CS3210 Assignment 1 - part 2 Report
Helena Crawford (A0207499N) & Luukas Kuusela (A0208658U)

Differences in CUDA implementation:

- Skeleton code provided was used, correcting input scanning
- Vector struct was removed, as vector operations were not returning correctly
- Specifying thread number was allowed

Parallelization

The calculation of wall and particle collision times for each particle is done on the device, with a block for each type of collision. If the number of threads exceeds the number of particles, threads in a block not corresponding to a particle do not calculate. If the number of particles exceeds the number of threads, the collision kernel is launched multiple times in sequence (passed the number of particles already calculated for) until all particles have been accounted for.

A second round of parallelization saw further improvements from advancing particles in parallel, contrary to the failure to speedup this change brought in OpenMP. Since no synchronization is needed for updating seperate values, the advancing kernel is launched once with as many blocks as needed to advance all particles with the specified number of threads per block.

Synchronization

Each block in the collision kernel synchronises thread access to the global minimum collision instances (wallCol and partCol) by atomic updating and syncthreads() barriers. These ensure every thread has compared with the shared minimum collision time (blocktime) before the thread with equal value can update the other collision fields.

Final Results:

New times are given in real/user/system format, in seconds

CUDA program times

Using input (particles = 100, size = 200, radius = 2, steps = 50, perf)

Jetsontx2-07

(n < threads) 512 threads: 3.647/0.344/0.508s, 3.658/0.376/0.496s, 3.012/0.404/0.384s
(n = threads) 100 threads: 4.31/0.596/692s, 3.778/0.564/0.632s, 4.194/0.660/0.548s
(n > threads) 16 threads: 12.794/2.324.1.876s, 12.641/2.324/1.832, 11.576, 2.256, 1.688s

xgpe0

512 threads: 1.016/0.398/0.250s, 0.673/0.392/0.248s, 1.863/0.375/0.247s
100 threads: 0.713/0.345/0.277s, 0.618.0.351/0.242s, 0.639/0.355/0.259s
16 threads: 2.199/1.406/0.734s, 2.523/1.481/8.24s, 2.232/1.45/0.726s

OpenMP program times

Using input (particles = 100, size = 200, radius = 2, steps = 50, perf)

Xeon

Sequential: 2.69s, 2.63s, 2.61s
Parallel [20 threads]: 0.582s, 0.632s, 0.585s

i7
Sequential: 1.55s, 1.56s, 1.52s
Parallel [8 threads]: 0.707s, 0.652s, 0.618s
Jetsontx2-07
Sequential: 3.391/3.368/0.020s, 3.418/3.404/0.012s, 3.358/3.332/0.024s
Parallel: 0.998/3.864/0.016s, 0.988/3.864/0.020s, 0.979/3.824/0.016s

Conclusion
Although incremental improvements were made, the CUDA implementation was unable to run faster than the OpenMP program. Some possible reasons:

- Method of testing. Perf was used to record time for the OpenMP testing, while bash time was used on the jetson and compute cluster.
- Specifying thread number. The OpenMP program automatically ran a number of threads depending on the problem size and hardware, while the chosen number of threads in the CUDA program may not optimize hardware performance.
- CUDA overhead. Although variables were declared as locally as possible and global access restricted, there may still be significant overheads in memory copying and access that are slowing the CUDA program down compared to its OpenMP counterpart.
- Sequential structure. Input reading and calculation of vector data had to be reworked in the CUDA program, which may have caused slowdown in the CPU.

However, differences in hardware, structure, runtime, profiling and testing mean this slowdown cannot be generalized. Given more time, we would have rewritten part 1 of the assignment to resemble part 2 structurally to give fairness to the testing, and profiled part 1 to gain more insight into the slowdown.