the fastest. This is because we see the full grid with 256 threads per block. We get the highest occupancy which allows the highest memory bandwidth. And we also have enough threads to deal with latency constraints. Scenario one (with just one block and one thread) is very slow in comparison since there is no parallelism at all and the GPU executes serial code slower than the CPU.

Part C:

1. Checksum: 122756344698240.0000
   Time: 30.389

2. Checksum: 122756344698240.0000
   Time: 29.661

3. Checksum: 122756344698240.0000
   Time: 547.152



Part C analysis:

In part C, we implement convolution in CUDA in three ways, 1, simple naive CUDA, 2 Tiled CUDA, and 3, using cuDNN. Naive CUDA does not use shared memory, has many redundant loads, and as a result is memory bound, it took 30.389 ms to complete the program. In part two, we see using shared memory can also reduce memory traffic which in theory should give us a decent speedup, however, we don't actually see a speedup at all. This could be explained by a couple of reasons. One reason could be that the convolution is memory-bandwidth-limited, not compute-limited. The 3×3 filter size provides extremely low arithmetic intensity, so the naive kernel already saturates DRAM bandwidth, and shared-memory tiling cannot improve performance significantly. Another reason might be that modern GPUs also have effective L1/L2 caches that already exploit most of the spatial reuse, making the shared-memory loads in C2 largely redundant. Lastly, the overhead of shared-memory loading and synchronization might offset the small gains in data reuse.

For part 3, we use cuDNN which typically performs much better than hand written kernels. However, cuDNN is optimzed for floating point values FP32 or FP16. However, I used FP64 which could explain the massive slow down we get in part 3.