CSCE-435 Homework #6

Andrew Kirfman

Due Date: 6/30/17

Question #1: Develop a CUDA-based parallel code to compute the distance between the closest pair of points on a GPU. Describe the changes that you made to the code.

In order to perform the computation of the closest point distance, I implemented the kernel function minimum_distance along with making some small changes to the main function. This kernel function is called with *n* threads, one thread per point (i.e. if the program is run with 2048 points, 2048 threads will be created). Threads are organized into blocks of 1024 threads each. The modifications that I made to main() to accomplish this are as follows. Note, block size is declared at the top of nbody.cu using a #define.

```
int num_blocks = num_points / block_size + ((num_points % block_size) != 0);

minimum_distance<<<num_blocks, block_size>>>(dVx, dVy, dmin_dist, num_points);
```

The kernel function performs the following operations:

- Compute the thread's global id with respect to all of the threads. This is calculated as follows: blockIdx.x * blockDim.x + threadIdx.x. This means that the thread in block 0, threadIdx = 0 will have an id of 0. Block 0, threadIdx of 1 will result in an id of 1, and so on. The global id defines which point this particular thread acts on (i.e. the thread with id 0 will act on point 0, and so on...).
- If a thread has an id that is larger than the number of points - 2, then it doesn't do anything. This if statement is necessary in the event that the number of points - 2 is not the same as the block size (i.e. there will be extra threads). There is no choice but to have these threads block until the rest of the threads are done.
- Inside of the if statement, each thread computes the distance between the point denoted by the thread's id and points [id + 1, num_points - 1]. For example, thread number 0 will compute the distance between point 0 and points 1 through n - 1. Thread number n - 2 will compute the distance between point n - 2 and point n - 1.
- Each thread then works to compute the minimum distance of all threads within its same block (since I can only synchronize threads on a block basis). This minimum is computed using the logarithmic reduction scheme described on Nvidia's developer website (Here: http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/ reduction/doc/reduction.pdf). This means that at this step, there are block_size reductions being performed concurrently. At the end of the reduction operation, the first thread in every block places the block's minimum value into D[blockIdx.x].
- While the threads are storing their block local minimums in the D array, one thread in each block is also incrementing a __device__ variable using the atomicInc function.
- The first thread in the last block to call atomicInc is the one which aggregates the final results. It does this by linearly iterating through the D array. The final global minimum is stored in D[0]. This variable is copied back to the host in the main function.

Note: See Question #2 for performance analysis of my code.

Question #2: Execute the code for $n = 2^k$ for k = 4, ..., 10. Plot GPU and CPU execution time verses k on the same plot to demonstrate how execution time varies with the problem size on these platforms. Use logarithmic scale for x-axis. Next, plot the GPU and CPU execution time for $n = 2^k$ for k = 11, ..., 16. For what value of n does the GPU code become faster than the CPU code?

Table #1: CPU vs. GPU Execution Times for $n = 2^k$, $k = 4, \cdots, 10$

# Points	GPU Runtime (ms)	CPU Runtime (ms)
16	0.077280	0.001239
32	0.062496	0.001705
64	0.088128	0.003675
128	0.132704	0.009710
256	0.224896	0.031227
512	0.405024	0.113245
1024	0.789536	0.436861

