COMPUTE ENVIRONMENT
NVIDIA T4 GPU, n1-standard-8 (8 vCPUs, 4 core, 30 GB Memory) machine for everything

Part A:

Q1:

```
(base) si2468@instance-20250329-183730:~/hpml_assignments/HW3/CUDA1$ ./vecadd00 500
Total vector size: 3840000
Time: 0.000800 (sec), GFlopsS: 4.800634, GBytesS: 57.607609
Test PASSED
(base) si2468@instance-20250329-183730:~/hpml_assignments/HW3/CUDA1$ ./vecadd00 1000
Total vector size: 7680000
Time: 0.001669 (sec), GFlopsS: 4.601093, GBytesS: 55.213121
Test PASSED
(base) si2468@instance-20250329-183730:~/hpml_assignments/HW3/CUDA1$ ./vecadd00 2000
Total vector size: 15360000
Time: 0.003393 (sec), GFlopsS: 4.527054, GBytesS: 54.324651
Test PASSED
```

In this experiment, we experiment with different values for the number of elements each thread in the grid operates on. As we double the values per thread, we roughly double the amount of time the operation takes to run, which makes sense since each thread is doing twice as much work. We don't see a significant difference in GFlopsS or GBytesS.

This experiment was done using a kernel that assigns each thread a unique array chunk. Thus memory accesses will not be coalesced.

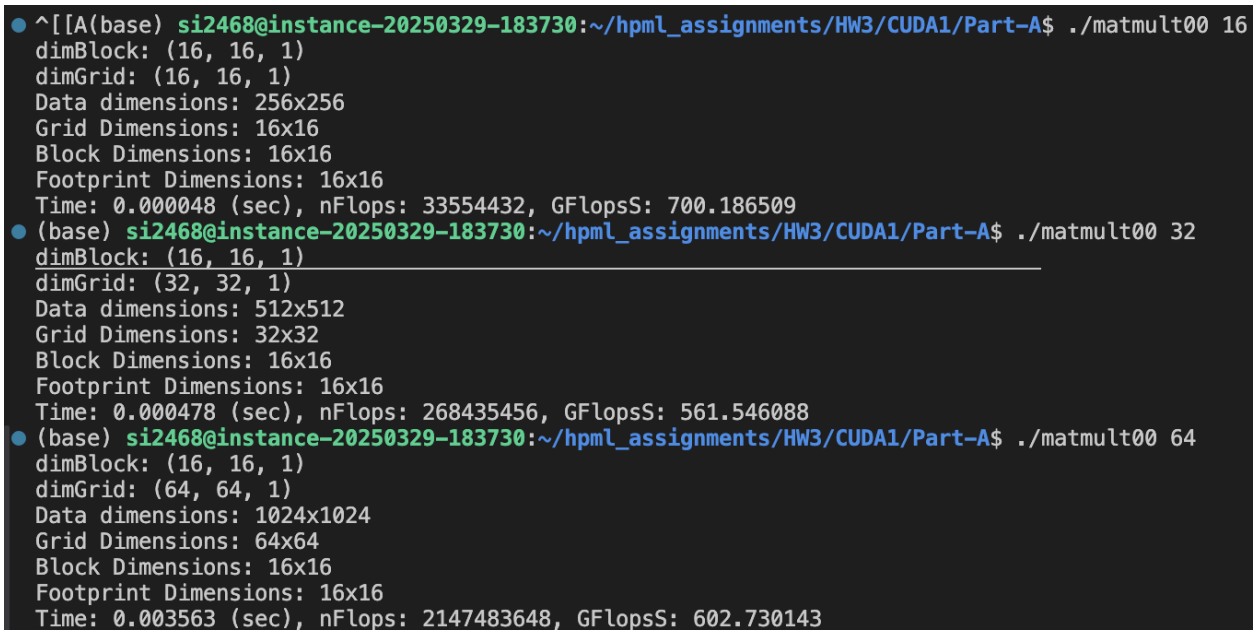
Q2:

```
(base) si2468@instance-20250329-183730:~/hpml_assignments/HW3/CUDA1$ ./vecadd01 500
Total vector size: 3840000
Time: 0.000349 (sec), GFlopsS: 11.008973, GBytesS: 132.107675
Test PASSED
(base) si2468@instance-20250329-183730:~/hpml_assignments/HW3/CUDA1$ ./vecadd01 1000
Total vector size: 7680000
Time: 0.000593 (sec), GFlopsS: 12.952254, GBytesS: 155.427043
Test PASSED
(base) si2468@instance-20250329-183730:~/hpml_assignments/HW3/CUDA1$ ./vecadd01 2000
Total vector size: 15360000
Time: 0.001144 (sec), GFlopsS: 13.427368, GBytesS: 161.128410
Test PASSED
```

