Parallel Particle Simulation

Au Liang Jun (A0173593W), Yan Hong Yao Alvin (A0159960R)

CS3210 Parallel Computing AY19/20 Sem 1

Part 1 (OpenMP)

30/09/2019

Introduction

In this report, we detail our attempt at simulating particle movement in a constrained space. We first elaborate on our program design, including our iterative changes and parallel strategy, before presenting and analysing experimental results.

Program Design

Assumptions

Due to the complexity of the problem, our scope was reduced for this project by having a few assumptions:

- Each particle only collides once in every step and will phase through subsequent particles.
- After a particle is involved in a collision, if it collides with a wall in the same time step, it will stop with the wall for the remaining duration of the timestep.
- If a particle begins a timestep within another particle, due to only being allowed to collide once in the previous timestep, they will immediately collide at the start of the timestep and follow collision rules as described (even though they are within one another).

Discussion

Even though the problem is presented to us in discrete timesteps *S*, particle movement is continuous and a collision can take place at any arbitrary time between two timesteps. The coordinates of each particle printed are simply snapshots of the particles at particular times.

Our program thus hinges on an algorithm **`moveParticles()`** that executes for each if the *S* timesteps:

> **`moveParticles():`**
> Input: $[p^0_t, p^1_t, p^2_t, \ldots, , p^{n-1}_t]$
> Output: $[p^0_{t+1}, p^1_{t+1}, p^2_{t+1}, \ldots, , p^{n-1}_{t+1}]$
>
> where p^i_t is an object representing the particle with index *i* at timestep *t*. Each particle has `x, y` coordinates and `vX, vY` velocities.

Note that the program input *N* - number of particles, *L* - size of square environment, *r* - radius of each particle, *S* - number of timesteps, are global constants in our program.

Apart from the assumptions listed in the previous section, we wanted our algorithm implementation to be as correct as possible in simulating particle collision. Some of the notable considerations include:

- If particle A is compared with particle B and found to collide at some arbitrary time within a timestep, the collision can only occur if particle B is not involved in an earlier collision with another particle.

- Corner collisions when a particle collides with two walls simultaneously.

Modelling

We utilised an Object-Oriented approach in modelling the problem and representing information.

Particle

Each particle contains its key attributes: `x` and `y` coordinates, `vX` and `vY` velocities and its index `i`.

CollisionEvent

This abstract class represents a particle event. Notably, it has a virtual method `execute()` that when called, applies the event and updates the position and velocities of the involved particles to be that of the end of the timestep. Each particle event also contains a time of occurence between 0 and 1.

ParticleCollisionEvent

Represents a collision between two particles.

WallCollisionEvent

Represents a collision between a particle and a wall.

NoCollisionEvent

Represents no collision taking place during the timestep. The time of occurence of this event is always 1.

Algorithm Design

The flow of the algorithm to perform collisions can be broken down into 3 broad steps, irrespective of algorithm implementation:

1. Determine time of all particle-particle and particle-wall collisions
2. Determine which collisions take place
3. Apply the collisions to obtain final particle positions and velocities

Iteration 1: Naive Sequential Algorithm

Naively, the 3 steps of our algorithm can be performed as such:

```
        moveParticlesV1():

1.      For each particle: -- $O(n^2)$
        1.1.    Compare every particle with every other particle to determine time of collision
                within the timestep. If a collision takes place, the time is stored in a n x n matrix
                -- [$O(n^2)$]
        1.2.    Calculate the time it next collides with a wall and store in an n length array --
                [$O(n)$]
2.      Iterate over the values calculated in steps 1.1 and 1.2 and select the global minimum
        value, accepting that as a valid collision. Repeat until all particles are accounted for. --
        $O(n^2 \times n)$
3.      Apply the valid collisions to the particles to get resultant positions and velocities. --
        $O(n)$
```

Overall Time Complexity: $O(n^3)$

Iteration 2: Efficient Sequential Algorithm

To reduce the time complexity of the algorithm, we explored a more efficient way to obtain the global minimum value in step 2. We considered the use of a heap to store the time of particle-particle and particle-wall collisions calculated in step 1, sorted by the time taken before a collision occurs. With ~(n x n)elements, a heap would be able to retrieve n elements in $O(n^2 \log n)$ time.

Overall Time Complexity: $O(n^2 \log n)$

However, when attempting to parallelise the algorithm, we found that we were not able to do so with Open MP's work-sharing constructs. The staggered thread create upon repeated extract-min on the heap is not possible within OpenMP, given that OpenMP's master thread necessarily has to wait for the slave thread to complete before looping.

