CS3210 Parallel Computing
Assignment 1 (Part 2) Report

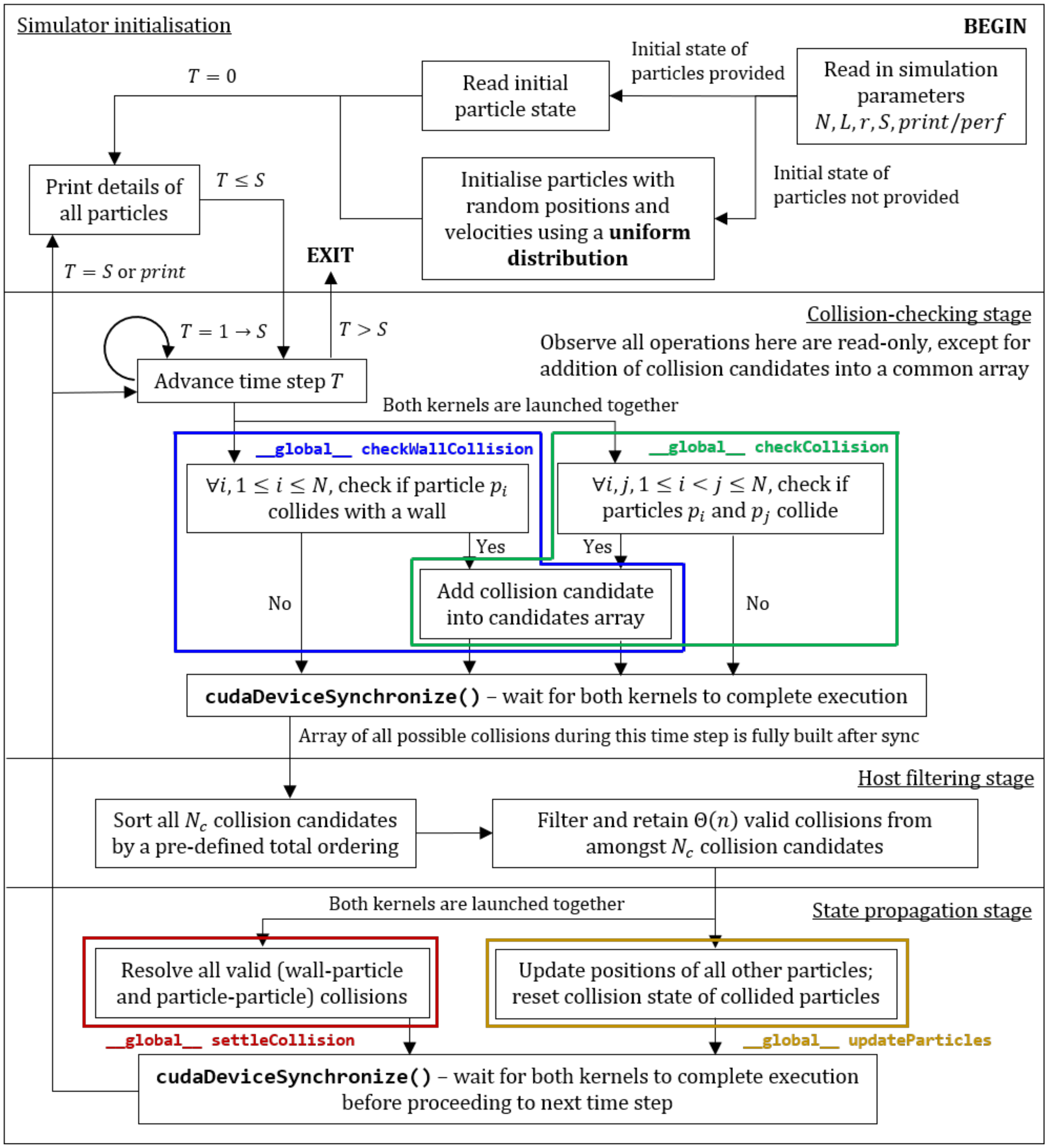
Discrete Particle Simulation with CUDA

Keven Loo Yuquan (A0183383Y) & Lee Yong Jie, Richard (A0170235N)

Contents

1 CUDA Program Design

The discrete particle simulation is implemented fully in CUDA, with no use of external libraries (e.g. Thrust). The overall architecture of the simulator is summarised in Figure 1 below.