Figure #1: Data From Table #1

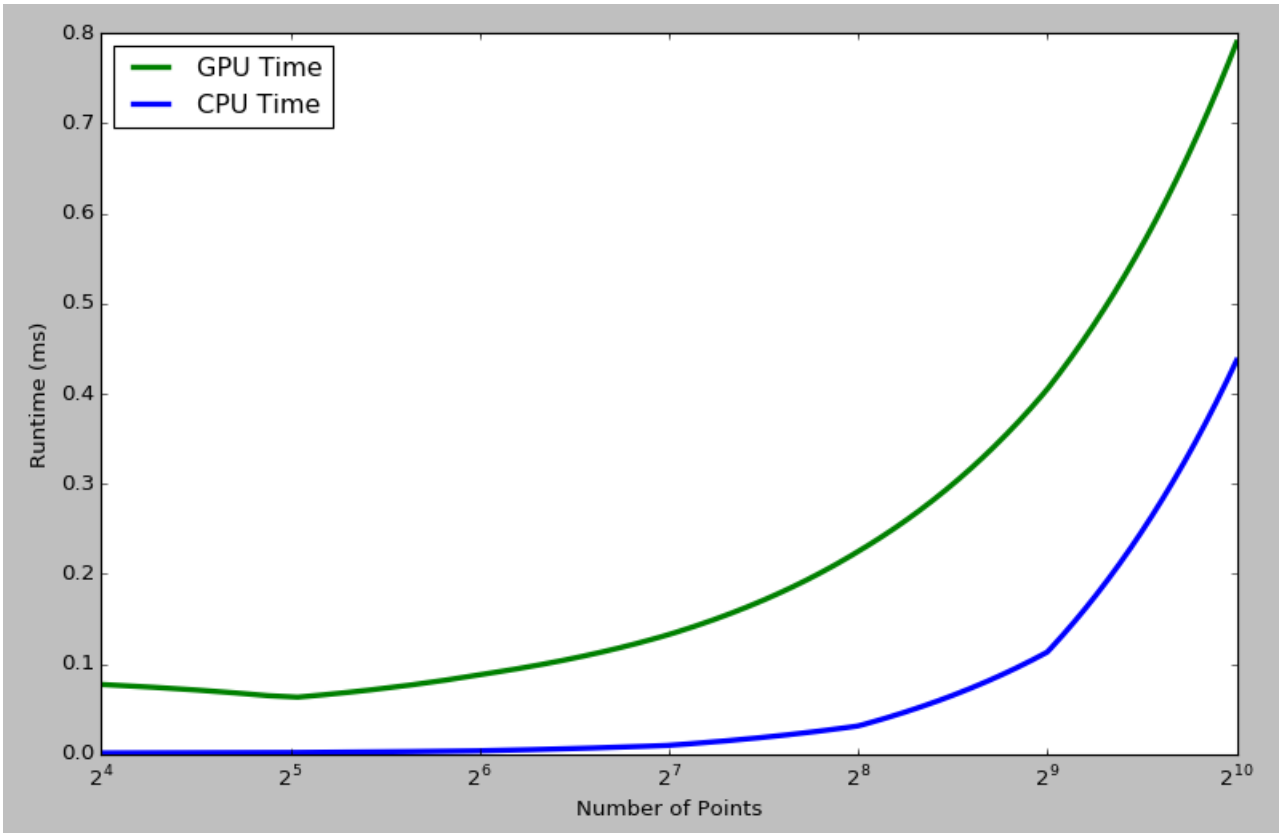
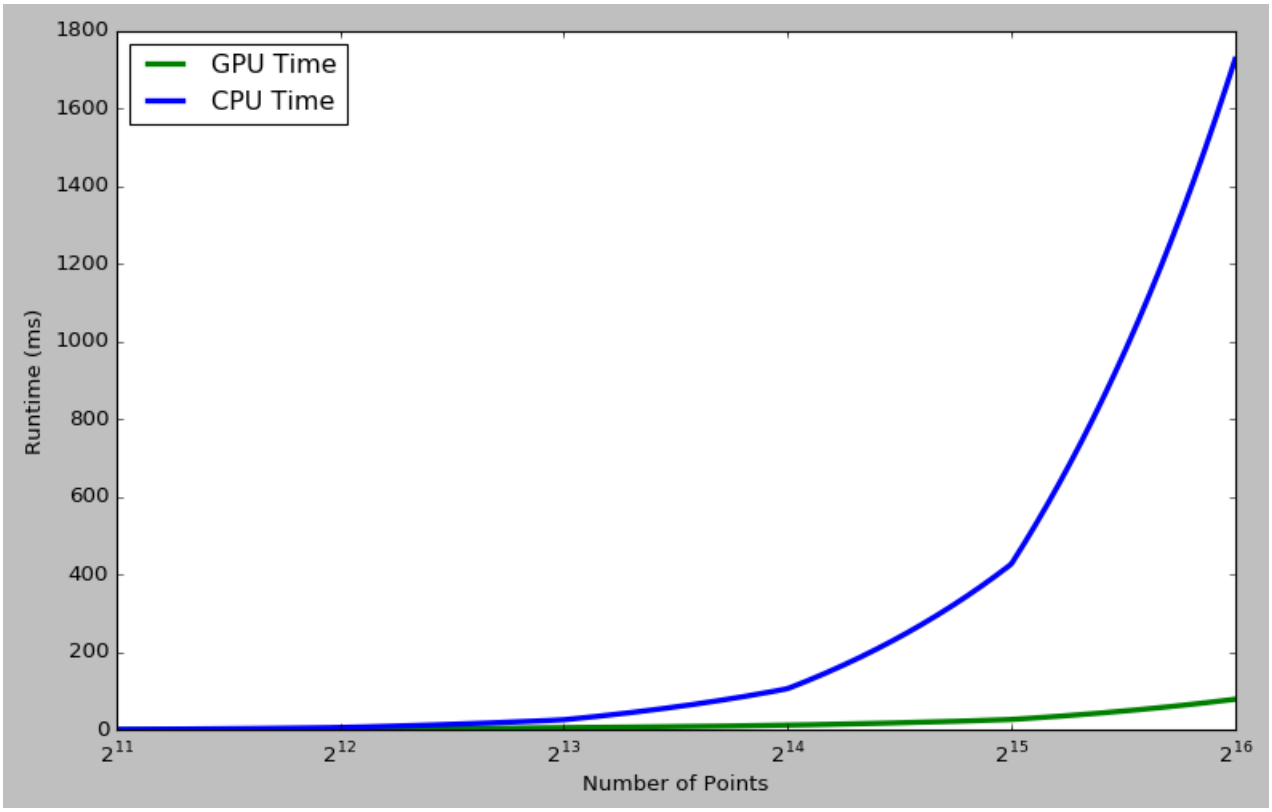


