Denzel Farmer
Homework 3
High Performance ML

GPU Infrastructure Used

All development and experimentation done on GCP, with a T4 GPU and the following output for nvcc -V:

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2023 NVIDIA Corporation
Built on Mon_Apr__3_17:16:06_PDT_2023
Cuda compilation tools, release 12.1, V12.1.105
Build cuda_12.1.r12.1/compiler.32688072_0
```

Part A

Problem 1 - Vector Addition

For both parts of this problem, I include a bash script “P1-tests.sh” that runs vecadd00 and vecadd01 with each size (500, 1000, and 2000). The output produced is shown below:

```
Rebuilding...
(Q1) Naive implementation:
Total vector size: 3840000
Time: 0.001150 (sec), GFlopsS: 3.338749, GBytesS: 40.064993
Test PASSED
Total vector size: 7680000
Time: 0.002335 (sec), GFlopsS: 3.289314, GBytesS: 39.471771
Test PASSED
Total vector size: 15360000
Time: 0.004820 (sec), GFlopsS: 3.186808, GBytesS: 38.241695
Test PASSED

(Q2) Coalesced Implementation:
Total vector size: 3840000
Time: 0.000409 (sec), GFlopsS: 9.385855, GBytesS: 112.630261
Test PASSED
Total vector size: 7680000
Time: 0.000817 (sec), GFlopsS: 9.399549, GBytesS: 112.794589
Test PASSED
Total vector size: 15360000
Time: 0.001645 (sec), GFlopsS: 9.338239, GBytesS: 112.058866
Test PASSED
```

Q1 - Naive Implementation

As shown in the output above, the provided kernel functions correctly. The provided makefile entry builds vecadd00, which uses vecaddKernel00 to naively sum the vector.

This program sums the vector by assigning each thread a subset of sequential element pairs to load, sum, and write. This is inefficient because each thread in isolation makes non-coalesced accesses, which in reality load in larger chunks of wasted global memory.

The experiment runs each vector size twice, once as a ‘warm up’ and then once as the measured output.

```
(Q1) Naive implementation:
Total vector size: 3840000
Time: 0.001150 (sec), GFlopsS: 3.338749, GBytesS: 40.064993
Test PASSED
Total vector size: 7680000
Time: 0.002335 (sec), GFlopsS: 3.289314, GBytesS: 39.471771
Test PASSED
Total vector size: 15360000
Time: 0.004820 (sec), GFlopsS: 3.186808, GBytesS: 38.241695
Test PASSED
```

Q2 - Coalesced Implementation

The new kernel, vecaddKernel01, performs the same operation as vecaddKernel00 but in a coalesced way.

The kernel is coalesced by having each thread calculate 'strided' elements of the output vector. So, on the first iteration thread 0 calculates element 0, thread 1 calculates element 1, etc. Then on the second iteration, thread 0 calculates num_threads + 0, thread 1 calculates num_threads + 1, etc.

This allows the threads to cooperatively access global memory, and not reload large chunks of global memory. The performance is significantly better, as shown below:

```
(Q2) Coalesced Implementation:
Total vector size: 3840000
Time: 0.000409 (sec), GFlopsS: 9.385855, GBytesS: 112.630261
Test PASSED
Total vector size: 7680000
Time: 0.000817 (sec), GFlopsS: 9.399549, GBytesS: 112.794589
```

```
Test PASSED
Total vector size: 15360000
Time: 0.001645 (sec), GFlopsS: 9.338239, GBytesS: 112.058866
Test PASSED
```

The time to calculate each vector drops in proportion to the vector size, by around three times. Similarly, the throughput is around three times greater for the coalesced implementation.

Problem 2 - Shared CUDA Matrix Multiply

For both parts of this problem, I include a bash script “P2-tests.sh” that runs matmult00 and matmult01 with each data size (256x256, 512x512, 1024x1024). The output produced is shown below:

```
Rebuilding...
(Q3) Naive implementation:
Data dimensions: 256x256
Grid Dimensions: 16x16
Block Dimensions: 16x16
Footprint Dimensions: 16x16
Time: 0.000113 (sec), nFlops: 33554432, GFlopsS: 296.914532
Data dimensions: 512x512
Grid Dimensions: 32x32
Block Dimensions: 16x16
Footprint Dimensions: 16x16
Time: 0.000750 (sec), nFlops: 268435456, GFlopsS: 357.996791
Data dimensions: 1024x1024
Grid Dimensions: 64x64
Block Dimensions: 16x16
Footprint Dimensions: 16x16
Time: 0.005793 (sec), nFlops: 2147483648, GFlopsS: 370.712403

(Q4) Coalesced Implementation:
Data dimensions: 256x256
Grid Dimensions: 8x8
Block Dimensions: 16x16
Footprint Dimensions: 32x32
Time: 0.000068 (sec), nFlops: 33554432, GFlopsS: 492.089120
Data dimensions: 512x512
Grid Dimensions: 16x16
Block Dimensions: 16x16
Footprint Dimensions: 32x32
```

```
Time: 0.000354 (sec), nFlops: 268435456, GFlopsS: 758.692660
Data dimensions: 1024x1024
Grid Dimensions: 32x32
Block Dimensions: 16x16
Footprint Dimensions: 32x32
Time: 0.002624 (sec), nFlops: 2147483648, GFlopsS: 818.464267
```

Q3 - Initial matmultKernel00 Code

The initial code implements tiling, but only has a footprint size of 16. Each thread calculates 1 output value, and the memory loads are not coalesced. The output is shown below:

```
Rebuilding...
(Q3) Naive implementation:
Data dimensions: 256x256
Grid Dimensions: 16x16
Block Dimensions: 16x16
Footprint Dimensions: 16x16
Time: 0.000113 (sec), nFlops: 33554432, GFlopsS: 296.914532
Data dimensions: 512x512
Grid Dimensions: 32x32
Block Dimensions: 16x16
Footprint Dimensions: 16x16
Time: 0.000750 (sec), nFlops: 268435456, GFlopsS: 357.996791
Data dimensions: 1024x1024
Grid Dimensions: 64x64
Block Dimensions: 16x16
Footprint Dimensions: 16x16
Time: 0.005793 (sec), nFlops: 2147483648, GFlopsS: 370.712403
```

As the size of the matrix increases, the throughput of the implementation increases significantly. If the block dimension is not an even multiple of 16x16

Q4 - Coalesced matmultKernel01 Code

My new kernel computes 4 values per thread, and so the footprint size is 32x32. This means that each block has 16x16 threads. To maximize performance, I do the computation in three steps for each tiled multiplication phase.

First, I cache the full footprint size of 32x32 elements in shared memory for each phase. To maximize the efficiency of loading the tile into shared memory, I ensure that groups of 32 threads (‘warps’) load sequential chunks of memory locations from each input matrix based on their thread id.

Then, I calculate the partial output values in an unrolled loop for each of the 4 thread values, in an order different from the loading order–because loads from shared memory do not need to be coalesced.

Finally (after all phases), I store all output values in a shared memory tile and do a coalesced write into shared memory.

The output is significantly faster, with 2-3 times performance improvement over the non-coalesced tiled implementation:

```
(Q4) Coalesced Implementation:
Data dimensions: 256x256
Grid Dimensions: 8x8
Block Dimensions: 16x16
Footprint Dimensions: 32x32
Time: 0.000068 (sec), nFlops: 33554432, GFlopsS: 492.089120
Data dimensions: 512x512
Grid Dimensions: 16x16
Block Dimensions: 16x16
Footprint Dimensions: 32x32
Time: 0.000354 (sec), nFlops: 268435456, GFlopsS: 758.692660
Data dimensions: 1024x1024
Grid Dimensions: 32x32
Block Dimensions: 16x16
Footprint Dimensions: 32x32
Time: 0.002624 (sec), nFlops: 2147483648, GFlopsS: 818.464267
```

The improvement increases as the block size increases, but the tile size cannot increase past 32x32 without changing the algorithm, because of the maximum number of threads per block. Increase from 16x16 to 32x32 does not increase the performance, and smaller sizes seem to not fully take advantage of coalescing, and have worse performance.

Q5 - Rules of thumb for good CUDA performance

- When accessing lots of sequential information within a block, loading it temporarily into shared memory before making the transfer can increase performance significantly
- The ideal size for loading in chunks of sequential memory is 32, to match the size of a warp
- Larger block sizes do not necessarily mean better performance, but can help better reduce repeating from memory
- If statements placed in threads can significantly hurt performance
- Loop unrolling might help performance, but it is possible that the compiler already performs unrolling when it is efficient–I did not see any performance improvement when I unrolled small loops

Part B - CUDA Unified Memory

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

To run this question, I include “Generate-Data.sh” that runs each test with each K number, and both prints the output and saves it to a results.csv. The printed results are shown below:

```
Rebuilding...
Running with param = 1

Q1 (CPU):
K=1M
Blocks=1
Threads/block=1
Calculation Time: 1742us
Last Sum Value: 2e+06

Q2 (GPU):
K=1000192

Blocks=1
Threads/block=1
Calculation Time: 140994us
Last Sum Value: 1.00019e+06

Blocks=1
Threads/block=256
Calculation Time: 2363us
Last Sum Value: 1.00019e+06

Blocks=3907
Threads/block=256
Calculation Time: 123us
Last Sum Value: 1.00019e+06

Q3 (GPU):
K=1000192

Blocks=1
Threads/block=1
Calculation Time: 141664us
Last Sum Value: 1.00019e+06
```

```
Blocks=1
Threads/block=256
Calculation Time: 3162us
Last Sum Value: 1.00019e+06

Blocks=3907
Threads/block=256
Calculation Time: 908us
Last Sum Value: 1.00019e+06
Running with param = 5

Q1 (CPU):
K=5M
Blocks=1
Threads/block=1
Calculation Time: 8579us
Last Sum Value: 1e+07

Q2 (GPU):
K=5000192

Blocks=1
Threads/block=1
Calculation Time: 449096us
Last Sum Value: 5.00019e+06

Blocks=1
Threads/block=256
Calculation Time: 6838us
Last Sum Value: 5.00019e+06

Blocks=19532
Threads/block=256
Calculation Time: 315us
Last Sum Value: 5.00019e+06

Q3 (GPU):
K=5000192

Blocks=1
Threads/block=1
Calculation Time: 461517us
```

```
Last Sum Value: 5.00019e+06

Blocks=1
Threads/block=256
Calculation Time: 6771us
Last Sum Value: 5.00019e+06

Blocks=19532
Threads/block=256
Calculation Time: 253us
Last Sum Value: 5.00019e+06
Running with param = 10

Q1 (CPU):
K=10M
Blocks=1
Threads/block=1
Calculation Time: 17925us
Last Sum Value: 2e+07

Q2 (GPU):
K=10000128

Blocks=1
Threads/block=1
Calculation Time: 747243us
Last Sum Value: 1.00001e+07

Blocks=1
Threads/block=256
Calculation Time: 13649us
Last Sum Value: 1.00001e+07

Blocks=39063
Threads/block=256
Calculation Time: 534us
Last Sum Value: 1.00001e+07

Q3 (GPU):
K=10000128

Blocks=1
Threads/block=1
```

```
Calculation Time: 743634us
Last Sum Value: 1.00001e+07

Blocks=1
Threads/block=256
Calculation Time: 13613us
Last Sum Value: 1.00001e+07

Blocks=39063
Threads/block=256
Calculation Time: 488us
Last Sum Value: 1.00001e+07
Running with param = 50

Q1 (CPU):
K=50M
Blocks=1
Threads/block=1
Calculation Time: 86727us
Last Sum Value: 1e+08

Q2 (GPU):
K=50000128

Blocks=1
Threads/block=1
Calculation Time: 3201279us
Last Sum Value: 5.00001e+07

Blocks=1
Threads/block=256
Calculation Time: 67227us
Last Sum Value: 5.00001e+07

Blocks=195313
Threads/block=256
Calculation Time: 2400us
Last Sum Value: 5.00001e+07

Q3 (GPU):
K=50000128

Blocks=1
```

```
Threads/block=1
Calculation Time: 3198078us
Last Sum Value: 5.00001e+07

Blocks=1
Threads/block=256
Calculation Time: 67604us
Last Sum Value: 5.00001e+07

Blocks=195313
Threads/block=256
Calculation Time: 2357us
Last Sum Value: 5.00001e+07
Running with param = 100

