Assignment 4

Wenkai Li, Zehao Chen

1.Implementation

In preprocessing, we first allocate memory for input and output arrays on the GPU, and then copy the content of host input array to device input array. We also implemented a kernel to write the 4 center values to the device output array because center values will be skipped in the computation.

In the computation phase, we implemented a kernel such that each thread calculates one array element. A thread calculates the index of its responsible array element using block index and thread index. We fill a temporary array in shared memory with values from the input array to speed up calculation. We synchronize threads here to make sure all threads in the block have filled the shared array. We skip computation and return if the element is on the edge or in the center, otherwise, we calculate the element in the output array by averaging its neighbors in the temporary array. After each iteration, we swap of the pointer to device input and output array.

In postprocessing, we copy the device input array from back to host. This is because of the swap of pointers after each iteration.

Optimization

The first optimization is about tuning the number of threads per block. This affects occupancy of the Streaming Multiprocessors and thus performance. We first try to determine the specs and limitations of the V100 GPU on the izar server. We obtained our knowledge from the architecture whitepaper (see https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf).

As described in the whitepaper, the GV100 GPU has 84 SMs (The actual Tesla V100 seems to have only 80, but it does not affect the following calculation a lot). It supports 32 threads per warp, and max 64 warp or 32 thread blocks or 2048 threads per SM. This means that we can have at most 2048 * 84 = 172032 concurrent threads. However, with a side length of 1000, we will have 1000000 array elements. Therefore, we will hit a hard upper bound and not be able to do all the computation in one run, and this leaves little room for parameter tuning.

In our scenario, we can have max 64 warps or 32 thread blocks per SM. It means that having 2 warps, i.e. 64 threads per thread block would already achieve maximum occupancy. Therefore, we consider 64 threads per thread block to be optimal and set thread block dimension as (8, 8), and the grid has the side length of ceil(array_length/8). We did not find any comment on whether having a same sized but differently layed out thread block would affect performance, but we did not observe significant difference in performance.

The second optimization we tried is the shared memory. This is shared per thread block, uses the same architecture as cache, and thus is much faster than the global memory. Since we have (8, 8) thread blocks, we declare a shared array of 10 x 10 to cover neighbor elements. With 8x8 thread blocks, we would have 64 * 9 = 576 global memory access. With

the shared memory array, we only need 100 (a few more in the actual implementation to cover the corner elements more easily). This can give a lot of speed up as well.

The last optimization we considered is reducing thread divergence. When control flow leads to different paths for threads within a warp, performance can suffer. A natural idea would be to remove the conditional statements that determine if at the center of heatmap, and assign core values afterwards. This may potentially reduce some thread divergence and improve performance.

2. Following tables present the execution time in seconds of different thread organizations and optimization methods. In the parentheses are the breakdown GPU computation times.

We can see from table 1 that 64 threads per block is indeed the optimal thread organization, although 128 threads comes very close. Having 32 threads per block seems to experience more scheduling overhead for more thread blocks, while having 256 threads per block gives the worst result possibly due to lower occupancy.

From table 2, we can see that with the (1000, 10000) combination, GPU takes around only 5% of the time as does CPU baseline. Adding shared memory gives another 30% decrease in execution time. This is as we expected in Q1 because shared memory is much faster than global memory, and we do need to access memory a lot of times (9 access per element to be specific).

On top of shared memory, the attempt to reduce thread divergence did not work. It may be that the path that calculates the core elements is rarely accessed and thus does not really add to thread divergence.

Therefore, the optimal implementation is having 64 threads per thread block, with shared memory.

	64 Thrs/TB	32 Thrs/TB	128 Thrs/TB	256 Thrs/TB
(100, 10)	0.1319 (7.091e-05)	0.1219 (7.341e-05)	0.1110 (7.494e-05)	0.1151 (8.256e-05)
(100, 1000)	0.1084 (0.003021)	0.2205 (0.003096)	0.1901 (0.003113)	0.1883 (0.003819)
(1000, 100)	0.2221 (0.005031)	0.1423 (0.005905)	0.2115 (0.005035)	0.1369 (0.008678)
(1000, 10000)	0.6541 (0.4559)	0.6532 (0.5302)	0.5756 (0.46)	0.9059 (0.783)

