Assignment III: Advanced CUDA

Group 12 - Chao Xiong, Chang Fu

Repository link: https://github.com/Kevin-XiongC/DD2360

Exercise 1 - CUDA Edge Detector using shared memory

Explain how the mapping of GPU thread and thread blocks (which is already implemented for you in the code) is working.

The basic idea is to process one pixel per thread, and we use a 2-dimensional grid and block to map each thread to each pixel. The mapping is done by the following code:

```
int index_x = blockIdx.x * blockDim.x + threadIdx.x;

int index_y = blockIdx.y * blockDim.y + threadIdx.y;
```

For those threads which are mapped to the outside of the image, we simply ignore them and assign no work to them.

Explain why shared memory can (theoretically) improve performance.

Shared memory resides in each thread block, which features low access latency. All threads in the block have access to the memory much quicker than access to the global memory. By moving frequently accessed data to shared memory decreases data access latency and improves performance.

Explain why the resulting image looks like a "grid" when the kernel is simply copying in pixels to the shared block. Explain how this is solved and what are the cases.

The reason for the “grid” comes from our convolution operation, where we use a 3 by 3 convolution kernel. This means that for a specific pixel, we need the values of all the pixels around that pixel to perform convolution, which is problematic for pixels lying in the rightmost and bottom. For instance, to calculate the convolution of the pixel in 16-th row and 16-th column, i.e. p[16][16], we need the value of pixels p[16][17], p[17][16] and p[17][17], which are all missing by simply copying the the 16 by 16 image window to our shared memory.

To solve this problem, we just need to copy the missing pixels to our shared memory. In this case, since we use a 3 by 3 convolution kernel, the missing part will be the 17-th and 18-th rows and columns, which is the reason why we use a 18 by 18 shared memory.

There are several images of different sizes in the image folder. Try running the program on them and report how their execution time relates to file sizes.

File	Size / MB	Step #1 / ms	Step #2 / ms	Step #3 / ms
hw3.bmp	24.9	6.321	6.737	7.585
rome.bmp	24.4	6.536	8.795	8.362
hk.bmp	39.8	9.820	11.207	10.161
nyc.bmp	72.0	17.854	17.667	19.112

** The file size is recorded on MacOS. The program is tested on Google Colab with GPU Tesla K80.*

The running time is approximately linear with the file size.

Exercise 2 - Pinned and Managed Memory

EX2 and EX3 were conducted on Google colab with Tesla V100 and cuda 9.0.

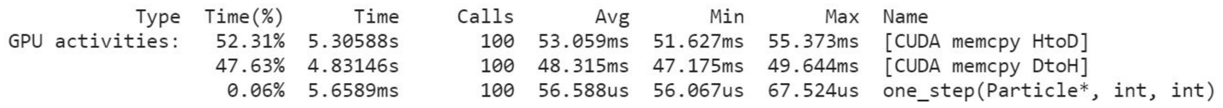
What are the differences between pageable memory and pinned memory, what are the tradeoffs?

Pinned memory allows us to directly transfer data from host to device without page faults, which benefits the performance. However, too large pinned memory may block other OS-level threads and affect the progress of the whole system while pageable memory would not affect OS.

Do you see any difference in terms of break down of execution time after changing to pinned memory from pageable memory?

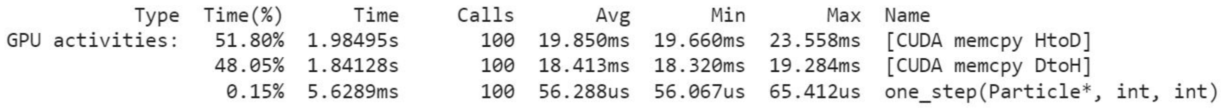
10M particles and 100 iterations.

Pageable:



```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   52.31%  5.30588s       100  53.059ms  51.627ms  55.373ms  [CUDA memcpy HtoD]
                   47.63%  4.83146s       100  48.315ms  47.175ms  49.644ms  [CUDA memcpy DtoH]
                    0.06%  5.6589ms       100  56.588us  56.067us  67.524us  one_step(Particle*, int, int)
```

Pinned:



```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   51.80%  1.98495s       100  19.850ms  19.660ms  23.558ms  [CUDA memcpy HtoD]
                   48.05%  1.84128s       100  18.413ms  18.320ms  19.284ms  [CUDA memcpy DtoH]
                    0.15%  5.6289ms       100  56.288us  56.067us  65.412us  one_step(Particle*, int, int)
```

From the above, we can see the data transfer has been improved a lot.

What is a managed memory? What are the implications of using managed memory?

Managed memory is shared memory between CPU and GPU. Using managed memory may cause many page faults because all threads can have access to the same memory and there surely will be many modifications on their local page tables.

If you are using Tegner or lab computers, the use of managed memory will result in an implicit memory copy before CUDA kernel launch. Why is that?