In this experiment, we performed the same experiment as the previous one, except we used a CUDA kernel that employs memory coalescing. This is done by having consecutive threads operate on consecutive elements in the array, to reduce the effect of global memory accesses via DRAM bursts. In this experiment, we also see that the time roughly doubles as the number of elements per thread doubles, which still makes sense. However, the times here are about 3

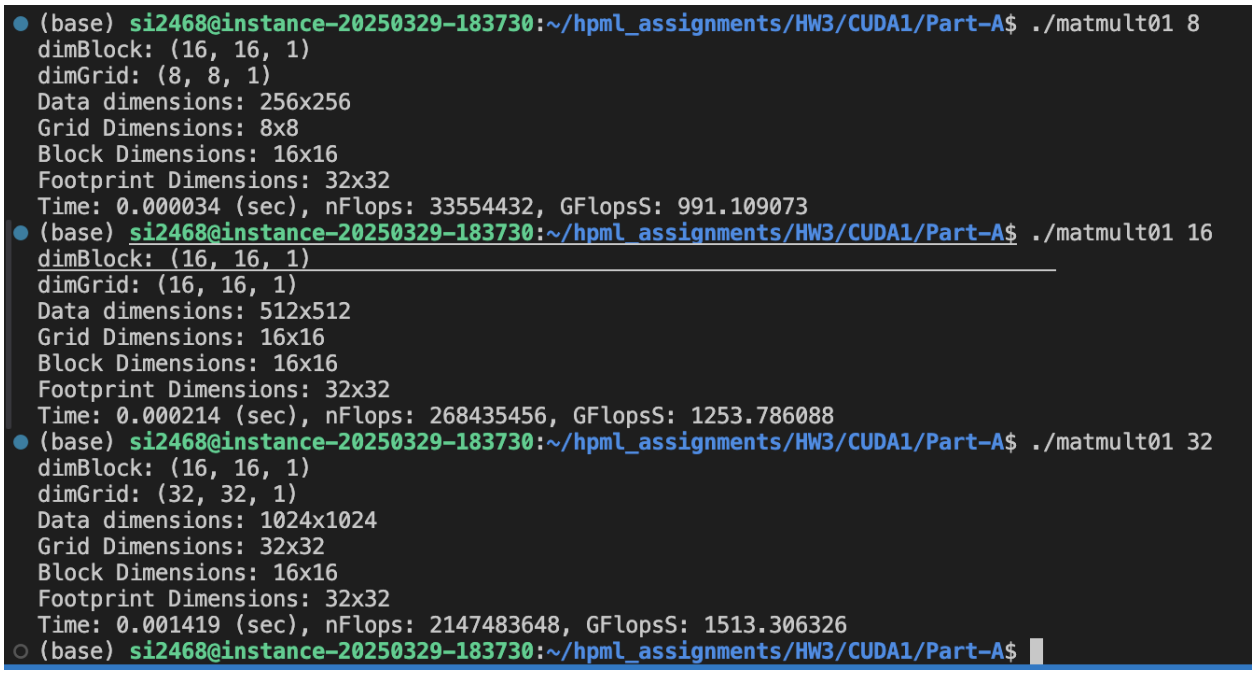
times as fast as the corresponding runs in the previous experiment without coalescing, which is expected considering we are taking advantage of DRAM bursts. We also see that the throughput and bandwidth utilization also roughly triples when we are able to use coalesced memory accesses.