Q1 (CPU):
K=100M
Blocks=1
Threads/block=1
Calculation Time: 175435us
Last Sum Value: 2e+08

Q2 (GPU):
K=100000000

Blocks=1
Threads/block=1
Calculation Time: 6223706us
Last Sum Value: 1e+08

Blocks=1
Threads/block=256
Calculation Time: 134636us
Last Sum Value: 1e+08

Blocks=390625
Threads/block=256
Calculation Time: 4712us
Last Sum Value: 1e+08

Q3 (GPU):
K=100000000
```

```
Blocks=1
Threads/block=1
Calculation Time: 6221558us
Last Sum Value: 1e+08

Blocks=1
Threads/block=256
Calculation Time: 134411us
Last Sum Value: 1e+08

Blocks=390625
Threads/block=256
Calculation Time: 4697us
Last Sum Value: 1e+08
```

Q1 - CPU/C++ Program

Write a C++ program that adds the elements of two arrays with a K million elements each. Here K is a command-line parameter of the program. Profile and get the time to execute this program for K=1,5,10,50,100. Use free() to free the memory at the end of the program.

My C++ program “q1.cpp” simply loops through the input arrays and sums all elements. The output, extracted from above is:

```
Q1 (CPU):
K=1M
Blocks=1
Threads/block=1
Calculation Time: 1742us
Last Sum Value: 2e+06

Q1 (CPU):
K=5M
Blocks=1
Threads/block=1
Calculation Time: 8579us
Last Sum Value: 1e+07

Q1 (CPU):
K=10M
```

```
Blocks=1
Threads/block=1
Calculation Time: 17925us
Last Sum Value: 2e+07

Q1 (CPU):
K=50M
Blocks=1
Threads/block=1
Calculation Time: 86727us
Last Sum Value: 1e+08

Q1 (CPU):
K=100M
Blocks=1
Threads/block=1
Calculation Time: 175435us
Last Sum Value: 2e+08
```

As shown, this is quite slow and time increases linearly with input size.

Q2 - CUDA w/o Unified Memory

This implementation uses the strided approach from Part-A, and explicitly allocates separate host and device memory and copies it to the device. The output is shown below:

```
Q2 (GPU):
K=1000192

Blocks=1
Threads/block=1
Calculation Time: 140994us
Last Sum Value: 1.00019e+06

Blocks=1
Threads/block=256
Calculation Time: 2363us
Last Sum Value: 1.00019e+06

Blocks=3907
Threads/block=256
Calculation Time: 123us
```

```
Last Sum Value: 1.00019e+06

Q2 (GPU):
K=5000192

Blocks=1
Threads/block=1
Calculation Time: 449096us
Last Sum Value: 5.00019e+06

Blocks=1
Threads/block=256
Calculation Time: 6838us
Last Sum Value: 5.00019e+06

Blocks=19532
Threads/block=256
Calculation Time: 315us
Last Sum Value: 5.00019e+06

Q2 (GPU):
K=10000128

Blocks=1
Threads/block=1
Calculation Time: 747243us
Last Sum Value: 1.00001e+07

Blocks=1
Threads/block=256
Calculation Time: 13649us
Last Sum Value: 1.00001e+07

Blocks=39063
Threads/block=256
Calculation Time: 534us
Last Sum Value: 1.00001e+07

Q2 (GPU):
K=50000128

Blocks=1
Threads/block=1
```

```
Calculation Time: 3201279us
Last Sum Value: 5.00001e+07

Blocks=1
Threads/block=256
Calculation Time: 67227us
Last Sum Value: 5.00001e+07

Blocks=195313
Threads/block=256
Calculation Time: 2400us
Last Sum Value: 5.00001e+07