The design that we have used in this iteration would be more suitable for a task pool pattern, where a master thread pops from the heap and puts valid collisions into a task pool for worker threads to execute.

Iteration 3: Parallelizable Sequential Algorithm

With consideration for OpenMP's parallel mechanisms, we then designed another algorithm that we knew could be exploited to run in parallel. We elaborate on the details of our parallel strategy in the next section; below details the final iteration of our design in sequential implementation:

```
     moveParticlesV3():

1.   (as in moveParticlesV1()) For each particle, compute the time it takes to collide with
     all other particles and the wall. -- $O(n^2)$

2.   Initialise an empty event found set. Repeat while event found set does not contain all
     particles: -- $O(n^3)$

     2.1.    For particles without found event, find its earliest event: -- [$O(n^2)$]

         2.1.1.    Check against wall collision times

         2.1.2.    For every other particle j without found collision:

             2.1.2.1.    Check against particle collision time with j

     2.2.    For each particle, check the earliest event found for the following three cases:
             -- [$O(n)$]
```

2.2.1. If a particle x has earliest collision time with another particle y, check that y has not been already set to collide with another particle, and then check that its earliest collision time found is with particle x. If so, set the particles to collide with each other, and add both particles to the event found set.

2.2.2. If its earliest collision time is with a wall, set it to collide with the wall, and add particle to event found set.

2.2.3. If a particle's earliest collision time is not within this timestep, then the particle will never collide with any particle or wall in this timestep and just move according to its velocity. Add the particle to the event found set.

3. (as in `moveParticlesV1()`) For each particle event (collision or no collision) found in the previous step, simulate the movement of the particle for the timestep -- O(n)

Overall Time Complexity: $O(n^3)$

Parallel Strategy

Analysing the steps in our algorithm, we noticed that data parallelism was more applicable to our approach than task parallelism. This was as each broad step depended on the previous and could not be executed concurrently. However, within each step, the same task was performed repetitively on different data, with no dependencies between each execution of the task (Steps 1, 2.1.1, 2.1.2, 2.2, 3 in `moveParticlesV3()`).

Work-Sharing Constructs

Given the four types of work-sharing constructs[1] in OpenMP that determined how threads were assigned work, we noted that while the *single, master* and *sections* constructs were suitable for task parallelism, the loop construct *for* was more suitable for data parallelism. We thus searched for locations where we could utilise the *for* construct while preserving correctness.

Scheduling

We used OpenMP's default scheduling policy of dynamic with a chunk size of 1. Having tested performance with static, guided and dynamic scheduling[2] with larger chunk sizes (on 20 threads on the Dell Precision and 2000 particle test data), we did not observe any consistent improvements in runtime.

[1] "OpenMP - Wikipedia." https://en.wikipedia.org/wiki/OpenMP. Accessed 29 Sep. 2019.

[2] "OpenMP: For & Scheduling - Jaka's Corner." 13 Jun. 2016, http://jakascorner.com/blog/2016/06/omp-for-scheduling.html. Accessed 30 Sep. 2019.

Furthermore, dynamic scheduling appeared suitable for our problem. Different parts of our parallelised problem have varying computation costs, such as for computing collisions. Particle collisions require many more floating point operations than wall collisions, and later particles (by index) do not need to perform a collision computation with another particle as the lower indexed particle would have performed all the collision computations in our code. Dynamic scheduling would enable thread workers who complete their tasks earlier to receive a new chunk to execute.

Implementation

The 3 steps of our algorithm were parallelised as follows:

1. The calculation of collision times between a particle and a wall and between particles solely depend on a particle's position and velocity at the start of a timestep. As such, the iterations detailed below are not dependent on one another. The first *for* directive first splits the work up by each particle. The wall collision time of the particle is calculated and then the work is split again with the second *for* directive, to calculate the time of particle-particle collisions. Note that the wall collisions and particle collisions were not run in parallel with each other with the *sections* construct due to the computationally short nature of calculating time of wall collisions. -- O(n^2) → O(1)

```
// time of particle-particle collisions
JaggedMatrix particleCollisionTimes = JaggedMatrix(n);
// time of particle-wall collisions
double wallCollisionTimes[n] = {};

// calculate collision times
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    wallCollisionTimes[i] = timeWallCollision(*particles[i]);

    #pragma omp parallel for
    for (int j = i+1; j < n; ++j) {
        double particleCollisionTime = timeParticleCollision(*particles[i],
*particles[j]);
        particleCollisionTimes.set(i, j, particleCollisionTime);
    }
}
```

2. For the second step, there were two key substeps represented by the two sets of for loops within the outer while loop. They correspond to steps 2.1 and 2.2 respectively in the **`movePartclesV3()`** pseudocode.