Simulator initialisation
BEGIN
Read in simulation parameters $N, L, r, S, print/perf$
Initial state of particles provided
Read initial particle state
Initial state of particles not provided
Initialise particles with random positions and velocities using a **uniform distribution**
$T = 0$
Print details of all particles
$T \leq S$
$T = S$ or $print$
EXIT
Collision-checking stage
Observe all operations here are read-only, except for addition of collision candidates into a common array
$T = 1 \to S$
$T > S$
Advance time step T
Both kernels are launched together
__global__ checkWallCollision
$\forall i, 1 \leq i \leq N$, check if particle p_i collides with a wall
__global__ checkCollision
$\forall i, j, 1 \leq i < j \leq N$, check if particles p_i and p_j collide
Yes
Yes
No
No
Add collision candidate into candidates array
cudaDeviceSynchronize() – wait for both kernels to complete execution
Array of all possible collisions during this time step is fully built after sync
Host filtering stage
Sort all N_c collision candidates by a pre-defined total ordering
Filter and retain $\Theta(n)$ valid collisions from amongst N_c collision candidates
State propagation stage
Both kernels are launched together
Resolve all valid (wall-particle and particle-particle) collisions
__global__ settleCollision
Update positions of all other particles; reset collision state of collided particles
__global__ updateParticles
cudaDeviceSynchronize() – wait for both kernels to complete execution before proceeding to next time step

Figure 1: Overall simulator design for CUDA implementation

2 Implementation Assumptions and Details

2.1 Assumptions

The first assumption from the earlier report has been removed due to changes in the testing conditions.

1. A particle is involved in no more than one collision per time step.
   - Implications
     (a) If a particle collides with a wall after any collision, it is placed at the wall at the end of that time step, and will immediately collide with the wall at the beginning of the next time step
     (b) If particle P collides with a particle Q after any collision, it will phase through the particle Q for the remainder of this time step, or will ignore Q the next time step if they happen to overlap
2. All collisions between particles and with the wall are elastic (kinetic energy and momentum are conserved).
3. It is possible to fit all N particles of radius r into the square of length L without overlapping.
   - Implication
     (a) The simulation exits if one of these two conditions fail: $L < 2r$ (not possible to fit a single particle) or $Nr^2 > L^2$ (not possible to pack N particles in a grid arrangement in a square with area L^2)
4. The set of possible collisions C satisfies a **total ordering**.
   - Implication
     (a) For each time step T, it is possible to sort all collision candidates by this total ordering to determine which collisions should be prioritised over others.

2.2 Implementation Details

Changes in general implementation details from the previous report are in blue.

1. If the initial state of particles are not provided, particles are generated with initial random positions and velocities using a **uniform distribution**.
   (a) We use the pseudo-random number generator function `rand` in the C library seeded with the number 3210.
   (b) Particles are placed randomly in the square without overlapping.
2. Each particle keeps track of its own state: ID, x, y, v_x, v_y, w_c, p_c.
3. Our simulation has an additional parameter `SLOW_FACTOR` in `simulator.cu` that increases the granularity of the simulation for greater accuracy.
   (a) Setting `SLOW_FACTOR` to an integer > 1 slows the initial velocities of all particles by that factor. `SLOW_FACTOR` should be set to a power of two to avoid introducing additional floating-point errors when dividing the particles' velocities.
   (b) The number of steps of the simulation is multiplied by `SLOW_FACTOR` to compensate, i.e. each original step now corresponds to `SLOW_FACTOR` "micro-steps".
4. Collisions are of two types: particle-wall collisions or particle-particle collisions. We describe a particle-wall collision as (P, `null`) and a particle-particle collision as (P, Q).
   (a) As particle-particle collisions are symmetric (i.e. P collides with $Q \iff Q$ collides with P), we generate these collisions such that P is the particle with lower integer ID to avoid duplicates.
5. When sorting collision candidates with the C library function `qsort`, we enforce this total ordering in the function `cmpCollision`. For two collision candidates C_1 and C_2,
   - ■ $C_1 < C_2$ if C_1 occurs before C_2 (priority by time)
   - ■ If C_1 and C_2 occur at the same time
     - □ $C_1 < C_2$ if ID of P in C_1 < ID of P in C_2 (priority by ID)
     - □ If C_1 and C_2 both involve the same particle P
       - ■ $C_1 < C_2$ if C_1 is a wall collision (priority by type)

- ■ If C_1 and C_2 are both particle-particle collisions
  - □ $C_1 < C_2$ if ID of Q in C_1 < ID of Q in C_2 (priority by ID)

6. For each particle in each time step, we check once if the particle collides with any of the four walls.
   (a) A particle is treated to have collided with a wall if it would come within a distance of $\epsilon =$ `1E-8` to the wall within that time step.