Table #2: CPU vs. GPU Execution Times for $n = 2^k$ $k = 11, \cdots, 16$

# Points	GPU Runtime (ms)	CPU Runtime (ms)
2048	1.580032	1.729494
4096	3.153792	6.709263
8192	6.312320	26.765841
16384	12.877216	106.740486
32768	27.455296	427.333282
65536	79.591953	1725.874512

Figure #2: Data From Table #2



Observation: Initially, the GPU execution time is much greater than the CPU time. This is most likely due to the extra work inherent in calculating the answer in parallel. This includes threads stalling on conditional statements, reduction operations to aggregate results within blocks, and a final single-thread reduction to aggregate the global result into D[0]. The GPU is also potentially initially slower due the device having a lower clock speed than the CPU (about 1/4th the speed for the K20s on ada).

For small numbers of points, this overhead is much greater than the speedup generated by parallelizing the computation. However, as the number of points increases, this overhead becomes less significant. Instead, the runtime becomes more dependent on the complexity of the actual algorithms being run on the CPU and the GPU.

The algorithm on the CPU has a complexity of O(n^2), where n is the number of points. On the GPU version, n threads are created, one for each point. Each thread computes between 1 and n - 1 distance computations, depending on the threads index. This means, for n threads, each thread does O(n) computations. Along with this, each thread participate in an intra block