   In 2.1, we were performing n linear scans to find the earliest collision event of each particle. While the linear scans were independent and thus parallelisable, finding the minimum in parallel would require maintaining a shared min variable. To avoid synchronisation overhead for a relatively simple task, we kept the portion in sequential -- [O(n^2)] → [O(n)]

In 2.2, if-else checks were performed on the earliest events found. Given that the checks did not perform mathematical computations, and the number of earliest events found would be strictly decreasing across iterations, we elected not to parallelise this section of code to reduce overhead. -- [O(n^2)] → [O(n^2)]

Lastly, the outer while loop cannot be parallelised given that each iteration is dependent on the previous outcome. -- O(n^3) → O(n^2)

```
while (foundCount != n) {
    CollisionEvent* temp[n];
    #pragma omp parallel for
    for (int i = 0; i < n; ++i){
        // first assume no collision
        temp[i] = new NoCollisionEvent(particles[i]);

        // check for particle-wall collision
        ...

        // check for particle-particle collision
        for (int j = 0; j < n; ++j) {
            ...
        }
    }

    for (int i = 0; i < n; ++i) {
        if (found[i] != NULL) continue;

        CollisionEvent* e = temp[i];

        // particle-particle collision
        if(ParticleCollisionEvent* v = dynamic_cast<ParticleCollisionEvent*>(e)){
            // determine if valid (other particle also found this collision)
            ...
        }

        // particle-wall collision or no collision
        else {
            found[i] = temp[i];
            ++foundCount;
        }
    }
}
```

3. Finally, each particle had its corresponding event (collision or no collision) applied in parallel. -- O(n) → O(1)

```
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    (*found[i]).execute();
}
```

Overall Parallel Time Complexity: $O(n^2)$

Results

Five measurements were taken and the lowest time to complete program execution was recorded. This was to maximise the likelihood of the program executing fully without the CPU time being taken up by other processes, such as those from the kernel. The recorded time is done within the program using the C++ high resolution chrono library, and is taken as the difference between the start of simulation and end of simulation, with IO operations excluded. The varying of threads used in the program execution is done by setting the OMP_NUM_THREADS environment variable.

Variables were number of particles (problem size) and number of threads, while the timesteps was left constant. As the time required for the problem is expected to scale linearly with respect to the timesteps with no parallelism involved, there is no significant value in changing it. We also left the particle radius and box size unchanged as these are largely related to simulator settings.

Mode: perf
L: 20000
r: 1
S: 500

Node 1

Dell Optiplex 7050
Intel® Core™ i7-7700[3]
4 Cores, 8 Threads
3.60GHz Base Frequency, 4.20GHz Turbo

Problem Size (N) /Number of Threads	**1000**	**2000**	**3000**	**4000**
1	22.936	133.034	265.754	503.858
2	15.996	90.139	185.252	352.421
4	9.481	57.749	116.284	228.299
8	6.519	42.331	88.990	159.705
10	6.749	42.588	85.600	155.069
12	6.804	41.214	80.916	145.730

[3] "Intel Core i7-7700 Retail - (1151/Quad Core/3.60GHz/8MB ...." https://www.amazon.co.uk/Intel-Core-i7-7700-Retail-BX80677I77700/dp/B01N0L41N7. Accessed 29 Sep. 2019.

