Setup Used: I just followed the setup posted in the Appendix section of the Assignment 3 PDF page. So I pretty much just used a n1-standard-8 (8 vCPU, 4 core, 50 GB of memory) VM with an NVIDIA T4 GPU for all the problems in this assignment.

Part A: CUDA Matrix Operations

Problem 1: Vector add and coalescing memory access
Q1:

```
(base) pkh2120@instance-20251107-20251113-170842:~/Part-A$ ./vecadd00 500
Total vector size: 3840000
Time: 0.001121 (sec), GFlopsS: 3.425378, GBytesS: 41.104536
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-A$ ./vecadd00 1000
Total vector size: 7680000
Time: 0.001932 (sec), GFlopsS: 3.975349, GBytesS: 47.704191
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-A$ ./vecadd00 2000
Total vector size: 15360000
Time: 0.004495 (sec), GFlopsS: 3.417202, GBytesS: 41.006424
Test PASSED
```

For this experiment, we just execute the provided vecadd00 program(uses a cuda kernel function to add two vectors without coalescing memory accesses) while varying the program argument k: the number of values that each thread is responsible for. Obviously, the total time the kernel function takes increases as we increase k since each thread is now responsible for more work. The speed of floating point operations and the bandwidth are also relatively similar across all three runs which makes sense since the kernel function remained the same across all runs and only the total number of elements each thread would handle was getting doubled in each run.

Q2:

```
(base) pkh2120@instance-20251107-20251113-170842:~/Part-A$ ./vecadd01 500
Total vector size: 3840000
Time: 0.000391 (sec), GFlopsS: 9.820809, GBytesS: 117.849712
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-A$ ./vecadd01 1000
Total vector size: 7680000
Time: 0.000798 (sec), GFlopsS: 9.621343, GBytesS: 115.456110
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-A$ ./vecadd01 2000
Total vector size: 15360000
Time: 0.001492 (sec), GFlopsS: 10.294744, GBytesS: 123.536931
Test PASSED
```

For this experiment, we just execute the newly created vecadd01 program(uses a cuda kernel function to add two vectors using coalesced memory reads) while varying the program argument k: the number of values that each thread is responsible for. Obviously, the total time the kernel function takes increases as we increase k since each thread is now responsible for more work. The speed of floating point operations and the bandwidth are also relatively similar across all three runs which makes sense since the kernel function remained the same across all

runs and only the total number of elements each thread would handle was getting doubled in each run. Also, this new program(vecadd01) takes significantly less time to perform the same work that was performed by the original program(vecadd00) for all three runs. Consequently, the speed of floating point operations and the bandwidth are also significantly higher for this new program as well.

Problem 2: Shared CUDA Matrix Multiply
Q3:

```
(base) pkh2120@instance-20251107-20251113-170842:~/Part-A$ ./matmult00 16
Data dimensions: 256x256
Grid Dimensions: 16x16
Block Dimensions: 16x16
Footprint Dimensions: 16x16
Time: 0.000099 (sec), nFlops: 33554432, GFlopsS: 339.126478
(base) pkh2120@instance-20251107-20251113-170842:~/Part-A$ ./matmult00 32
Data dimensions: 512x512
Grid Dimensions: 32x32
Block Dimensions: 16x16
Footprint Dimensions: 16x16
Time: 0.000567 (sec), nFlops: 268435456, GFlopsS: 473.465058
(base) pkh2120@instance-20251107-20251113-170842:~/Part-A$ ./matmult00 64
Data dimensions: 1024x1024
Grid Dimensions: 64x64
Block Dimensions: 16x16
Footprint Dimensions: 16x16
Time: 0.004717 (sec), nFlops: 2147483648, GFlopsS: 455.253943
```

For this experiment, we just execute the provided matmult00 program(uses a cuda kernel function to multiply two matrices where each thread computes one value in the resultant output matrix) while varying the program argument k: the dimension size for the grids. When k = 16, the grid shape is 16 blocks x 16 blocks, when k = 32, the grid shape is 32 x 32 blocks, when k = 64, the grid shape is 64 x 64 blocks. Also, each block automatically has 16 x 16 = 256 threads. The results show that as the value of k, the overall size of the matrices, and the size of the grid increases, the total time that the kernel function takes increases as well. This makes sense since as the matrix size and size of the grid increases, more threads need to be created so that each thread is able to compute just one value in the resultant output matrix.