Q2 (GPU):
K=100000000

Blocks=1
Threads/block=1
Calculation Time: 6223706us
Last Sum Value: 1e+08

Blocks=1
Threads/block=256
Calculation Time: 134636us
Last Sum Value: 1e+08

Blocks=390625
Threads/block=256
Calculation Time: 4712us
Last Sum Value: 1e+08
```

This approach is significantly faster than the CPU only approach (when using more than 1 thread/1 block), and still scales linearly with input size. As the number of threads and blocks increases, the performance also increases as more of the GPU's parallel capabilities are utilized.

Q3 - CUDA w/ Unified Memory

This implementation is very similar to Q2, but rather than explicitly separating device and host memory, it uses cudaMallocManaged to allocate memory, and replaces cudaMemcpy operations with cudaMemPrefetchAsync calls. Note that like Q2, sometimes it shifts the input size slightly to be a multiple of 256. Again, the output of the runner script:

```
Q3 (GPU):
K=1000192

Blocks=1
Threads/block=1
Calculation Time: 141664us
Last Sum Value: 1.00019e+06

Blocks=1
Threads/block=256
Calculation Time: 3162us
Last Sum Value: 1.00019e+06

Blocks=3907
Threads/block=256
Calculation Time: 908us
Last Sum Value: 1.00019e+06
Running with param = 5

Q3 (GPU):
K=5000192

Blocks=1
Threads/block=1
Calculation Time: 461517us
Last Sum Value: 5.00019e+06

Blocks=1
Threads/block=256
Calculation Time: 6771us
Last Sum Value: 5.00019e+06

Blocks=19532
Threads/block=256
Calculation Time: 253us
Last Sum Value: 5.00019e+06
Running with param = 10

Q3 (GPU):
K=10000128

Blocks=1
Threads/block=1
```

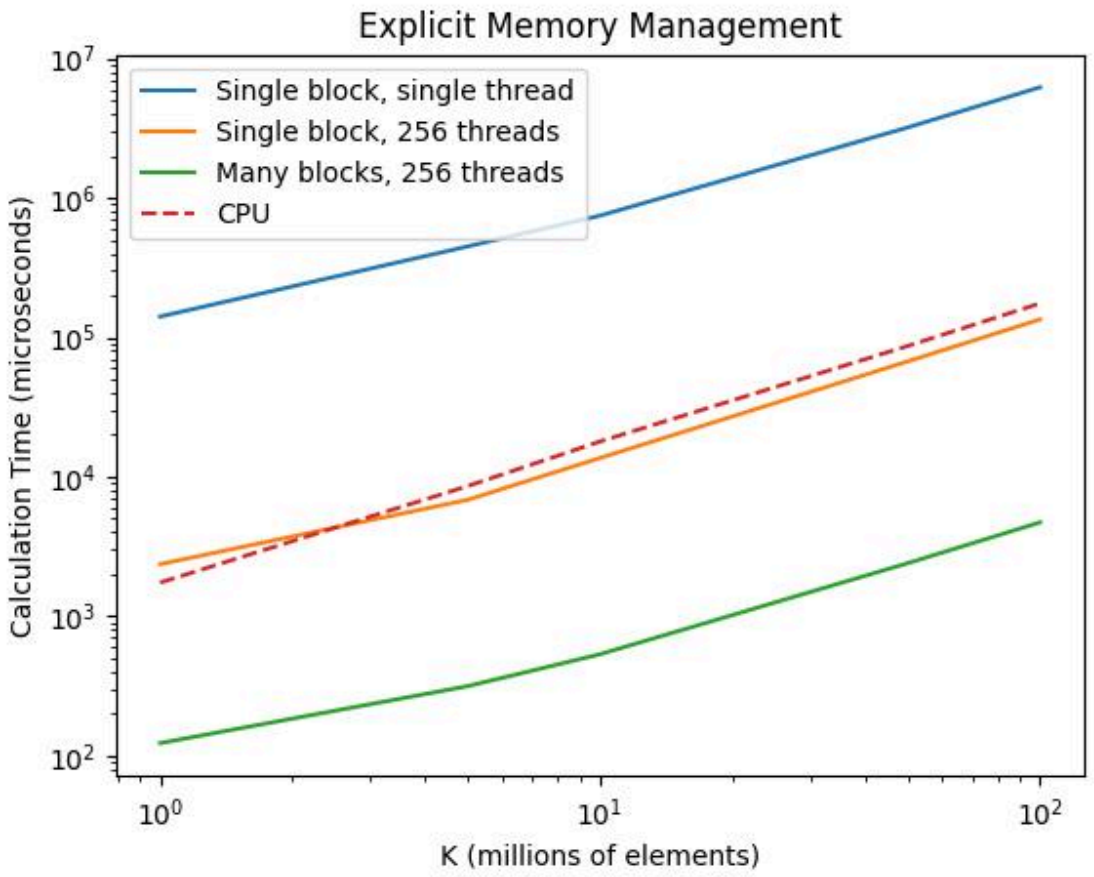
```
Calculation Time: 743634us
Last Sum Value: 1.00001e+07