Q3:



In this experiment, we are performing matrix multiplication on an input matrix, and we are allowing the user to define the dimensions of the grid in the granularity of blocks. The dimensions of each block are always set to 16 x 16. Thus, in our three calls, we have a grid of 16x16 blocks, 32x32 blocks, and 64x64 blocks, where each block has 256 threads. Thus, we are quadrupling the number of threads deployed in consecutive program calls. Each thread processes one output element. We see that the time also roughly quadruples with each successive experiment.

Q4:



In this experiment, we are performing matrix multiplication on an input matrix, and we are allowing the user to define the dimensions of the grid in the granularity of blocks. The dimensions of each block are always set to 16 x 16. Thus, in our three calls, we have a grid of 8x8 blocks, 16x16 blocks, and 32x32 blocks, where each block has 256 threads. Thus, we are quadrupling the number of threads deployed in consecutive program calls. Each thread processes 4 output elements. In this case, the time for each successive experiment increases (from 256x256 to 512x512 is almost 7 times as slow, and from 512x512 to 1024x1024 is about 7 times as slow).

Comparing this to the previous experiment, we can also get an idea of how thread coarsening affects performance (the number of output elements each threadblock/thread is responsible for computing). By allowing each thread to compute more values, and not just be responsible for computing one element, we reduce the overhead of launching/managing too many threads and potentially increase arithmetic intensity and occupancy. In our experiments, for sizes (256x256), (512x512), and (1024x1024), the speedups for the program with 4 output elements per thread are about 1.4x, 2.23x, and 2.51x. These are all significant decreases in execution time. It makes sense that our kernel is faster because in a workload like matrix multiplication, there is a lot of data reuse that can happen. This is improved by thread coarsening because the data is stored in registers instead of other memory types that will take longer to access. This also suggests that there are diminishing returns of speedup as we increase our matrix size, just from our small sample size.

Q5:

Can you formulate any rules of thumb for obtaining good performance when using CUDA?

The most important things to consider for obtaining good performance in CUDA are probably the following:

1, Try to use more threads per block so that we can increase GPU core utilization.
2. Don't use so many threads that the GPU does not have the resources to feed every one of them -> there is limited shared memory and register space.
3. Maximize memory coelescing to reduce the effect of global memory accesses.
4. Use shared memory if we would otherwise constantly use global memory because shared memory access time is much faster than global memory access time.
5. Avoid warp divergence because the warp will split into two paths, and will no longer be able to execute in parallel.

Part B:

Q1:

```
(base) si2468@instance-20250329-183730:~/hpml_assignments/HW3/CUDA1/Part-B$ ./array_add_cpu 1
Array size: 1000000 took 0.00176422 seconds.
(base) si2468@instance-20250329-183730:~/hpml_assignments/HW3/CUDA1/Part-B$ ./array_add_cpu 5
Array size: 5000000 took 0.00709667 seconds.
(base) si2468@instance-20250329-183730:~/hpml_assignments/HW3/CUDA1/Part-B$ ./array_add_cpu 10
Array size: 10000000 took 0.0150437 seconds.
(base) si2468@instance-20250329-183730:~/hpml_assignments/HW3/CUDA1/Part-B$ ./array_add_cpu 50
Array size: 50000000 took 0.0723035 seconds.
(base) si2468@instance-20250329-183730:~/hpml_assignments/HW3/CUDA1/Part-B$ ./array_add_cpu 100
Array size: 100000000 took 0.145249 seconds.
(base) si2468@instance-20250329-183730:~/hpml_assignments/HW3/CUDA1/Part-B$
```

In this experiment, we are performing vector addition on the CPU. We allow the user to input the vector size in millions. We also time the operation itself. The timing makes sense, as the time it takes seems to scale roughly linearly with the size of the vector.