Table 1 Execution time (s) with different thread organizations

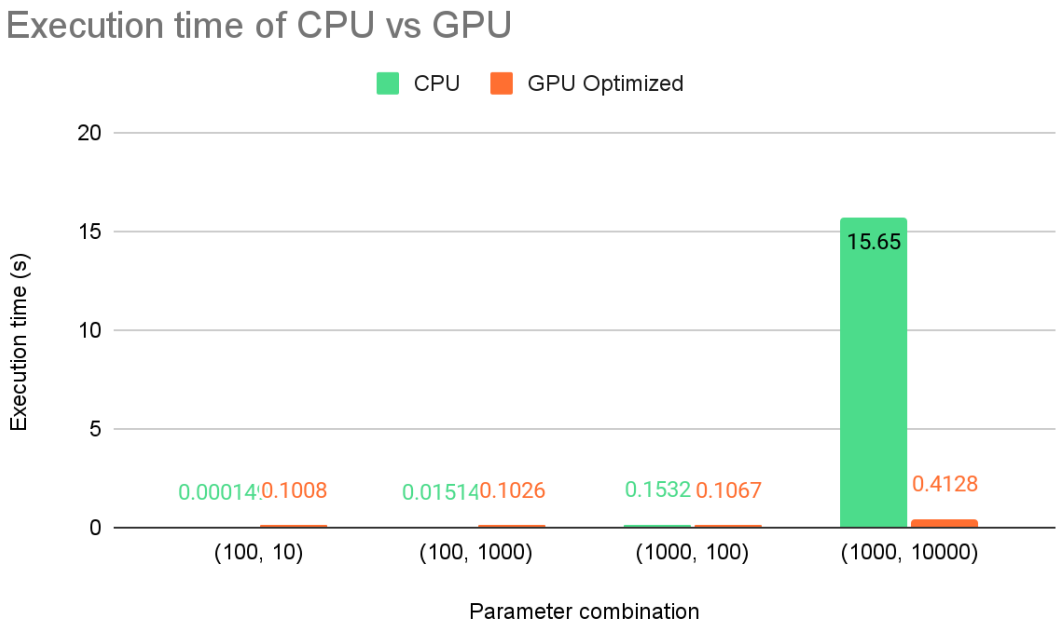
	CPU	64 Thrs/TB	+ shared memory	+ less thread divergence
(100, 10)	0.000149	0.1319 (7.091e-05)	0.1008 (8.211e-05)	0.1861 (9.805e-05)
(100, 1000)	0.01514	0.1084 (0.003021)	0.1026 (0.002743)	0.1101 (0.005323)
(1000, 100)	0.1532	0.2221 (0.005031)	0.1067 (0.003439)	0.1239 (0.003528)
(1000, 10000)	15.65	0.6541 (0.4559)	0.4128 (0.3104)	0.4415 (0.3254)

Table 2 Execution time (s) of baseline and with different optimizations

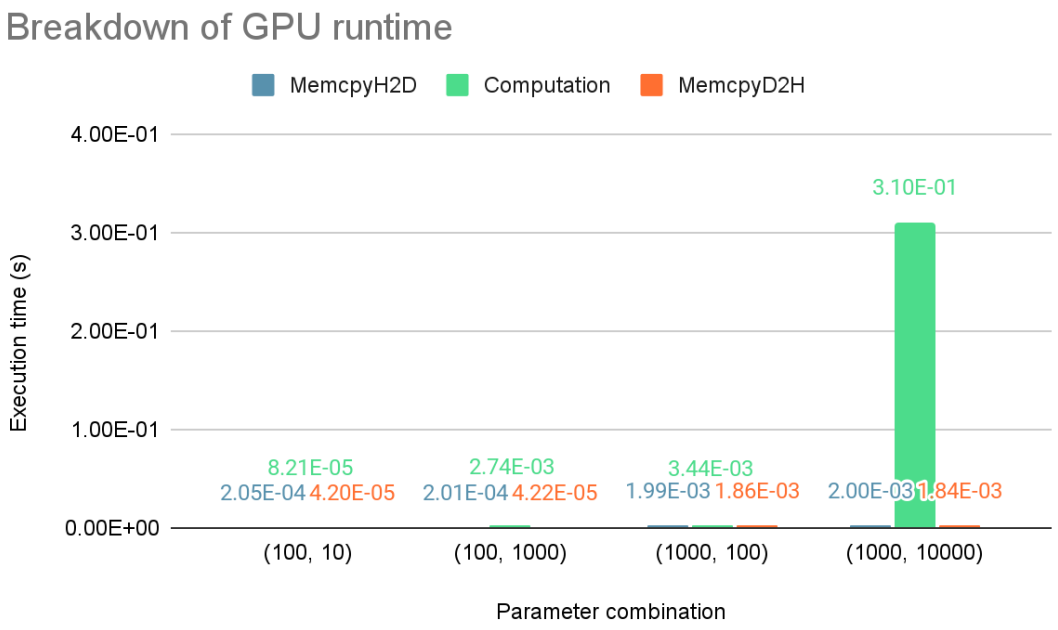
3.

	(100, 10)	(100, 1000)	(1000, 100)	(1000, 10000)
MemcpyH2D	0.0002047	0.0002007	0.001992	0.001996
Computation	8.211e-05	0.002743	0.003439	0.3104
MemcpyD2H	4.202e-05	4.224e-05	0.001858	0.001842

Table 3 Breakdown of GPU runtime of optimized implementation



Graph 1 Execution time of CPU baseline vs optimized GPU version



Graph 2 Breakdown of optimized GPU runtime

4. In the first graph, we can see that CPU actually outperforms GPU when the size of computation is relatively small. This is because GPU computation has more overhead in transferring data and thread initialization. But when computation is heavy and can be well parallelized, GPU can be multitudes faster than CPU.

In the second graph, we can see that data transfer time is linear to data size, and computation time is linear to the number of iterations. When the size of computation is small, transferring data can take more time than computation and it may not be the most efficient to use GPU. This partly explains the result in the first graph. When the size of computation gets large, the overhead will be negligible compared to computation time, and the parallel compute power of GPU will really start to show its strength.