Fall 2024 | CS8803 GPU | Project 2 Report | Jun Liang, Ho (jho88@gatech.edu)

Introduction and Summary of Approach

I made sure to get functional correctness from task 1 first by implementing the pseudocode given for it. However, I realized that I would need to amend the functions and split it up for task 2. I was rather confused about how bitonic sort worked, and spent some time to watch the YouTube videos recommended as well as to watch the HPC videos by Prof. Vuduc. Those were really helpful and helped to clarify my approach. Following that, I iteratively made improvements based on optimization ideas from Ed Discussions, the internet as well as various NVIDIA CUDA webpages.

Implementation and Optimizations Made

1. Use of pinned memory.

   I read about pinned memory on the host side, which could help speed up data transfers from host to device and device to host and hence made use of them. As we could not adjust the allocation, I made use of cudaHostRegister to register the memory. I hid the latency for this optimization behind the compute kernels on the device side. I also noticed that directly allocating arrCpu and arrSortedGpu to pinned memory gave performance improvements of 250MEPs to 400MEPs, which meant that cudaHostRegister did have significant latency – I subsequently tried to hide this registering behind other GPU computation and managed to reduce the latency.

2. Use of streams.

   I tried to have some overlapping in asynchronous calls for the copying of data from host to device by creating separate streams for the memory copy of the original array into device, and then the copying of the offset by the padding size into device, since there should be no dependency between them. I tried to split up the copying of d_arr into arrSortedGpu into streams, but there was no speed increase, and hence it was abandoned. Streams provided more control over how to overlap computation, as I read that the default stream is blocking across the entire device whereas custom streams are not.

3. Padding to power of 2.

   I explored various methods to optimize the computation of the next power of 2, including the use of mathematical functions like log and exponent, but eventually settled on bit shifting which I read was the most efficient. I also made sure that the cudaMemCpy was efficient and did not copy unnecessarily by setting the offset appropriately to separately copy arrCpu then the padding needed (and not to set to 0 across the whole created device array, then override with arrCpu).

4. Using UVA for created device array ('d_arr') – bad optimization, so not used.

   I tried to utilize this, but realized that this was a poor optimization because of the need to do 'memcpy' back into arrSortedGpu eventually, which was slower than cudaMemCpy. It would make sense if we were allowed to directly allocate to UVA for arrCpu and arrSortedGpu, but this was not the case. Hence, I did not end up using this optimization.

5. cudaMemSet to 0 instead of other methods.

I padded with 0s using cudaMemSet as that produced the best results compared to setting to infinity using other functions.

6. Bitshifting.

Similarly, I made use of bitshifting instead of * and / operators as I read that it is more efficient and faster. I made use of this wherever I could.

7. Shared memory and multiple kernels.

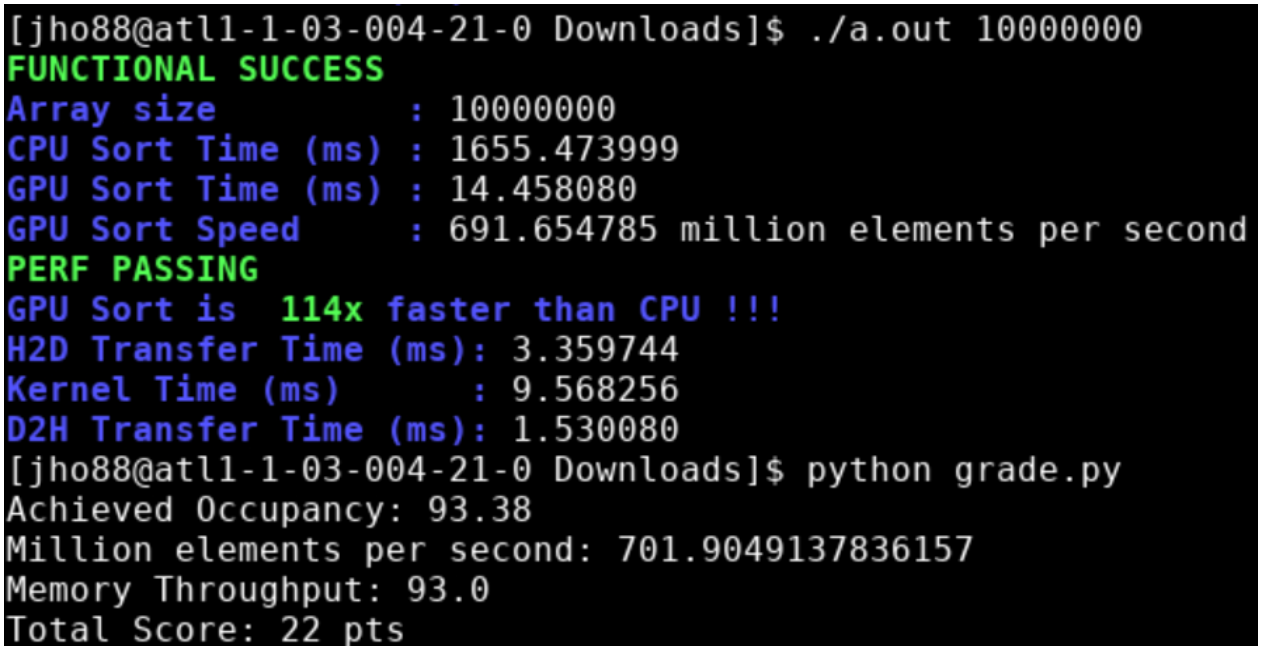
This was used to copy from global memory to shared memory, as shared memory is faster. However, shared memory is of smaller size so this was limited to those arrays or chunks that could fit within shared memory size. I tuned the shared memory limit after printing out the shared memory per chunk maximum using deviceProp, then tuned manually after that from there depending on which produced the best results (taking into account occupancy and throughput as well). I split the kernels up to separately handle those chunks that fit within shared memory and those that did not.

8. Interleaving of CPU and GPU functions.

As much as possible I tried to interleave these computations to make use of idle time and the fact that some computations have no dependencies with each other.

Results:

On an L40S NVIDIA GPU on PACE –



```
[jho88@atl1-1-03-004-21-0 Downloads]$ ./a.out 10000000
FUNCTIONAL SUCCESS
Array size           : 10000000
CPU Sort Time (ms)   : 1655.473999
GPU Sort Time (ms)   : 14.458080
GPU Sort Speed       : 691.654785 million elements per second
PERF PASSING
GPU Sort is  114x faster than CPU !!!
H2D Transfer Time (ms): 3.359744
Kernel Time (ms)        : 9.568256
D2H Transfer Time (ms): 1.530080
[jho88@atl1-1-03-004-21-0 Downloads]$ python grade.py
Achieved Occupancy: 93.38
Million elements per second: 701.9049137836157
Memory Throughput: 93.0
Total Score: 22 pts
```

Possible next steps

I would spend more time to better learn how to use the profiler and optimize based on that instead which would be more targeted.

References:

1. **https://developer.nvidia.com**
2. **https://github.com/NVIDIA-developer-blog/code-samples**