Q2:

```
(base) si2468@instance-20250329-183730:~/hpml_assignments/HW3/CUDA1/Part-B$ ./array_add_gpu_non_unified 1
1 block, 1 thread per block: 0.0923722 seconds.
Array addition is correct!
1 block, 256 threads per block: 0.00169802 seconds.
Array addition is correct!
3907 blocks, 256 threads per block: 6.29425e-05 seconds.
Array addition is correct!
(base) si2468@instance-20250329-183730:~/hpml_assignments/HW3/CUDA1/Part-B$ ./array_add_gpu_non_unified 5
1 block, 1 thread per block: 0.359234 seconds.
Array addition is correct!
1 block, 256 threads per block: 0.006814 seconds.
Array addition is correct!
19532 blocks, 256 threads per block: 0.000255108 seconds.
Array addition is correct!
(base) si2468@instance-20250329-183730:~/hpml_assignments/HW3/CUDA1/Part-B$ ./array_add_gpu_non_unified 10
1 block, 1 thread per block: 0.750198 seconds.
Array addition is correct!
1 block, 256 threads per block: 0.0135269 seconds.
Array addition is correct!
39063 blocks, 256 threads per block: 0.000494003 seconds.
Array addition is correct!
(base) si2468@instance-20250329-183730:~/hpml_assignments/HW3/CUDA1/Part-B$ ./array_add_gpu_non_unified 50
1 block, 1 thread per block: 3.42735 seconds.
Array addition is correct!
1 block, 256 threads per block: 0.0675011 seconds.
Array addition is correct!
195313 blocks, 256 threads per block: 0.00236893 seconds.
Array addition is correct!
(base) si2468@instance-20250329-183730:~/hpml_assignments/HW3/CUDA1/Part-B$ ./array_add_gpu_non_unified 100
1 block, 1 thread per block: 6.75108 seconds.
Array addition is correct!
1 block, 256 threads per block: 0.135606 seconds.
Array addition is correct!
390625 blocks, 256 threads per block: 0.00471091 seconds.
Array addition is correct!
(base) si2468@instance-20250329-183730:~/hpml_assignments/HW3/CUDA1/Part-B$
```

In this experiment, we are performing vector addition on the GPU. We allow the user to input the vector size in millions. We also time the operations. For each experiment, we consider three scenarios, each representing a different thread/block configuration. The three scenarios are (in <<<blocks, threads_per_block>>> format: <<<1, 1>>>, <<<1, 256>>>, <<<input_size / 256, 256>>>.

