EE595 Lab 1. CUDA Programming

Yassawe Kainolda (20214229)

All results presented were obtained on NVIDIA GTX1070 (Haedong Lounge).

1. Saxpy

```
Running 3 GPU timing tests:
Effective BW by CUDA saxpy: 211.459 ms          [5.285 GB/s]
Time taken to run the kernel: 6.006 ms.
Effective BW by CUDA saxpy: 194.528 ms          [5.745 GB/s]
Time taken to run the kernel: 5.983 ms.
Effective BW by CUDA saxpy: 191.774 ms          [5.828 GB/s]
Time taken to run the kernel: 5.988 ms.

Running 3 CPU timing tests:
CPU elapsed time: 59.567 ms.
CPU elapsed time: 59.591 ms.
CPU elapsed time: 59.615 ms.
```

Figure 1. Result of ./cudaSaxpy

Question 1:

We observe that the time taken to run the saxpy kernel is much smaller than the time taken to compute the sequential CPU saxpy (~x10 speedup). However, overall CUDA saxpy time is much larger. The reason for that is that the memory transfer is the bottleneck in this case. Memory transfer overhead should be an important consideration when we decide on parallelizing something.

GTX 1070 max theoretical memory bandwidth = 256 GB/s [1]
PCIe 3.0 x16 max theoretical transfer speed = 15.754 GB/s [2]
Average effective bandwidth observed = 5.619 GB/s

Since the transfer between host and device (and vise versa) is constrained by PCI transfer speed, we can see that our transfer is about 3 times slower than the theoretical maximum.

2. Matrix Mul

```
- MatrixA(320,320), MatrixB(640,320)

[Step-1] Computing reference result using host-side CPU ... DONE!

[Step-2] Computing result using naive version of CUDA kernel ... DONE!
- Math Size = 131072000 OPs
- Performance = 348.75 GFLOP/sec (Time = 0.376 msec)
- Correctness (reference vs. CUDA): PASS

[Step-3] Computing result using shmem version of CUDA kernel ... DONE!
- Math Size = 131072000 OPs
- Performance = 828.03 GFLOP/sec (Time = 0.158 msec)
- Correctness (reference vs. CUDA): PASS
```

Figure 2. Performance of my implementation.

```
- MatrixA(320,320), MatrixB(640,320)

[Step-1] Computing reference result using host-side CPU ... DONE!

[Step-2] Computing result using naive version of CUDA kernel ... DONE!
- Math Size = 131072000 OPs
- Performance = 344.08 GFLOP/sec (Time = 0.381 msec)
- Correctness (reference vs. CUDA): PASS

[Step-3] Computing result using shmem version of CUDA kernel ... DONE!
- Math Size = 131072000 OPs
- Performance = 872.84 GFLOP/sec (Time = 0.150 msec)
- Correctness (reference vs. CUDA): PASS
```

Figure 3. Performance of ./matrixMul_reference

Question 2:

1) My shared memory implementation of matrix multiplication is about 2.4 times faster than the naive approach. The reason for that is that 1) shared memory access is much faster than global memory access that happens in naive implementation at every k.

2) My naive and shared memory matrix multiplications show approximately the same performance as the reference (within 5% bounds). My naive matrix multiplication slightly outperforms the one given in the reference.

3) Maximum theoretical single-precision throughput on Nvidia GTX 1070 (assuming base clock) is 5783.04 GFLOP/S [1]. In my implementation, GFLOP/S is much smaller than that number. This is partially due to the fact that in both naive and shared memory versions, we inevitably process some operations sequentially.

3. Circle Renderer

```
------------
Score table:
------------
-------------------------------------------------------------------------
| Scene Name      | Ref Time (T_ref) | Your Time (T)   | Score           |
-------------------------------------------------------------------------
| rgb             | 0.2219           | 0.2603          | 12              |
| rand10k         | 3.3792           | 4.4087          | 10              |
| rand100k        | 33.2858          | 43.4230         | 10              |
| pattern         | 0.4037           | 0.4136          | 12              |
| snowsingle      | 22.6648          | 10.7088         | 12              |
| biglittle       | 21.5954          | 50.3495         | 7               |
-------------------------------------------------------------------------
|                                  | Total score:    | 63/72           |
-------------------------------------------------------------------------
```

Figure 4. The score achieved by my implementation.

1) The first approach that I implemented was naive pixel parallelism. The image was divided into 32x32 blocks, within which each thread was responsible for a single pixel. Inside each thread, there was a for loop that iterated over the array of all circles and called a function to print them indiscriminately.

Since the provided *shadePixel()* function already performs a check on whether a circle contributes to a certain pixel or not, this approach worked. It guaranteed *atomicity* since each pixel is accessed only by one thread, and *order* since circles were iterated sequentially. This approach scored only 25/72.

2) The inefficiency of the naive algorithm is that each thread performs useless work by iterating over circles that are not even in the block's scope. Let the bounding box on an image be defined as a subset of 32x32 pixels processed by a single thread block. If a circle does not exist inside this box, no thread in the corresponding block should iterate over it.

It would be preferable to first check if a circle exists in the box, if yes then add its index to *validIdx* array, and then iterate only over those circles. However, this step will not offer any optimization, unless the check will happen in parallel. Since I have 32x32=1024 threads in a thread block, I can check 1024 circles in parallel per iteration. I further refer to the set of 1024 circles, that are processed concurrently, as a batch.

By checking circles in batches, I would reduce the time complexity of a single thread from O(n) to $O(m_i*n/1024)$, where m_i is the number of circles in the i^{th} batch that passed the check. Since the maximum value of m_i is 1024 (all circles in the batch are present in the box), in the worst-case scenario the thread time complexity is O(n), in all other cases, it offers optimization.

3) At each iteration, every thread is assigned a circle of index [*i+ThreadId*], where ThreadId is a 1D local index of the current thread (from 0 to 1023) and i is the batch number. Each thread then checks if its circle exists in the box. If yes, the circle index (relative to the batch number) is recorded in a shared memory array *tempIdx* at a *ThreadId* position. Note, that this preserves order.

tempIdx is now a sparse array of relevant circle indices which looks like:

[| 0 | 0 | 0 | 0 | index1| 0 | 0 | 0 | index2 | 0 | 0 | index3 | 0 | 0 | ...]

I then perform *stream compaction* (remove zeros) using *parallel exclusive scan* in order to populate *validIdx* shared memory array, which then would look like:

[| index1 | index2 | index3 | ...]

Now, this is a subset of circles that are present in the box corresponding to the current thread block. Each thread then prints them in order to their corresponding pixel. Atomicity is preserved since only one thread in the grid can access its pixel. Order is also preserved.

4) At every batch, three functions are invoked by every thread:

- The function that checks if a circle given to a thread exists in the box and writes *tempIdx*
- The *exclusive scan* function, which is needed for implementation.
- The function that populates *validIdx*

Thread synchronization happens after each of those functions. It also happens after pixel update, to ensure that the next batch won't start until all threads have written their pixels.

References:

[1] https://www.nvidia.com/en-in/geforce/products/10series/geforce-gtx-1070/
[2] https://web.archive.org/web/20140201172536/http://www.pcisig.com/news_room/faqs/pcie3.0_faq