Blocks=1
Threads/block=256
Calculation Time: 13613us
Last Sum Value: 1.00001e+07

Blocks=39063
Threads/block=256
Calculation Time: 488us
Last Sum Value: 1.00001e+07
Running with param = 50

Q3 (GPU):
K=50000128

Blocks=1
Threads/block=1
Calculation Time: 3198078us
Last Sum Value: 5.00001e+07

Blocks=1
Threads/block=256
Calculation Time: 67604us
Last Sum Value: 5.00001e+07

Blocks=195313
Threads/block=256
Calculation Time: 2357us
Last Sum Value: 5.00001e+07
Running with param = 100

Q3 (GPU):
K=100000000

Blocks=1
Threads/block=1
Calculation Time: 6221558us
Last Sum Value: 1e+08

Blocks=1
Threads/block=256
```

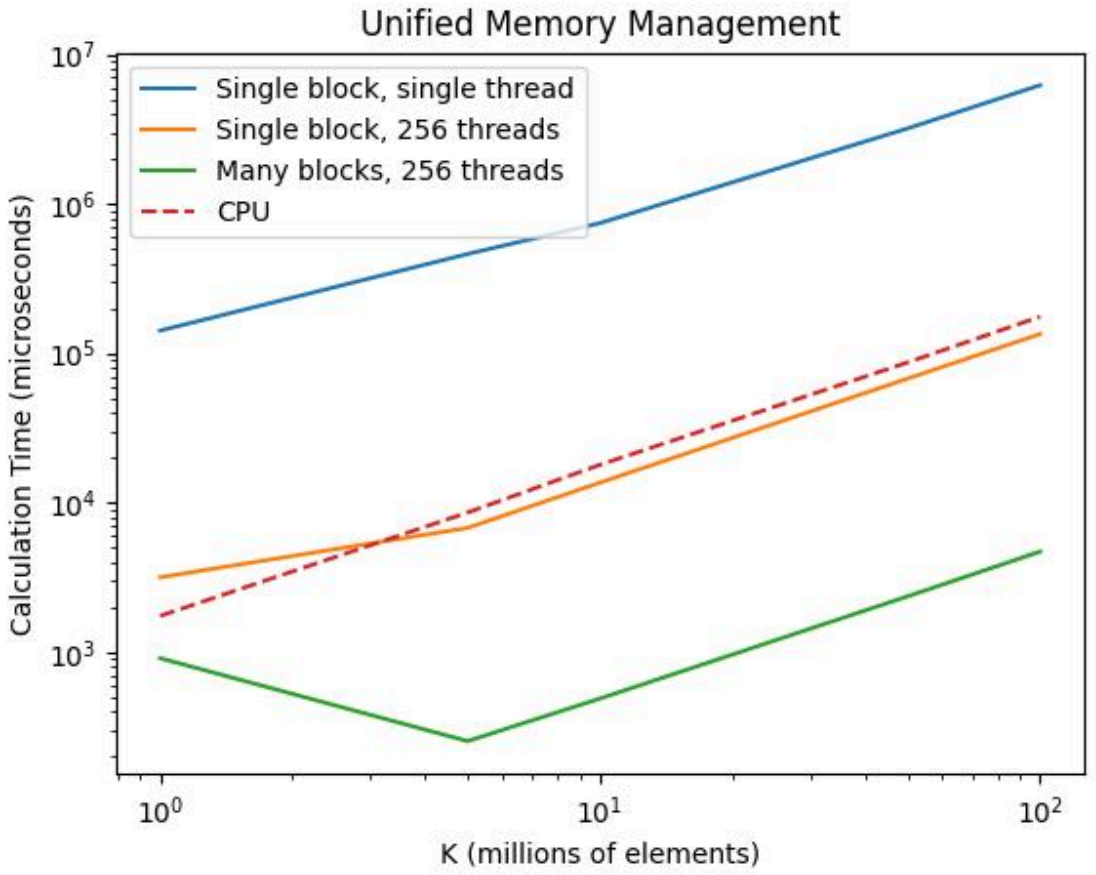
```
Calculation Time: 134411us
Last Sum Value: 1e+08