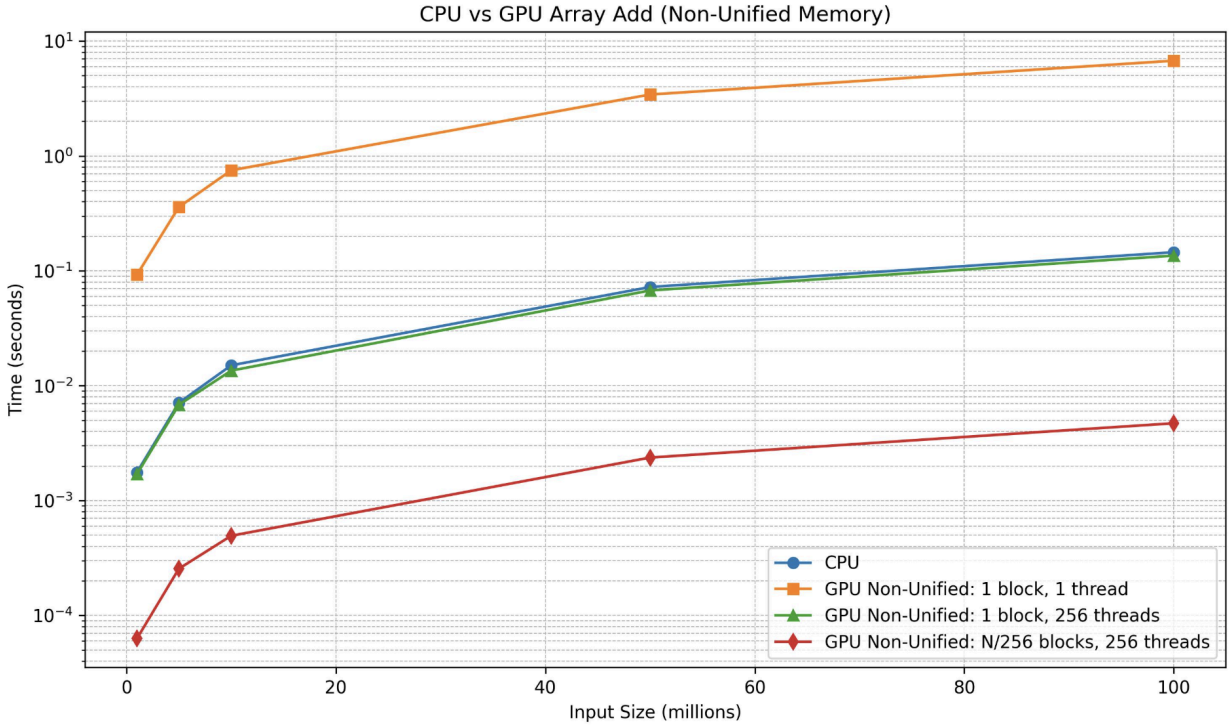
We see that the time to complete the operation increases for each experiment as we proceed through the scenarios, and this makes sense because we are increasing the number of threads, which allows for more parallelism. We also see that as we increase the size of the array, we see a roughly linear increase in the time it takes to complete the program, which makes sense since each thread's workload increase is also linearly increasing.