Q4:

```
(base) pkh2120@instance-20251107-20251113-170842:~/Part-A$ ./matmult01 8
Data dimensions: 256x256
Grid Dimensions: 8x8
Block Dimensions: 16x16
Footprint Dimensions: 32x32
Time: 0.000057 (sec), nFlops: 33554432, GFlopsS: 588.859784
(base) pkh2120@instance-20251107-20251113-170842:~/Part-A$ ./matmult01 16
Data dimensions: 512x512
Grid Dimensions: 16x16
Block Dimensions: 16x16
Footprint Dimensions: 32x32
Time: 0.000286 (sec), nFlops: 268435456, GFlopsS: 938.249922
(base) pkh2120@instance-20251107-20251113-170842:~/Part-A$ ./matmult01 32
Data dimensions: 1024x1024
Grid Dimensions: 32x32
Block Dimensions: 16x16
Footprint Dimensions: 32x32
Time: 0.002207 (sec), nFlops: 2147483648, GFlopsS: 973.014935
```

For this experiment, we just execute the newly created matmult01 program(uses a cuda kernel function to multiply two matrices where each thread computes four values in the resultant output matrix) while varying the program argument k: the dimension size for the grids. When k = 8, the grid shape is 8 blocks x 8 blocks, when k = 16, the grid shape is 16 blocks x 16 blocks, when k = 32, the grid shape is 32 blocks x 32 blocks. Also, each block automatically has 16 x 16 = 256 threads. The results again show that as the value of k, the overall size of the matrices, and the size of the grid increases, the total time that the kernel function takes increases as well. This makes sense since as the matrix size and size of the grid increases, more threads need to be created so that each thread is responsible for computing just four values in the resultant output matrix. In addition, these new changes made in the matmult01 program clearly improve the overall efficiency of the matrix multiplication kernel function since the overall time that the kernel takes to complete the matrix multiplication is approximately halved in comparison to the matmult00 program for each run configuration(square matrices of each of the following sizes: 256, 512, 1024) tested. Consequently, the new changes made in the matmult01 program also show a drastic increase in speed of floating point operations across all three run configurations.
Speedup that new kernel achieves over the provided kernel:
Square Matrices Size 256: Speedup Achieved = 0.000099 / 0.000057 = 1.74
Square Matrices Size 512: Speedup Achieved = 0.000567 / 0.000286 = 1.98
Square Matrices Size 1024: Speedup Achieved = 0.004717 / 0.002207 = 2.14

Q5:

From the previous experiments and programs there's definitely a few rules of thumb that I can formulate easily for obtaining good performance when using CUDA. First, increasing the load that each thread handles instead of just creating more threads when the size of data increases can often result in much better overall performance. In addition, maximizing the usage of memory coalescing will always improve the performance of kernel functions as it helps to improve the efficiency of any global memory accesses. Also, shared memory should be used whenever possible or appropriate since accessing shared memory is much more efficient that accessing global memory.

Part B: CUDA Unified Memory

Q1:

```
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ g++ q1.cpp -o q1
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q1 1
Total size of each array: 4000000
Time: 0.002749 (sec), GFlopsS: 0.363742, GBytesS: 4.364899
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q1 5
Total size of each array: 20000000
Time: 0.013780 (sec), GFlopsS: 0.362842, GBytesS: 4.354099
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q1 10
Total size of each array: 40000000
Time: 0.027743 (sec), GFlopsS: 0.360453, GBytesS: 4.325437
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q1 50
Total size of each array: 200000000
Time: 0.139319 (sec), GFlopsS: 0.358889, GBytesS: 4.306665
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q1 100
Total size of each array: 400000000
Time: 0.277253 (sec), GFlopsS: 0.360682, GBytesS: 4.328178
Test PASSED
```

For this experiment, we just execute the newly created q1 C++ program(uses CPU to add the elements of two arrays) while varying the program argument k: the total length of each of the two arrays in millions. Obviously, the total time the program takes increases as we increase k since the size of each array increases by millions as k increases. The speed of floating point operations and the bandwidth are also relatively similar across all five runs which makes sense since the core algorithm used to add the two arrays always remained the same across all runs and only the total number of elements in each array was changing across runs.