7. All possible particle-particle collision pairs are checked for each time step. The total number of collision checks performed is thus

$$\begin{aligned} N_{potential\ collisions} &= N_{wall-particle} + N_{particle-particle} \\ &= N + \frac{N(N-1)}{2} \\ &= \Theta(N^2) \end{aligned}$$

8. Particle-wall collisions are checked by solving the trajectory equation of a particle and the position equations of the wall. The equations of the walls are $x = 0, x = L, y = 0, y = L$.

9. Particle-particle collisions are checked by solving trajectory equations of two particles during the given time step.

   Consider two particles P, Q during a given time step $0 \leq \Delta t \leq 1$. From Pythagoras' theorem, the distance between them is

$$d = \sqrt{((x_Q + v_{xQ}\Delta t) - (x_p + v_{xP}\Delta t))^2 + ((y_Q + v_{yQ}\Delta t) - (y_p + v_{yP}\Delta t))^2}$$

   or re-written in terms of deltas (differences in state)

$$d = \sqrt{(\Delta x + \Delta v_x \Delta t)^2 + (\Delta y + \Delta v_y \Delta t)^2}$$

   The particles intersect when $d = 2r$, i.e. the particles touch at their circumference, hence by expanding and collecting terms we get the quadratic equation of form $A(\Delta t)^2 + B\Delta t + C = 0$,

$$\begin{aligned} & \Delta x^2 + 2\Delta x \Delta v_x \Delta t + \Delta v_x^2 (\Delta t)^2 + \Delta y^2 + 2\Delta y \Delta v_y \Delta t + \Delta v_y^2 (\Delta t)^2 = (2r)^2 \\ \implies & (\Delta v_x^2 + \Delta v_y^2)(\Delta t)^2 + (2\Delta x \Delta v_x + 2\Delta y \Delta v_y)\Delta t + (\Delta x^2 + \Delta y^2 - 4r^2) = 0 \end{aligned}$$

We observe that $A = (\Delta v_x^2 + \Delta v_y^2) > 0$ and thus the curve $y = d(\Delta t)$ is concave up. The discriminant for this quadratic equation, $B^2 - 4AC$, is

$$\text{discriminant} = (2\Delta x \Delta v_x + 2\Delta y \Delta v_y)^2 - 4(\Delta v_x^2 + \Delta v_y^2)(\Delta x^2 + \Delta y^2 - 4r^2)$$

If this discriminant is ≥ 0, then the particles collide for some value of Δt, and we solve for this Δt. There are two possible roots,

$$\Delta t = \frac{-B \pm \sqrt{\text{discriminant}}}{2A}$$

Since the quadratic curve is concave up, we only compute and examine the first root (when the particles are approaching each other). This is

$$\Delta t = \frac{-B - \sqrt{\text{discriminant}}}{2A}$$

If $0 \leq \Delta t \leq 1$, then particles P, Q collide during this time step.

Note that, it is possible for $\Delta t < 0$ - this corresponds to cases where particles are overlapping from a previous step. We do not consider these as collisions in the current step and let these two particles phase through each other.

10. To ensure the collision candidates array can accommodate $\Theta(N^2)$ potential collisions (in the limit of large N when particle-particle collisions dominate), we dynamically allocate an array in managed memory with sufficient space to store $N^2/2$ collision structs.

    - Unfortunately, `cudaMallocManaged()` fails to allocate the required space when $N = 64000$.

3 Problem Decomposition

3.1 Host-device Work Division

The entire simulation can be divided into multiple phases:

1. [**Host**] Initialisation: reading in parameters and optionally initial particle states; randomising particles otherwise
2. [**Host + device**] Step-wise simulation: see below
3. [**Host**] Output of simulation state: printing states of all particles ($S = 0$ and $S = N$ for `perf` mode, or all steps for `print` mode)

We decided to let the **host** perform the simulator initialisation and output of the simulation stage, since these tasks are inherently serial.

- The initialisation of particle with random positions is serial as our simulator guarantees that particles are placed without overlapping, requiring each new particle placed to be checked against all previous particles placed.
- The printing of the simulation state can be parallelised, but this would make the order of the output indeterminate and difficult to check.