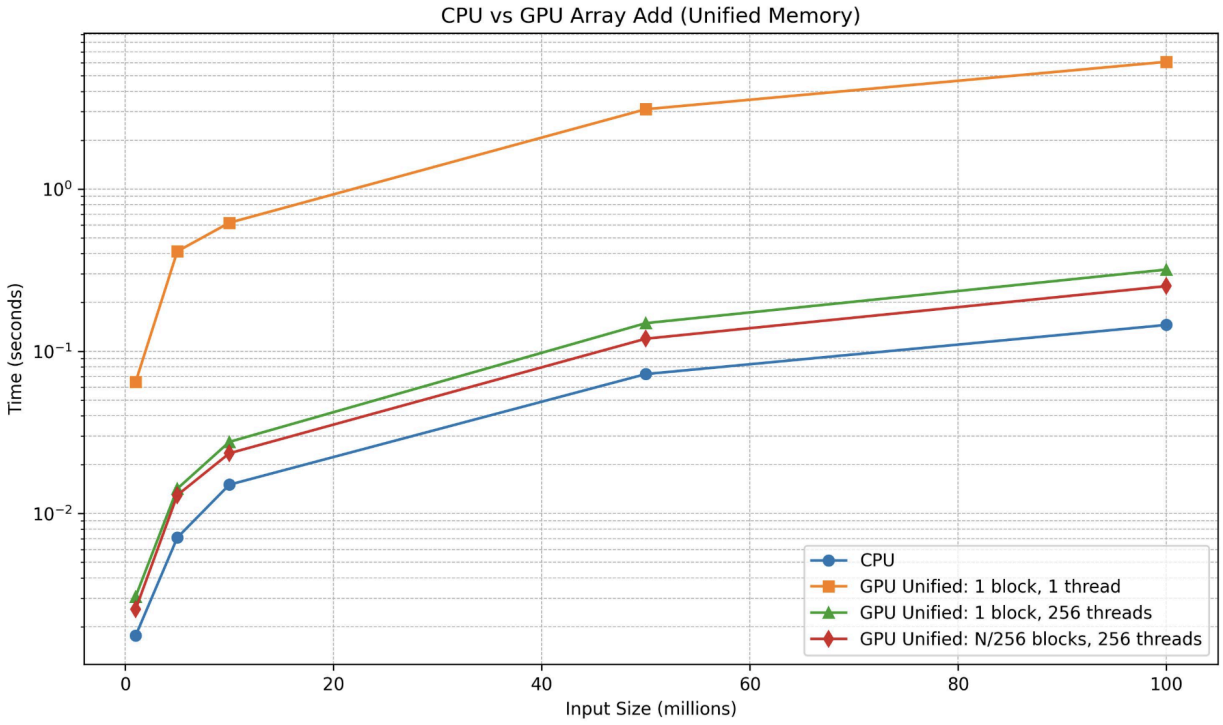
Q3:

```
^[[A(base) si2468@instance-20250329-183730:~/hpml_assignments/HW3/CUDA1/Part-B$ ./array_add_gpu_unified 1
1 block, 1 thread per block: 0.0646691 seconds.
Array addition is correct!
1 block, 256 threads per block: 0.00307393 seconds.
Array addition is correct!
3907 blocks, 256 threads per block: 0.00256801 seconds.
Array addition is correct!
(base) si2468@instance-20250329-183730:~/hpml_assignments/HW3/CUDA1/Part-B$ ./array_add_gpu_unified 5
1 block, 1 thread per block: 0.41271 seconds.
Array addition is correct!
1 block, 256 threads per block: 0.0142119 seconds.
Array addition is correct!
19532 blocks, 256 threads per block: 0.012964 seconds.
Array addition is correct!
(base) si2468@instance-20250329-183730:~/hpml_assignments/HW3/CUDA1/Part-B$ ./array_add_gpu_unified 10
1 block, 1 thread per block: 0.619475 seconds.
Array addition is correct!
1 block, 256 threads per block: 0.027621 seconds.
Array addition is correct!
39063 blocks, 256 threads per block: 0.0234928 seconds.
Array addition is correct!
(base) si2468@instance-20250329-183730:~/hpml_assignments/HW3/CUDA1/Part-B$ ./array_add_gpu_unified 50
1 block, 1 thread per block: 3.1036 seconds.
Array addition is correct!
1 block, 256 threads per block: 0.148956 seconds.
Array addition is correct!
195313 blocks, 256 threads per block: 0.119468 seconds.
Array addition is correct!
(base) si2468@instance-20250329-183730:~/hpml_assignments/HW3/CUDA1/Part-B$ ./array_add_gpu_unified 100
1 block, 1 thread per block: 6.08235 seconds.
Array addition is correct!
1 block, 256 threads per block: 0.318715 seconds.
Array addition is correct!
390625 blocks, 256 threads per block: 0.252037 seconds.
Array addition is correct!
```

This experiment is the same as the previous experiment, except that we use Unified Memory instead. Unified memory provides convenience to the developer, as we don't have to perform explicit memcopies. However, it does come at a performance cost due to page-faulting mechanisms to communicate memory between the CPU and GPU. We see this in the experiment above, as this is much slower than it's non-unified memory counterpart.

Q4 - Plotting





These graphs can be used to further analyze performance. From the first graph on non-unified memory, we can see that the CPU implementation performs just about as well as a GPU implementation with 1 block and 256 threads. This is probably because CPUs don't need