Q2:

```
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ nvcc q2.cu -o q2
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q2 1 1 1
Time: 0.113437 (sec), GFlopsS: 0.008815, GBytesS: 0.105786
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q2 1 256 1
Time: 0.002198 (sec), GFlopsS: 0.454963, GBytesS: 5.459556
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q2 1 256 3
Time: 0.000720 (sec), GFlopsS: 1.388842, GBytesS: 16.666109
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q2 5 1 1
Time: 0.300549 (sec), GFlopsS: 0.016636, GBytesS: 0.199635
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q2 5 256 1
Time: 0.010550 (sec), GFlopsS: 0.473933, GBytesS: 5.687192
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q2 5 256 3
Time: 0.003510 (sec), GFlopsS: 1.424502, GBytesS: 17.094025
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q2 10 1 1
Time: 0.602630 (sec), GFlopsS: 0.016594, GBytesS: 0.199127
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q2 10 256 1
Time: 0.019459 (sec), GFlopsS: 0.513901, GBytesS: 6.166809
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q2 10 256 3
Time: 0.007036 (sec), GFlopsS: 1.421268, GBytesS: 17.055216
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q2 50 1 1
Time: 3.012353 (sec), GFlopsS: 0.016598, GBytesS: 0.199180
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q2 50 256 1
Time: 0.086969 (sec), GFlopsS: 0.574918, GBytesS: 6.899018
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q2 50 256 3
Time: 0.035145 (sec), GFlopsS: 1.422676, GBytesS: 17.072108
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q2 100 1 1
Time: 6.021396 (sec), GFlopsS: 0.016607, GBytesS: 0.199289
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q2 100 256 1
Time: 0.134431 (sec), GFlopsS: 0.743875, GBytesS: 8.926504
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q2 100 256 3
Time: 0.066698 (sec), GFlopsS: 1.499294, GBytesS: 17.991524
Test PASSED
```

For this experiment, we just execute the newly created q2 program(uses cuda kernel function to add the elements of two arrays) while varying the program arguments: k(the total length of each of the two arrays in millions), numberOfThreadsPerBlock(the number of threads in each block), and numberOfBlocks(the number of blocks in the grid). For each value of k(1, 5, 10, 50, 100) we can see that the total time decreases, speed of floating point operations increases, and bandwidth increases as we increase the number of threads per block and the number of blocks which makes sense since increasing these values allows for more parallelization. And obviously, the total time the program takes for each scenario increases as we increase k since the size of each array increases by millions as k increases.