16	6.674	39.779	78.772	143.766
20	6.741	39.592	76.532	141.060

Node 2

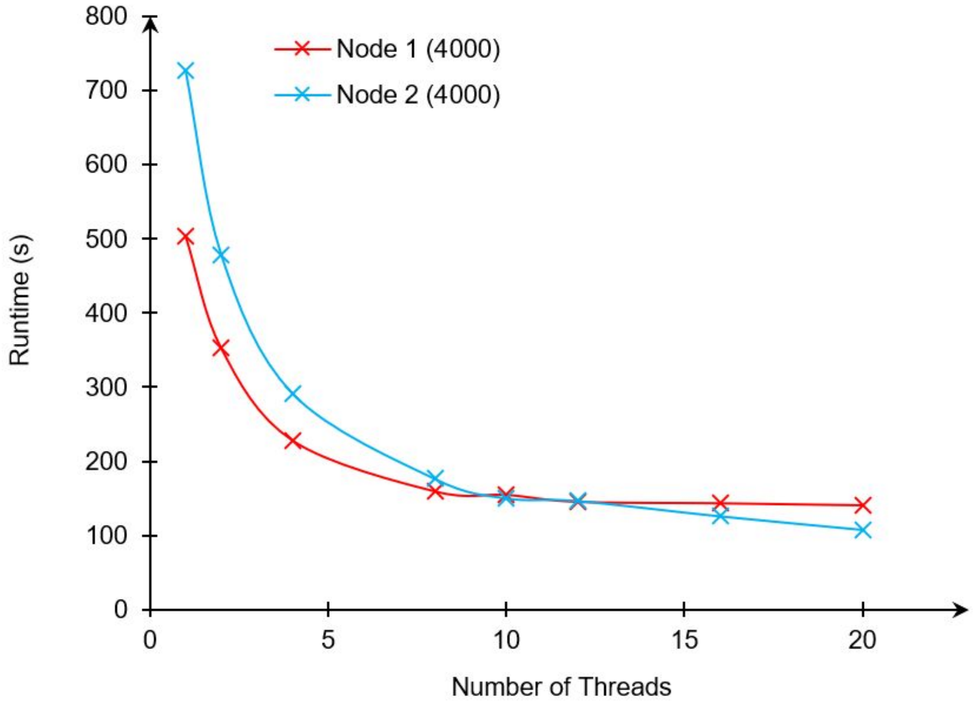
Dell Precision 7820
Intel® Xeon® Silver 4114[4]
10 Cores, 20 Threads
2.20GHz Base Frequency, 3.00GHz Turbo

Problem Size (N) /Number of Threads	**1000**	**2000**	**3000**	**4000**
1	32.965	144.508	381.855	727.296
2	22.556	98.269	256.356	478.756
4	13.942	60.187	153.418	291.548
8	8.932	38.772	95.564	176.328
10	7.659	36.143	81.104	150.260
12	7.569	31.934	79.263	146.728
16	6.666	27.781	68.560	126.072
20	5.548	23.849	58.967	107.812

[4] "Intel® Xeon® Silver 4114 Processor (13.75M Cache, 2.20 ...." https://ark.intel.com/content/www/us/en/ark/products/123550/intel-xeon-silver-4114-processor-13-75m-cache-2-20-ghz.html. Accessed 29 Sep. 2019.