Blocks=390625
Threads/block=256
Calculation Time: 4697us
Last Sum Value: 1e+08
```

The performance (with prefetching) is almost identical to the explicitly-controlled version.

Q4 - Chart Plotting

The provided plotting code “q4.py” requires running the ./generate_data.sh script to produce a results.csv file. The python code reads the csv file and plots it, with the output shown below on log-log scales:





As shown in the above charts, the scaling and absolute performance of the unified and explicit memory management techniques are quite similar, if prefetching is used to indicate when the unified memory system should probably do a transfer under the hood.

There is a slight outlier for the K=1 case, where the overhead of the many-block, many-thread implementation appears to be enough to disrupt the linear performance scaling of larger counts. More investigation would be required to pinpoint exactly why this happens.

Part C: Convolution in CUDA

For these three implementations, I included a simple runner script ‘compare-convolutions.sh’ that runs each script with warmup, printing their times. The output is shown below:

```
Rebuilding...
(C1) Running simple convolution:
Checksum: 122756344698240.000000
Kernel execution time: 58ms
```

```
(C2) Running shared memory convolution:
Checksum: 122756344698240.000000
Kernel execution time: 20ms

(C3) Running cuDNN convolution:
Checksum: 122756344698240.000000
Convolution forward pass time: 2.065140 ms
```

Note that they all give the same checksum. For all implementations, I switched my datatype from double to float, since my T4 was so limited by memory utilization when using double that the differences in the implementations were not evident. The data type can easily be changed with the macro in the header included by C1 and C2.

C1: Simple Convolution in CUDA

The simple convolution does not use shared memory, and simply sets a block size of 32x32x1. Each thread in each block calculates one output element, calling a helper function to retrieve each value from the input image and filter in global memory. Then, each thread writes one result to the global memory output.

The input image is padded before being loaded into the convolution, to avoid synchronization issues from boundary guards.

Timing this on the T4 with datatype float, as shown below, gives around 58ms:

```
(C1) Running simple convolution:
Checksum: 122756344698240.000000
Kernel execution time: 58ms
```

C2: Shared Memory Convolution in CUDA

The shared memory convolution improves on C1 performance by pre-loading shared memory for an entire block of threads in a coalesced way. The block size is still 32x32x1, but there is now a ‘tile size’ of 34x34x3 over the input image for each block, since the edge threads in every block need a bit of overlapping data. The input image is padded before being loaded into the convolution, to avoid synchronization issues from boundary guards.

The kernel first loads in the entire tile of input, with each thread loading in a single block and a few threads loading in the extra ‘strips’ of required tile. While this introduces some lack of synchronization, it is made up for by the memory access performance.

Then the kernel loads in the current 3x3x3 filter, also to shared memory.

With the filter and input tile, the kernel computes one output element with each thread, and writes out the final values.

The performance is roughly ⅓ that of the simple convolution, and could likely be optimized further by reducing synchronization issues:

```
(C2) Running shared memory convolution:
Checksum: 122756344698240.000000
Kernel execution time: 20ms
```

C3: CuDNN Convolution in CUDA

The CuDNN implementation involved initializing descriptors, initializing images in memory, associating the two, and calling cudnnConvolutionForward with a number of arguments. Rather than explicitly padding, the implementation simply sets the padding input to cuDNN to be 1.

This implementation is significantly faster than the other two, likely because the cuDNN convolution kernel is highly optimized:

```
(C3) Running cuDNN convolution:
Checksum: 122756344698240.000000
Convolution forward pass time: 2.065140 ms
```