For step-wise simulation, we see from Figure 1 that computing a single step comprises three distinct stages, two of which contain multiple tasks:

1. [**Device**] Collision-checking: particle-wall and particle-particle
2. [**Host**] Filtering: prioritising valid collisions
3. [**Device**] State propagation: resolving collisions and updating positions

The three stages must be completed serially for each step, requiring inter-stage synchronisation points. However, within each stage, it is possible to complete the tasks in parallel, eliminating the need for intra-stage synchronisation points.

1. Synchronisation between the host and device between each stage is achieved with a call to `cudaDeviceSynchronize()`
2. During collision-checking, each CUDA thread is only reading the current state of its assigned particle(s)
    - For a given particle P, this allows its collision checks with walls and other particles to proceed concurrently

    - However, to prevent race conditions, synchronisation is required when a CUDA thread
        - adds a new collision candidate to the global array, and
        - increments the number of collision candidates in the global counter

3. During state propagation, a particle's state will only be updated by one CUDA thread
    - If a particle was involved in a collision, its state will be updated when the collision is resolved
    - Otherwise, its state will be updated when its next position is computed; the collision status of collided particles will also be reset back to `false` during this single pass

Therefore, we decided to represent each task in each of these two stages with a separate kernel, for a total of four. They are

1. `__global__ checkWallCollision` - checks all particle-wall collisions
2. `__global__ checkCollision` - checks all particle-particle collisions
3. `__global__ settleCollision` - resolves all valid collisions after filtering
4. `__global__ updateParticles` - updates state of uncollided particles; resets collision status of collided particles

For a given time step, we launch the first two kernels together during the first stage. Similarly, after the host has completed filtering out the valid collisions, we launch the latter two kernels together during the third stage.

We opted to let the **host** continue to perform the sorting and filtering of the valid collisions, since

- A good parallel sorting algorithm on the GPU is difficult to implement by hand, without the use of external libraries (e.g. Thrust)
- Filtering out valid collisions is inherently serial since selecting an earlier may invalidate later collisions, due to the constraint that a particle can only collide once in a given time step

3.2 Memory Choices

Our simulator makes use of the following variables that are shared between the host and device:

- Simulation parameters `N`, `l`, `r` and `S`
- Constant parameters `minPosMargin`, `maxPosMargin` and `maxPos` that depends on the above values of `l`, `r` and the floating-point error allowance `EPS = EDGE_TOLERANCE = 1E-8`
- Shared particle array `ps`
- Shared collision candidates array `cs`
- Shared particle collision status array `states`
- Shared counter for number of entries in the `cs`, `numCollisions`

Since the following variables are required by all CUDA threads and does not change throughout a simulation, we opted to place the following in **constant memory**.

- Variables: `N`, `l`, `r`, `S`, `minPosMargin`, `maxPosMargin` and `maxPos`
- The constant memory is slow and read-only, but is **cached**
- If all threads in a warp are reading the same address from the *constant cache*, it is significantly faster than the alternatives
- Accesses to the same address in global memory by different warps cannot be coalesced into a single memory transaction, increasing memory traffic.
- Accesses to the same address in shared memory by threads in a warp lead to **bank conflicts**, leading memory requests to be serialised.

We opted to avoid the use of shared memory in favour of local memory for multiple reasons:

1. Local memory is faster than shared memory, and avoids the issue of **bank conflicts** that arise from multiple threads in a warp accessing an address in the same bank.
2. For kernels `checkWallCollision` and `checkCollision`, a CUDA thread is assigned particle(s) on which to perform a computation - where possible, the assigned structs are copied entirely into local memory to reduce strain on the memory subsystem.

3. For the kernel `settleCollision`, a CUDA thread is assigned a collision struct which we also copied entirely into local memory.

4. For the kernels `settleCollision` and `updateParticles`, most of the work involves updating the state of the associated particles in the global memory.

   - We rely on the L1 cache on each of the Streaming Multiprocessors (SMs) to reduce the read latency for particle attributes from global memory.
     - Since the 64KB of fast memory on each SM is split between the L1 cache and shared memory, avoiding the use of shared memory allows us to prioritise the fast memory for the L1 cache (see Section 10.1)

5. Reduces the length of the CUDA program by avoiding the need to manually shuffle data between local, shared and global memory.

We opted to use **managed memory** for convenience, with its corresponding performance loss. Since the size of the arrays scale with the number of particles N, this allows the unified memory subsystem to manage and shuffle data between the host's RAM and the device's global memory, as required for computation.

3.3 Kernel Design

Notable implementation details for any of the kernels are discussed here.