Because K80 is based-on Kepler architecture which is older than Pascal. For pre-Pascal GPUs, the cudaMallocManged() first allocate memory on GPU, and upon launching a kernel, the CUDA runtime must migrate all pages previously migrated to host memory back to the GPU memory.

Exercise 3 - CUDA Streams / Asynchronous Copy - Particle Batching

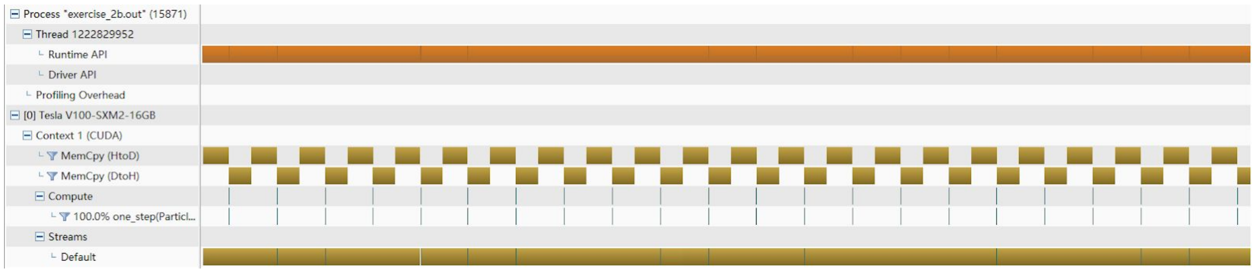
What are the advantages of using CUDA streams and asynchronous memory copies?

For most of the time, the data transfer should be the most time-consuming part. And with streams asynchronous memory copies, these two parts can proceed in parallel, which boosts the performance.

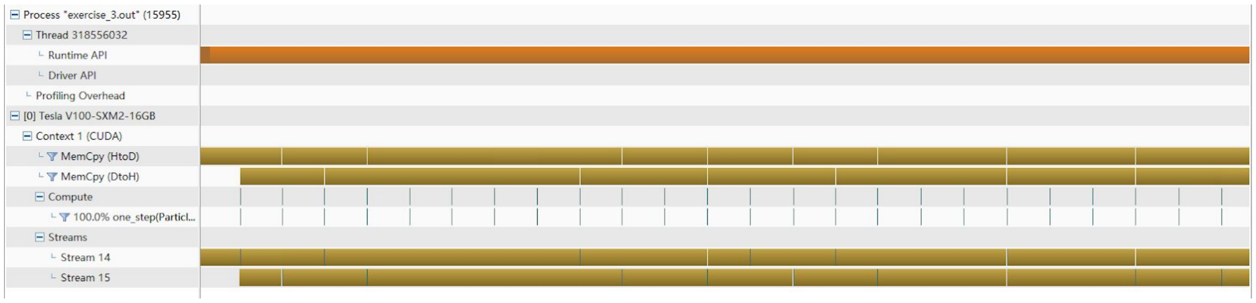
What is the performance improvement (if any) in using more than one CUDA stream?

EX2(synchronous, Without batches)	2 steams	4 streams
`4.049895s`	`2.598656s`	`2.648858s`

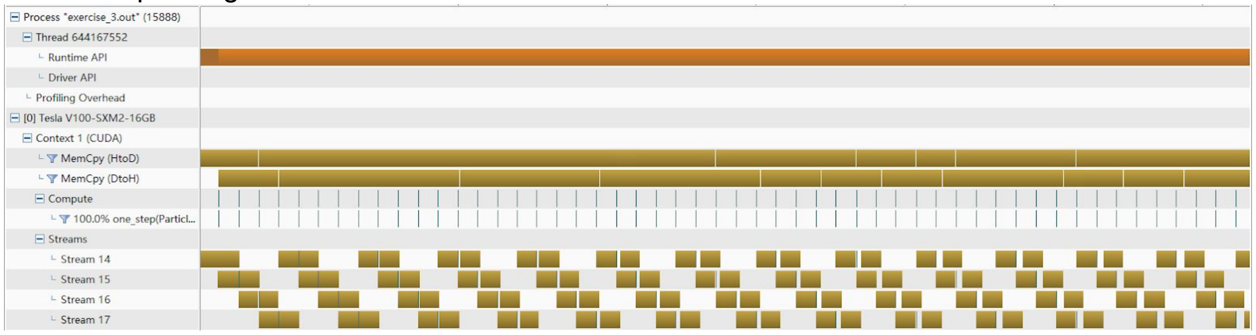
Default stream profiling:



2 streams profiling:



4 streams profiling:



Because V100 only has two copy engines, with two streams, the data transfers (HtoD and DtoH) completely overlap with each other while with 4 streams, streams have to be blocked waiting for some previous stream to release the control of engines. And there will only be extra costs rather than improvements with more streams.

What is the impact of batch size on the performance?