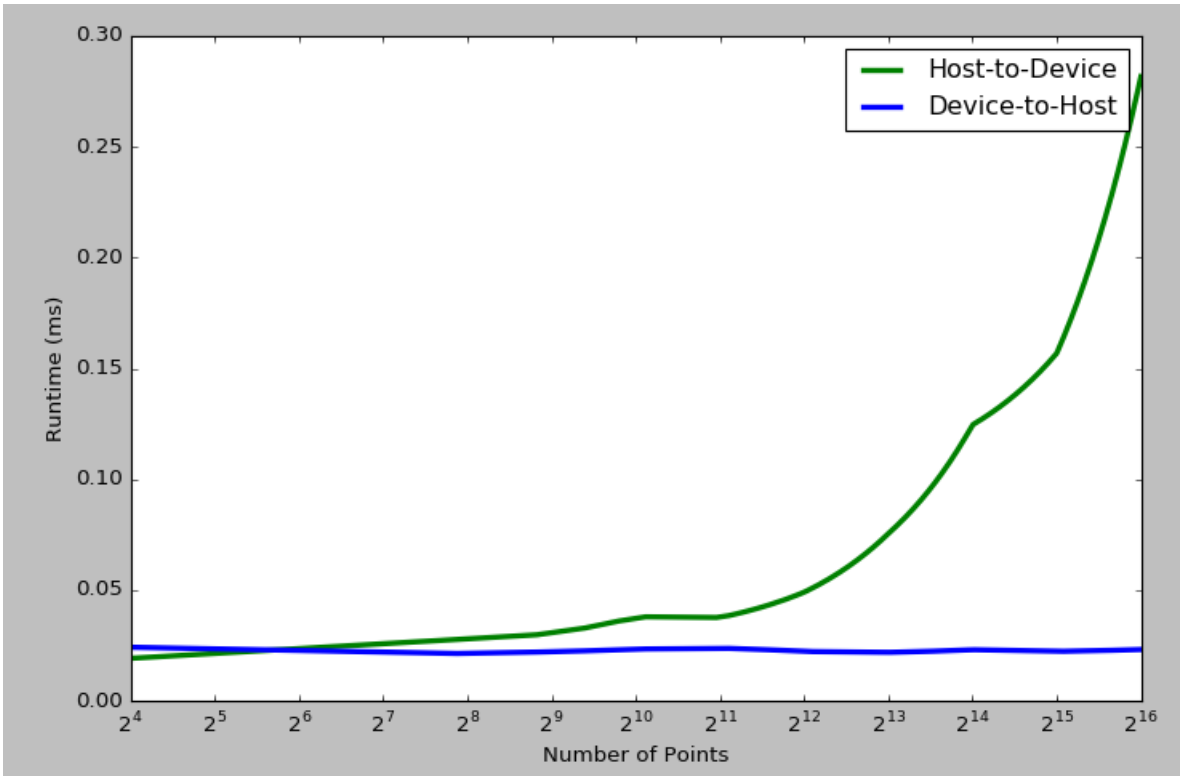
Because of this analysis, it is reasonable to expect that the runtime of the CPU code will scale approximately quadratically with the number of points. Additionally, the GPU time should scale in a linear manner for an increasing input size. This organization is demonstrated in Figures #1 and #2. Since the GPU complexity is less than the CPU complexity, by definition, there is some point where they will switch places so that CPU runtime will be greater than GPU runtime. In the experiments run on ada, this occurs at $\boxed{n = 2048}$.

Question #3: Plot the data transfer time from host to device and device to host on the same plot for $n = 2^k$ for k = 4, ..., 16

Table #3: Data Transfer Times

# Points	Host-to-Device	Device-to-Host
16	0.019200	0.024192
32	0.018976	0.022016
64	0.018976	0.020512
128	0.029024	0.020736
256	0.027456	0.021504
512	0.030528	0.022176
1024	0.037952	0.023392
2048	0.037600	0.023808
4096	0.049024	0.022272
8192	0.075488	0.021888
16384	0.124544	0.023072
32768	0.156832	0.022336
65536	0.281632	0.023168

Figure #3: Data From Table #3



Observations: The device-to-host copy time remains rather static throughout all of the program tests, regardless of the number of points. On the other hand, the host-to-device time increases drastically as the num_points increases. Only one point is copied back to the device after the computation is complete, which explains the constant device-to-host copy time. However, in order to start the computation, data that scales linearly with the number of points must be copied from the host to the device, explaining the approximately linear increase in host-to-device copy time.