For the kernels `checkWallCollision` and `checkCollision`, synchronisation is required to ensure correct execution when the shared counter `numCollisions` and shared collisions array `cs` are updated by multiple CUDA threads.

- We use the provided atomic function `atomicAdd` to add `1` to the shared counter `numCollisions` in global memory when a CUDA thread computes a new collision candidate.

- Since `atomicAdd` returns the previous value at that memory address, this gives each CUDA thread a unique index in the `cs` array to add its collision candidate to.

3.4 Grid and Block Dimensions

We used 0-based indexing for this entire section, and flr denotes the floor function.

For the kernels `checkWallCollision` and `updateParticles`, we only need a single pass over all N particles to perform computations. Therefore, we decided to launch both kernels with the following parameters:

- 1D kernel grid: $flr((N + 32 - 1)/32)$ blocks
- 1D thread block: 32 threads

Since the thread block is 1D with 32 threads, we assign it a contiguous 32-particle section of the particle array `ps`, with each thread responsible for performing computations on a single particle. This increases the spatial locality of memory accesses to global memory (where `ps` resides), allowing memory accesses by threads in the same warp to be coalesced to reduce memory traffic.

The mapping of blocks in the kernel grid B_i to sections of the array is shown in Figure 2.

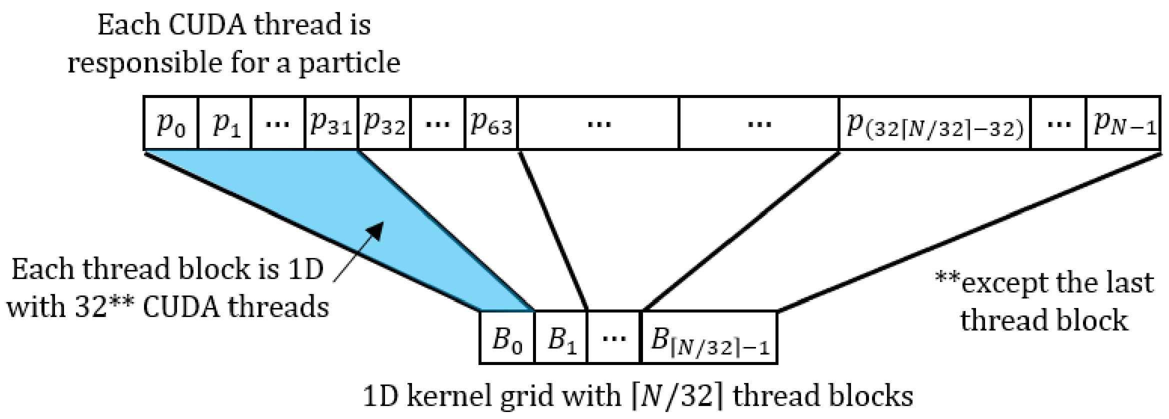

Each CUDA thread is responsible for a particle
p_0
p_1
p_{31}
p_{32}
p_{63}
$p_{(32\lceil N/32\rceil-32)}$
p_{N-1}
Each thread block is 1D with 32** CUDA threads
**except the last thread block
B_0
B_1
$B_{\lceil N/32\rceil-1}$
1D kernel grid with $\lceil N/32\rceil$ thread blocks

Figure 2: Mapping of 1D particles array to 1D kernel grid

For the kernel `settleCollision`, we only need a single pass over the array of N_c valid collisions (note that N_c varies with time step S). Thus, similar to the above, we launch the kernel with the following parameters:

- 1D kernel grid: $flr((N_c + 32 - 1)/32)$ blocks
- 1D thread block: 32 threads

The mapping of blocks in the thread grid B_i to sections of the array is shown in Figure 3.