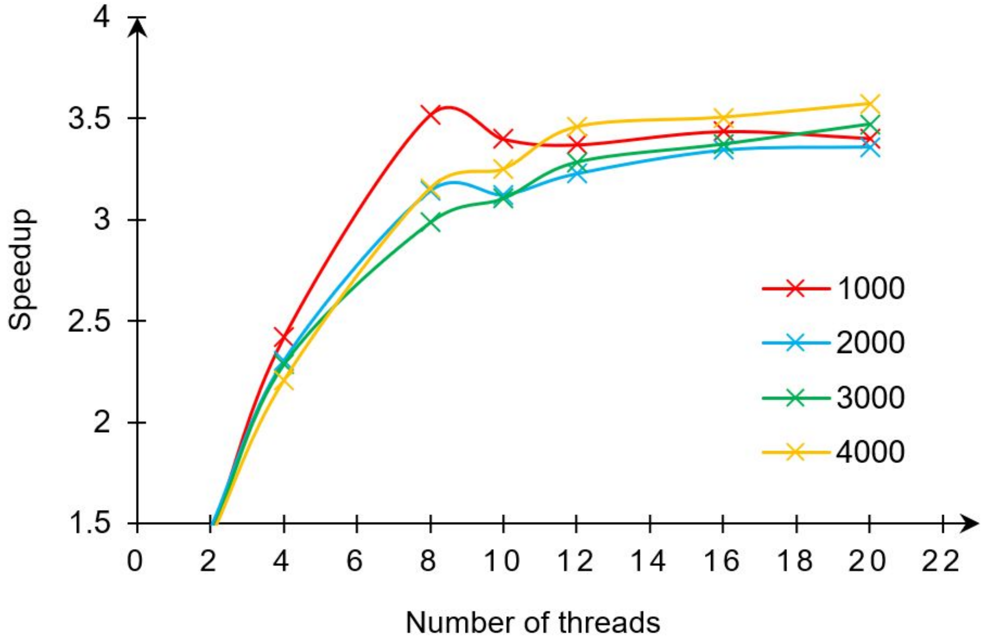
Analysis

Node 1 vs Node 2



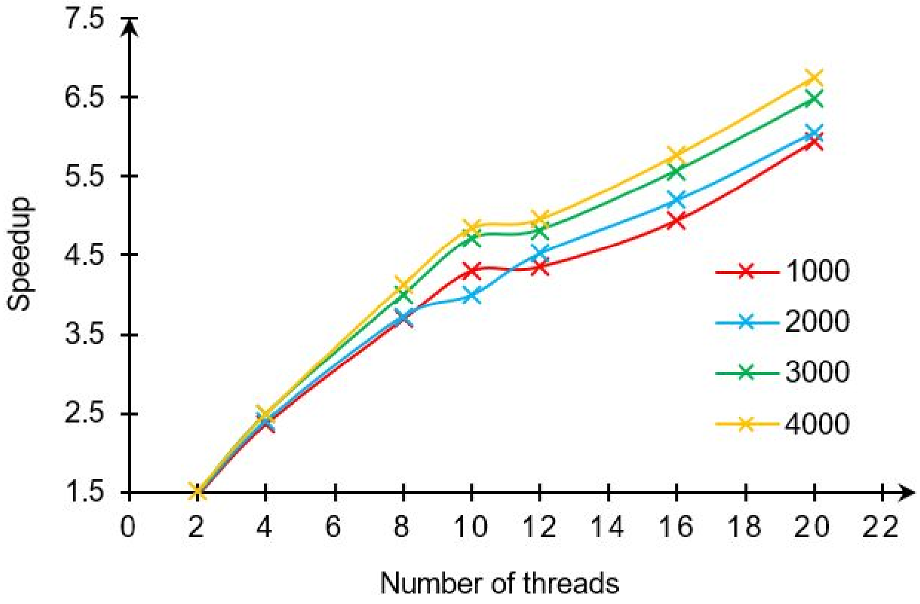
Comparing the two nodes at a fixed problem size (*N = 4000)*, we see that the runtime of Node 1 is lower than that of Node 2 at a small number of threads, but the trend later inverts beyond around 12 threads. This is in line with the hardware specifications of the nodes. Node 1 has fewer physical and logical cores than Node 2; however, each individual core has a higher number of CPU cycles/second than Node 2 (3.60GHz vs 2.20GHz). At thread counts below 8, assuming the same number instructions per cycle between the two machines, the total number of instructions completed per second would be greater for Node 1 as compared to Node 2. Above 8 threads, more cores of Node 2 would be active, while the number of active cores of Node 1 cannot increase further (even taking into account logical cores). This means that the instructions completed per second would get increasingly similar until Node 2's surpasses Node 1's, resulting in the inversion in relative runtime.

Node 1 Speedup



A counterintuitive result is that for large problem sizes, increasing the number of threads past 8 on the optiplex still results in a reduction in the execution time of the program. As the i7 core on the optiplex is capable of running at most 8 threads simultaneously, increasing the number of threads created to run tasks in OpenMP beyond 8 should not result in increased parallelism. A possible reason for this is that when creating only 8 threads, each of the threads is not able to fully utilise all of the CPU computation due to issues like memory access contention, especially for different threads trying to access data for particles stored in adjacent in the memory. Having more threads allows the blocking threads cpu usage to be transferred to another thread instead, allowing computation to continue and increasing throughput.

Node 2 Speedup



The speedup is still increasing roughly linearly with respect to the number of threads used in our program execution. This suggests that our program is still able to scale further with a larger number of cores and threads available before starting to plateau following Amdahl's law. As the sequential portion of the program is roughly the time it takes for a particle to compute its collision timings and collision execution, as we increase the problem size, we expect the proportion of the sequential part of the program to reduce as the problem size increases. Therefore with a large number of particles to simulate (as might be the case in reality), our implementation can enjoy good speedup even on large thread counts. Another observation is that the gradient of the speedup vs thread count graph is lower after passing the 10 thread threshold. This is likely due to the further threads running on the same cores via hyperthreading, which is not as efficient as running on a completely different CPU with its own resources.

Simulation vs Experimentation

Simulation offers some unique advantages over performing experimentation, though it comes with some caveats. First, simulation can be performed much faster than in real time. In our project, we were able to run a test case of 4000 particles in 107 seconds on the Dell Precision for 500 timesteps, which could represent 500 seconds in a real life scenario. This allows results to be obtained much faster using a simulation compared to performing a real life experiment.

Another benefit of simulations is that experimentation can be costly, infeasible, or outright impossible. For example, particle simulations when studying high energy related physics such as ion beams for medical treatment or nanoparticles. In such situations, simulation can allow for a finer grain analysis when working with nanoparticles, such as to simulate particles in extremely small time steps which actual experiment recording cannot capture. For medical treatment, simulations can help to ensure that the treatment is safe before trial on real subjects.