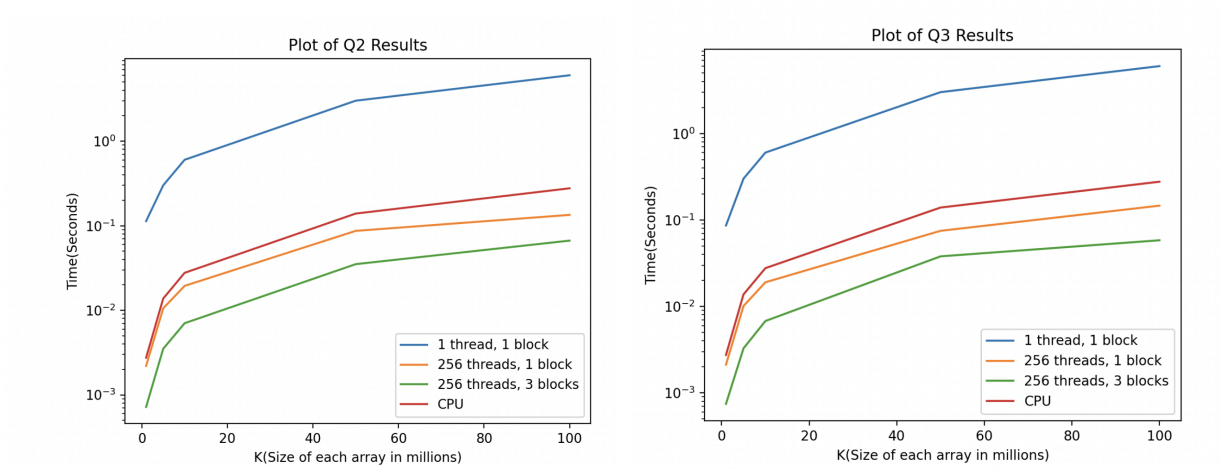
Q3:

```
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ nvcc q3.cu -o q3
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q3 1 1 1
Time: 0.086428 (sec), GFlopsS: 0.011570, GBytesS: 0.138844
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q3 1 256 1
Time: 0.002123 (sec), GFlopsS: 0.471058, GBytesS: 5.652701
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q3 1 256 3
Time: 0.000747 (sec), GFlopsS: 1.338750, GBytesS: 16.065001
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q3 5 1 1
Time: 0.300575 (sec), GFlopsS: 0.016635, GBytesS: 0.199617
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q3 5 256 1
Time: 0.010118 (sec), GFlopsS: 0.494180, GBytesS: 5.930161
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q3 5 256 3
Time: 0.003276 (sec), GFlopsS: 1.526200, GBytesS: 18.314405
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q3 10 1 1
Time: 0.602516 (sec), GFlopsS: 0.016597, GBytesS: 0.199165
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q3 10 256 1
Time: 0.019027 (sec), GFlopsS: 0.525569, GBytesS: 6.306829
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q3 10 256 3
Time: 0.006785 (sec), GFlopsS: 1.473806, GBytesS: 17.685670
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q3 50 1 1
Time: 3.011691 (sec), GFlopsS: 0.016602, GBytesS: 0.199224
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q3 50 256 1
Time: 0.074960 (sec), GFlopsS: 0.667022, GBytesS: 8.004270
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q3 50 256 3
Time: 0.037977 (sec), GFlopsS: 1.316587, GBytesS: 15.799044
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q3 100 1 1
Time: 6.022818 (sec), GFlopsS: 0.016604, GBytesS: 0.199242
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q3 100 256 1
Time: 0.146477 (sec), GFlopsS: 0.682701, GBytesS: 8.192413
Test PASSED
(base) pkh2120@instance-20251107-20251113-170842:~/Part-B$ ./q3 100 256 3
Time: 0.058270 (sec), GFlopsS: 1.716150, GBytesS: 20.593795
Test PASSED
```

For this experiment, we just execute the newly created q3 program(uses cuda kernel function to add the elements of two arrays using CUDA unified memory) while varying the program arguments: k(the total length of each of the two arrays in millions), numberOfThreadsPerBlock(the number of threads in each block), and numberOfBlocks(the number of blocks in the grid). For each value of k(1, 5, 10, 50, 100) we can see that the total time decreases, speed of floating point operations increases, and bandwidth increases as we increase the number of threads per block and the number of blocks which makes sense since increasing these values allows for more parallelization. And obviously, the total time the program takes for each scenario increases as we increase k since the size of each array increases by

millions as k increases. The metrics gathered across all runs are pretty similar for both q3 and q2 so I guess using CUDA unified memory mainly just provides convenience for developers since it clearly doesn't noticeably improve the performance of the cuda kernel function.

Q4:



Part C: Convolution in CUDA

C1:

```
(base) pkh2120@instance-20251107-20251113-170842:~/Part-C$ nvcc c1.cu -o c1
(base) pkh2120@instance-20251107-20251113-170842:~/Part-C$ ./c1
Checksum: 122756344698240.000000
Kernel Execution Time: 49.707008 ms
```

C2:

```
(base) pkh2120@instance-20251107-20251113-170842:~/Part-C$ nvcc c2.cu -o c2
(base) pkh2120@instance-20251107-20251113-170842:~/Part-C$ ./c2
Checksum: 122756344698240.000000
Kernel Execution Time: 108.154274 ms
```

C3:

```
(base) pkh2120@instance-20251107-20251113-170842:~/Part-C$ nvcc c3.cu -o c3 -lcudnn
(base) pkh2120@instance-20251107-20251113-170842:~/Part-C$ ./c3
Checksum: 122756344698240.000000
Kernel Execution Time: 1070.204956 ms
```

It's pretty weird how the shared memory convolutional kernel implementation(C2) was slower than the regular convolution kernel implementation(C1). Maybe we will only be able to see the performance benefits of shared memory if we test with larger convolutions or inputs. It's also really interesting to see how slow the cuDNN implementation(C3) performed in comparison to the other two convolutional kernel implementations. Maybe there's a lot of overhead from the many function calls that C3 makes which causes it to be slower than C2 and C1. In addition,

maybe we will only be able to see the performance benefits of cuDNN if we test with larger convolutions or inputs.