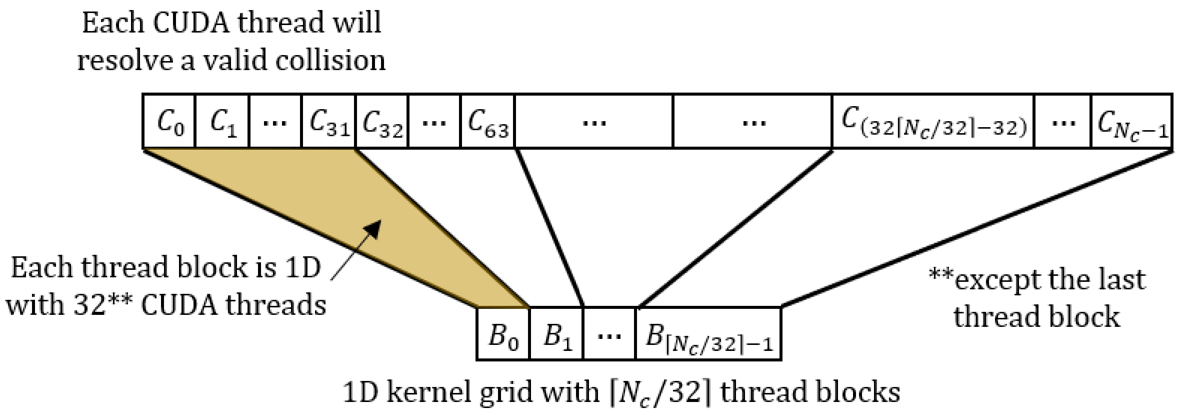

Each CUDA thread will resolve a valid collision
C_0 C_1 ⋯ C_{31} C_{32} ⋯ C_{63} ⋯ ⋯ $C_{(32\lceil N_c/32\rceil-32)}$ ⋯ C_{N_c-1}
Each thread block is 1D with 32** CUDA threads
**except the last thread block
B_0 B_1 ⋯ $B_{\lceil N_c/32\rceil-1}$
1D kernel grid with $\lceil N_c/32\rceil$ thread blocks

Figure 3: Mapping of 1D collisions array to 1D kernel grid

The mapping for the kernel `checkCollision` is not as trivial. Since we only check particle-particle collision pairs for p_i and p_j where $0 \leq< i < j < n$ (where i, j are the indices of the particle in the array `ps`, we can visualise all the computations to be performed for each time step in the upper diagonal of a matrix, as shown in Figure 4.

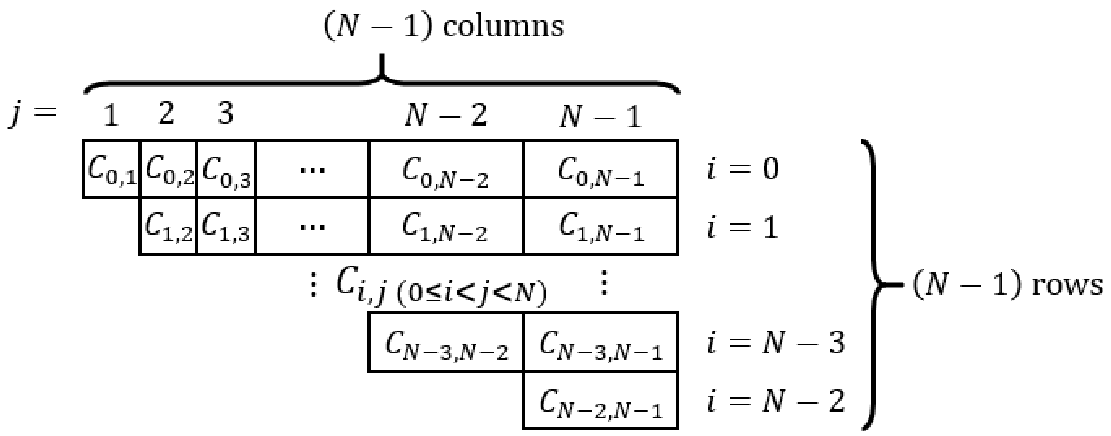
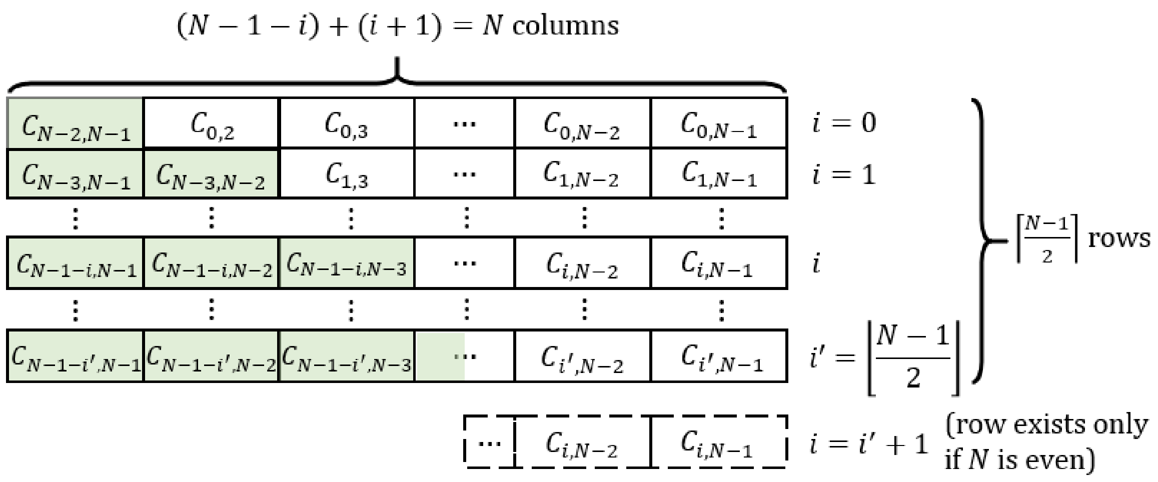