However, the drawback of simulations is that it is dependent on our capability to model the real system. Such as for this project, the simulation model is not fully realistic as particles only collide once per timestep. This reduces the accuracy of the model compared to a real experiment and must be taken into account when making use of its results.

Part 2 (CUDA)

21/10/2019

Program Design

Assumptions

Our assumptions remain the same as with the first part of the assignment.

Algorithm Design

The algorithm used in the OpenMP section of the assignment to perform collisions and the Object-Oriented approach have been retained in our CUDA implementation (albeit with some enhancements). Despite the additional clarifications and corrections, we decided to use the same algorithm to correctly handle issues related to a particle being involved in multiple collisions in a single timestep. This also allows for a more fair comparison between the execution speed of our algorithm on OpenMP and CUDA.

However, we discovered a number of bugs in our OpenMP algorithm, and have fixed them in our CUDA implementation. We have also updated our original OpenMP implementation and reran them to obtain new runtime observations. As a result of our modifications, our OpenMP results are marginally faster than in Part 1 as additional erroneous collisions were removed.

CUDA: Implementation Changes

Firstly, we removed the use of the JaggedMatrix. As particle-particle collision times are the same between any two particles within a timestep, we initially avoided duplicating the information by using a jagged matrix (i-j collision time was stored only if i > j), and controlling access such that the correct value is returned regardless of whether matrix[i][j] or matrix[j][i] was accessed. However, we felt that implementing this data structure would introduce more overhead than benefit in CUDA, and as such defaulted to using a 2-dimensional matrix.

While our previous implementation of the algorithm utilised inheritance to model the three types of events (particle-particle collision, wall-particle collision, no collision) a particle can undergo in one time step, we encountered difficulties utilising this model inside CUDA. Objects needed to be copied into GPU-accessible memory, and as such simply passing an array of pointers to the kernel would not suffice.One way to pass in a list of objects and still be able to access them would be to have the list contain the objects themselves. As CUDA is not compatible with the C++ standard library vector data structure, we attempted to utilise arrays. However, as derived classes each take up a different amount of memory space, an array of the parent class cannot store all instances of derived classes reliably. As such, we removed the derived classes and utilised a single class to represent the different types of events.

CUDA: Optimisations

As each warp in CUDA executes all code within divergent control flows, we seeked to minimise the use of if/else statements within device code to reduce the performance loss. An example of this optimisation is our implementation of particle-wall collisions, where there are 3 large divergent flows. The program was rewritten to maintain the same logic, but reducing the code within the conditional branches to just 3 lines in total.

```
if (xCollide < yCollide) {
        first->x += xCollide * first->vX;
        first->y += xCollide * first->vY;
        first->vX = -first->vX;
        //after handling x collision, need to stop the ball at the edge of box if it collides with y too
        if (yCollide < 1) {
                first->x += (yCollide-xCollide) * first->vX;
                first->y += (yCollide-xCollide) * first->vY;
        }
        else {
                first->x += (1-xCollide) * first->vX;
                first->y += (1-xCollide) * first->vY;
        }
}
//collision with corner of box reverses both
else if (xCollide == yCollide) {
        first->x += xCollide * first->vX;
        first->y += xCollide * first->vY;
        first->vX = -first->vX;
        first->vY = -first->vY;
        first->x += (1-xCollide) * first->vX;
        first->y += (1-xCollide) * first->vY;
}
//same as x collision but for y wall collision happening first
else {
        first->x += yCollide * first->vX;
        first->y += yCollide * first->vY;
        first->vY = -first->vY;
        if (xCollide < 1) {
                first->x += (xCollide-yCollide) * first->vX;
                first->y += (xCollide-yCollide) * first->vY;
        }
        else {
                first->x += (1-yCollide) * first->vX;
                first->y += (1-yCollide) * first->vY;
        }
}
```

OpenMP Time Particle-Wall Collision

As seen above, the logic was split into 3 separate conditions and a different set of instructions were executed for each of them. This allowed each thread to execute only a small amount of instructions depending on the wall collision scenario the particle was involved in.

```
double earlierTime = fmin(xCollide, yCollide);
double laterTime = fmax(xCollide, yCollide);
first->x += earlierTime * first->vX;
first->y += earlierTime * first->vY;
//Reverse direction depending on which collision happens first
if (xCollide <= yCollide) {
       first->vX = -first->vX;
}
if (yCollide <= xCollide) {
       first->vY = -first->vY;
}
//artificially set timing to allow particle to continue after hitting corner
if (xCollide == yCollide) {
       laterTime = 1;
}
first->x += (fmin(1.0, laterTime)-earlierTime) * first->vX;
first->y += (fmin(1.0, laterTime)-earlierTime) * first->vY;
```

CUDA Time Particle-Wall Collision

In contrast, this is the implementation in CUDA. While now each thread needs to execute slightly more instructions than in the OpenMP version, the total number of instructions considering the divergent flow execution is significantly reduced, which will give better performance in CUDA.