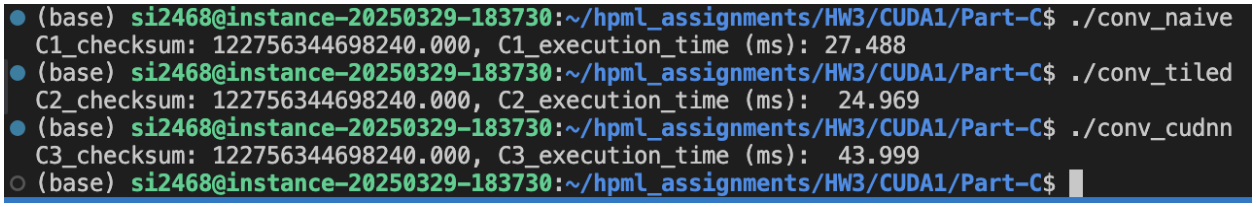
massive amounts of threads to achieve good performance, because they have very advanced hardware mechanisms and caching protocols that can improve performance already, as opposed to GPUs which do not have these in place and solely rely on massive parallelism to get throughput. As expected, the GPU experiment with 1 block and 1 thread performs the worst, significantly, because the GPU lacks those advanced CPU blocks and we are not taking advantage of massive multithreading. As expected, the GPU experiment with multiple blocks and multiple threads performs the best, although not as fast as one might expect, since there are GPU hardware limitations and occupancy issues.

From the second graph on unified memory, we can see a somewhat similar trend - the GPU experiment with 1 thread and 1 block performs significantly slower than every other run. However, this time, the CPU performance beats all the GPU implementations. This indicates that even with the massive amount of multithreading available with GPUs, the cost of using unified memory outweighs the benefits associated with having hundreds of threads working in parallel.

Looking across both graphs and comparing the same scenario (CPU, GPU with 1 thread 1 block, etc.), we can directly compare the effect of using unified memory. Interestingly enough, for the scenario with 1 thread and 1 block, unified memory performs a little better than non-unified memory. For a larger grid (take the scenario with the most threads deployed, for example), the unified memory times are about 50 times as large as the non-unified memory times. This indicates that unified memory overhead increases as we increase the number of threads. This makes sense because the more threads we have, the more likely we are to have concurrent memory accesses, which means we are more likely to incur page faults and therefore have to go through the expensive process of transferring memory.

Part C:

C1, C2, C3 (one screenshot):



From the performance here, we can see that the conv_tiled and conv_naive implementations are about the same, while conv_cudnn is significantly higher. Although conv_cudnn is an optimized library for performing convolutions, it will inherently suffer from some overhead of function calls and using the actual library rather than doing it directly. Thus, its true value can probably only be seen when performing larger convolutions, perhaps with larger filters or larger

inputs. Our tiled convolution barely beats our naive convolution, and it is possible that that's because of the same reason - the size of the input isn't big enough to really see the slowing effect of repeated global memory accesses (which the naive implementation has). Whereas with shared memory and tiling, we reduce global memory accesses by putting elements into shared memory and reading from it, where each memory access has shorter latency. We can hypothesize that for larger-scale computations, these differences would be more drastic.

C4:

```
    grid = (input_width, input_height, output_channels)

    # TODO: Call the triton kernel (my_triton_kernel) and measure execution time

    start = time.time()
    my_triton_kernel[grid](
        input_channels, input_height, input_width, output_channels, padding,filter_height
        BLOCK_H=1, BLOCK_W=1
    )
    # synchronize to make sure kernel is done
    torch.cuda.synchronize()
    end = time.time()
    exec_time_ms = (end - start)

    # TODO: Return output (output should include execution time)

    return output, exec_time_ms
```

[20]

```
# Testing
# Comparing the result from my_conv2d and Conv from torch
my_output, execution_time = my_conv2d(tensor_I, tensor_F)
torch.testing.assert_close(golden_out, my_output) # Assert statement should be passed
# Printing the execution time
print(f"Execution Time for triton kernel (ms): {execution_time * 1000:.3f}")
```

[21]

```
Execution Time for triton kernel (ms): 1031.304
```

The triton kernel finishes the convolution in 1031.304 ms. This is much slower than the CUDA kernels that do the same thing. This is most likely because triton is a Python API, so it is impossible for it to achieve the same speed as a fully C/C++/CUDA pipeline. However, it provides a much easier interface and a much easier development process.