$(N-1)$ columns
$j =$ 1 2 3 $N-2$ $N-1$
$C_{0,1}$ $C_{0,2}$ $C_{0,3}$ ⋯ $C_{0,N-2}$ $C_{0,N-1}$
$C_{1,2}$ $C_{1,3}$ ⋯ $C_{1,N-2}$ $C_{1,N-1}$
$C_{i,j\ (0\le i<j<N)}$
$C_{N-3,N-2}$ $C_{N-3,N-1}$
$C_{N-2,N-1}$
$i = 0$
$i = 1$
$i = N-3$
$i = N-2$
$(N-1)$ rows

Figure 4: Matrix of particle-particle computations

We considered distributing the ith row of the matrix to the ith thread block in a 1D kernel grid of $(N-1)$ blocks to compute. However, this leads to an uneven distribution of work amongst blocks, since the amount of potential collisions to compute for the ith row is $(N-1-i)$. Worse still, it does not fully utilise the GPU since many warps would be executing with fewer than 32 active threads.

Since the upper half of the matrix resembles a triangle, we decided to "fold" the lower half of the matrix onto the upper half, by reflecting it in both axes and stitching it together to form a rectangle, as shown in Figure 5.



$(N-1-i)+(i+1)=N$ columns
$C_{N-2,N-1}$ $C_{0,2}$ $C_{0,3}$ ⋯ $C_{0,N-2}$ $C_{0,N-1}$ $i=0$
$C_{N-3,N-1}$ $C_{N-3,N-2}$ $C_{1,3}$ ⋯ $C_{1,N-2}$ $C_{1,N-1}$ $i=1$
$C_{N-1-i,N-1}$ $C_{N-1-i,N-2}$ $C_{N-1-i,N-3}$ ⋯ $C_{i,N-2}$ $C_{i,N-1}$ i
$C_{N-1-i',N-1}$ $C_{N-1-i',N-2}$ $C_{N-1-i',N-3}$ ⋯ $C_{i',N-2}$ $C_{i',N-1}$ $i'=\left\lfloor\frac{N-1}{2}\right\rfloor$
⋯ $C_{i,N-2}$ $C_{i,N-1}$ $i=i'+1$
(row exists only if N is even)
$\left\lceil\frac{N-1}{2}\right\rceil$ rows