We also explored the use of CUDA streams[5] to enable parallel execution of different kernels. By default, consecutive kernel calls run synchronously. However, there were sections in our code that had independent tasks to be run, and as such could be split into separate streams to run asynchronously on the GPU to maximise utilisation, instead of blocking and waiting for the previous kernel to complete computation.

```
timeWallCollision<<<(n-1)/64+1, 64, 0, streams[0]>>>();
...
timeParticleCollision<<<blocksPerGrid, threadsPerBlock, 0, streams[1]>>>();
```

CUDA Time Collision

[5] "CUDA C/C++ Streams and Concurrency - Nvidia." https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf. Accessed 19 Oct. 2019.

For example, the logic for calculating the time of particle-wall collisions and particle-particle collisions are entirely different, and as such were split into separate kernel functions. As both tasks were independent, they were run on different streams concurrently, as seen above.

```
#pragma omp parallel for
for (int i = 0; i < n; ++i)
{
        (*found[i]).execute();
}
```

OpenMP Execute Collisions

Another example of concurrent CUDA execution was the applying of valid collisions to particles. Each particle is exclusively involved in 1 of 3 cases - a particle collision, a wall collision or not in any collision - within 1 timestep. While previously, all three types of collisions were applied together in parallel chunks with the CollisionEvent's execute() method (as seen above), we now grouped the collisions by their types in order to write kernel functions that each contained minimal branching and similar computation.

```
executeParticleCollision<<<(particleCollisionsCount-1)/64+1, 64, 0, streams[0]>>>();
executeWallCollision<<<(wallCollisionsCount-1)/64+1, 64, 0, streams[1]>>>();
executeNoCollision<<<(n-1)/64+1, 64, 0, streams[1]>>>();
```

CUDA Execute Collisions

To run them in parallel, we invoke the kernel to execute particle collisions on one stream, and the kernel to execute wall collisions and no collision movement in a separate stream. As the wall collision and no collision logic is much shorter in instructions in comparison to particle collisions, we run the former two on the same stream and the latter in a different stream as the latter is more likely to take a greater computation time.

Results

The testing methodology remains the same as part 1. We have created test cases beyond the original 4000 particles to 10000 particles, to have a larger dataset for comparison between CUDA and OpenMP implementations. Our observations are in seconds.

Mode: perf
L: 20000
r: 1
S: 500

CUDA

NVIDIA RTX Titan

Problem Size (N)	**1000**	**2000**	**3000**	**4000**	**5000**
	1.219	2.931	5.718	8.292	11.740
	6000	**7000**	**8000**	**9000**	**10000**
	16.589	21.959	28.117	35.293	43.403

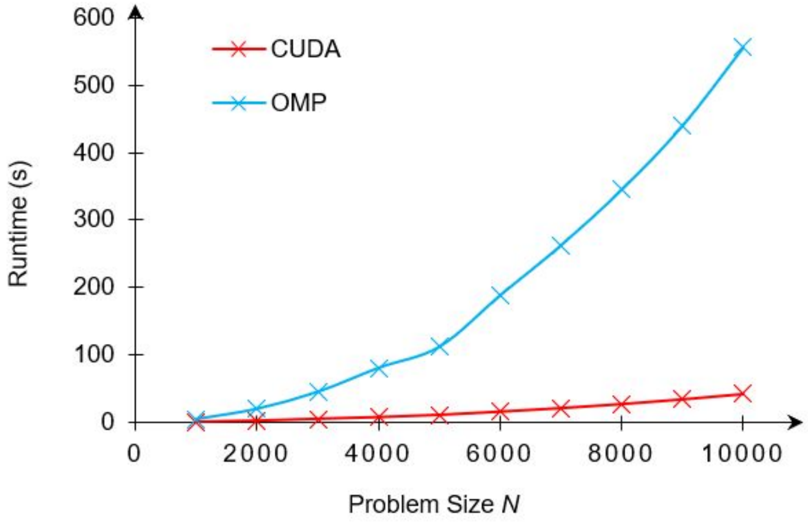
OpenMP

Dell Precision 7820
Intel® Xeon® Silver 4114[6]
10 Cores, 20 Threads
2.20GHz Base Frequency, 3.00GHz Turbo