In our implementation, the batch size is defined by NUM_PARTICLES/NUM_STREAMS. In general,dividing the whole data into small batches can overlap but there are overheads for launching a single data transfer and kernel and thus we should batch small transfers.

Bonus Exercise - CUDA Libraries - cuBLAS

Explain why the matrix size has to be a multiple of 16?

To perform tiled matrix multiplication, we need to partition the matrix into blocks with the same size. If the matrix size is not a multiple of 16, we may have the risk of accessing uninitialized data. Basically this is for the simplicity of our calculation.

Refer to **shared_sgemm_kernel()**. There are two **__syncthreads()** in the loop. What are they used for, in the context of this code?

1. What is the directive that can potentially improve performance in the actual multiplication? What does it do?
2. There is a large speedup after switching from using global memory to shared memory, compared to the Edge Detector in Exercise 1. What might be the reason?

The first **__syncthreads()** is used to ensure that all threads in the block have finished copying data to the shared memory before it’s used. The second **__syncthreads()** is used to ensure that the matrix multiplication has finished and the shared memory is ready to be flushed with new data.

The pragma directive we use is

```
#pragma loop(hint_parallel(TILE_SIZE))
```

This is a hint to the compiler that the loop should be paralleled across TILE_SIZE threads, and we can speed up the multiplication if we have more threads to do this for loop.

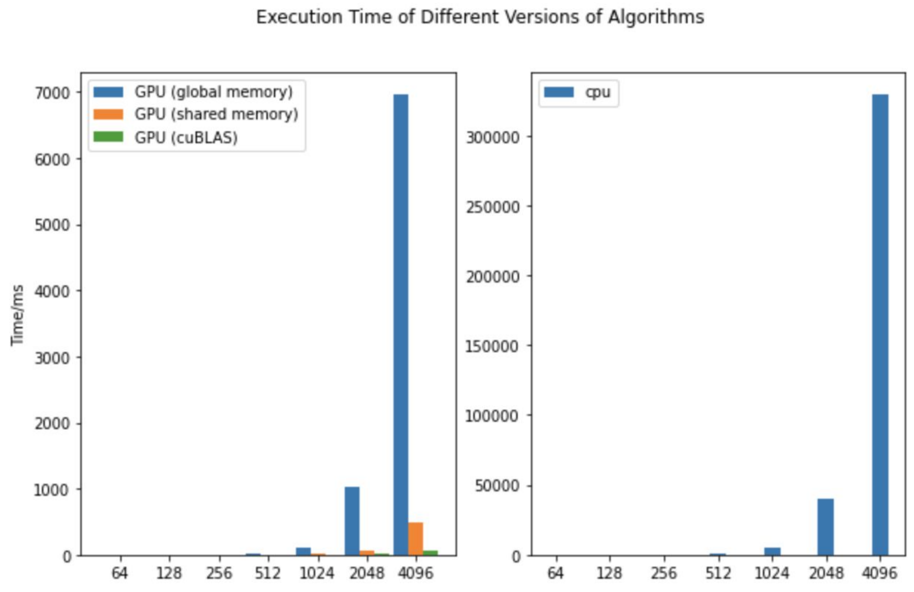
The reason for the large speedup after using shared memory might be the frequent access to data in this problem. Not like Exercise 1 where the pixels are accessed only a few times, in this problem we need frequent access to the values of each matrix, where taking advantage of shared memory gives us a boost of performance.

Refer to **cublas_sgemm()**. We asked that you compute **C = BA** instead of **C = AB**. It has to do with an important property of cuBLAS. What is that, and why do we do **C = BA**?

In cuBLAS, matrices are stored in column-major format, while in normal C programs in row-major format, and cuBLAS provides several ways to manipulate matrices. Parameter CUBLAS_OP_N means no operation to the input matrix, and CUBLAS_OP_T means doing transposition to the input matrix. Since in our program we specify no operation, and matrices are stored in row-major format, we need to do **C = BA** rather **C = AB** to get the correct result.

Run the program with different input sizes, for example from 64, 128, ... , to 4096. Make a grouped bar plot of the execution times of the different versions (CPU, GPU Global, GPU Shared, GPU cuBLAS). You can plot CPU results in a separate figure if the execution time goes out of the scale comparing to the rest.

From the plots we can see that the CPU takes much longer time to compute the matrix multiplication, especially when the matrix is pretty large. On the other hand, GPU with global memory performs much better than the CPU, but still takes a rather long time when the matrix is large. GPU with cuBLAS performs best, especially when the size of the matrix is large.



The way the execution time benchmark that is implemented in the code is good enough for this exercise, but in general it is not a good way to do a benchmark. Why?

There are several reasons for this. First, the implementation of this kind of benchmark is trivial, and we have more convenient and accurate tools to measure the execution time - nvprof. Moreover, the execution time is not the only factor to test a program. Other factors such as energy consumption and space occupation also serve as important benchmarks.