Problem Size (N) /Number of Threads	**1000**	**2000**	**3000**	**4000**	**5000**
20	5.487	20.669	45.892	81.386	112.672
	6000	**7000**	**8000**	**9000**	**10000**
20	188.897	262.770	345.355	439.981	555.825

[6] "Intel® Xeon® Silver 4114 Processor (13.75M Cache, 2.20 ...." https://ark.intel.com/content/www/us/en/ark/products/123550/intel-xeon-silver-4114-processor-13-75m-cache-2-20-ghz.html. Accessed 29 Sep. 2019.

Analysis



The results show that the CUDA implementation are significantly faster than the OpenMP implementation. This is expected as the total computation capability of the RTX Titan is more than that of the Xeon Silver 4114, especially in terms of FLOPS as majority of the simulation steps requires floating point operations.

Further, as the problem size (particle counts) increases, the speedup of CUDA over OpenMP increases. There are some possible reasons for this.

First, in some kernel invocations such as to execute collisions, the number of blocks created is equal to ParticleCount/64. With 72 streaming multiprocessors in the RTX Titan, we would need at least 4608 particles before our implementation would start creating enough blocks such that each streaming multiprocessor is allocated multiple blocks to execute. This would reduce throughput in computations, especially when a block needs to idle due to global memory access delays. Therefore a larger particle count allows our implementation to achieve higher utilisation rate on the GPU and increases speedup.

Second, as particle counts increases, the number of threads created increases rapidly for both OpenMP and CUDA implementations. CUDA threads have very low creation overhead and scheduling cost compared to OpenMP threads. The additional overhead of parallelism in OpenMP will be significant as our implementation performs multiple fork-joins within a single timestep for computations. This also allows the CUDA implementation to have a lower runtime compared to the OpenMP implementation, giving a better speedup.

Possible Enhancements

As we strove to keep the CUDA and OpenMP implementations similar for a fair comparison, there were a number of changes that we believe might improve the CUDA implementation's performance that we did not have the opportunity to explore.

In particular, we believe that memory access could be more controlled to optimise performance, such as coalescing accesses of global memory access and/or use of shared memory.

Shared Memory: In our implementation, managed memory was used for convenience. As managed memory essentially maintains a copy of the information in both host memory and device global memory, there is non-trivial overhead introduced from copying and synchronisation. We could have perhaps copied data that is accessed repeatedly into shared memory to decrease access time, and kept the memory not needed only in the host.

Global Memory: Furthermore, we could reduce the use of pointers in our kernel code (e.g. CollisionEvent objects containing pointers to Particle objects). Pointers are likely to contain memory addresses that are not contiguous, and thus accesses cannot be coalesced. A possible alternative would be to use minimal structs containing particle data, and directly copying the data in the structs to pass the information around.

Granularity of tasks could also be further experimented on and improved, using tools like the CUDA profiler[7] to obtain information about such as the program runtime, waiting time and number of memory accesses.

Conclusion

CUDA achieves significantly faster runtimes on the particle simulation problem due to its ability to exploit the GPU to perform computations. Between CUDA and OpenMP, we believe that this observation would largely hold across different problems, granted that the considerations of using CUDA are taken care of (granularity, branching etc.). Possible situations where OpenMP would be more advantageous would be:

- Scenarios with task parallelism, where there are distinct tasks to be executed once each in parallel. There would be no need to perform the same computation on multiple data, and OpenMP would be able to parallelise such a problem with its *single*, *master* and *section* work constructs. CUDA's data parallelism would not be helpful here.

[7] "Profiler :: CUDA Toolkit Documentation - NVIDIA Developer ...." 19 Aug. 2019, http://docs.nvidia.com/cuda/profiler-users-guide/index.html. Accessed 19 Oct. 2019.

- When advanced C++ constructs or features are needed. CUDA supports a subset of C++ features (no recursion, standard library), and while it may be possible to adapt the code to be compatible with CUDA, it might instead introduce greater development overhead and reduce maintainability. On the other hand, OpenMP is closely integrated with C and C++ and has markedly less restrictions, allowing the programmer to make only small changes to the original program. Ultimately, this is problem specific and the tradeoff between runtime and developer effort has to be assessed.

- Trivially, when there is no GPU available. OpenMP is suitable when you do not have GPU resources as it only requires a CPU to parallelise your program.