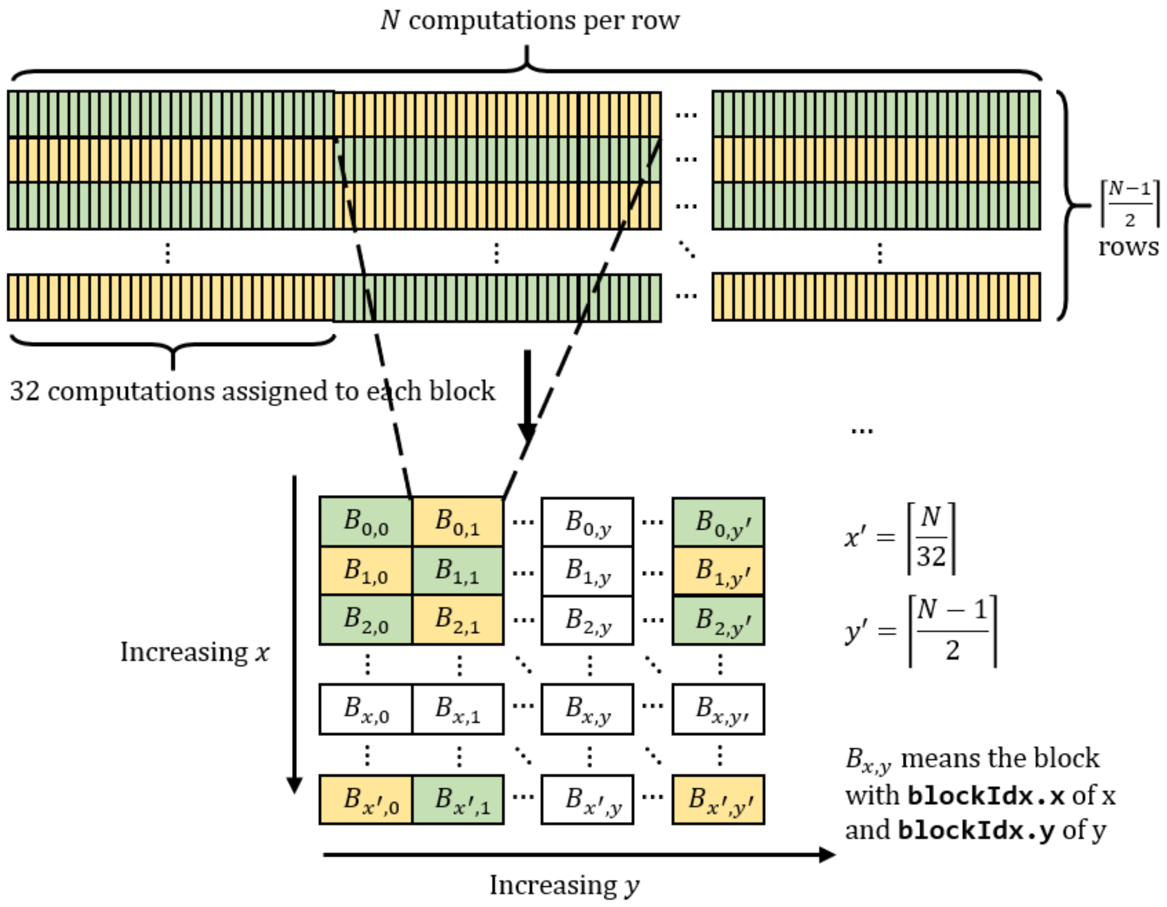
Figure 5: Folded matrix of particle-particle computations

This produces a rectangular matrix with $ceil((N-1)/2)$ rows and exactly N columns, with the shaded green cells denoting the reflected portion. Note that if $(N-1)$ is odd, i.e. N is even, that the matrix will not fold perfectly and there will be a leftover (last) row with only $N/2$ computations.

With this, we can use a 2D kernel grid, and distribute each row of computations to the corresponding row of thread blocks in the grid, as per the previous kernels. This kernel grid would have a dimension of $ceil(N/32)$ in the x-axis and $ceil((N-1)/2)$ in the y-axis.

Note the ith row would thus comprise $N-1-i$ computations from the upper half and $i+1$ computations from the lower half. Figure 6 on the next page demonstrates how the mapping is done.

This increases overhead slightly from the creation of more thread blocks, but utilises the GPU better since there are fewer warps that would execute with less than 32 active threads.



N computations per row
$\left\lceil \frac{N-1}{2} \right\rceil$ rows
32 computations assigned to each block
Increasing x
Increasing y
$B_{0,0}$ $B_{0,1}$ $B_{0,y}$ $B_{0,y'}$
$B_{1,0}$ $B_{1,1}$ $B_{1,y}$ $B_{1,y'}$
$B_{2,0}$ $B_{2,1}$ $B_{2,y}$ $B_{2,y'}$
$B_{x,0}$ $B_{x,1}$ $B_{x,y}$ $B_{x,y'}$
$B_{x',0}$ $B_{x',1}$ $B_{x',y}$ $B_{x',y'}$
$x' = \left\lceil \frac{N}{32} \right\rceil$
$y' = \left\lceil \frac{N-1}{2} \right\rceil$
$B_{x,y}$ means the block with **blockIdx.x** of x and **blockIdx.y** of y

Figure 6: Mapping of 2D computations array to 2D kernel grid

3.5 Thread Arrangement

All our kernels are launched with grids comprising only 1D thread blocks. Hence, the arrangement of threads within a block is trivial - a thread's coordinates in its block is simply `threadIdx.x`. Excess threads are checked for and `return` immediately, performing no work.

For kernels `checkWallCollision` and `updateParticles`, the mapping of a thread to a particle is

- `particleIdx = blockIdx.x * blockDim.x + threadIdx.x`

Similarly, for the kernel `settleCollision`, the mapping of a thread to a collision to resolve is

- `collisionIdx = blockIdx.x * blockDim.x + threadIdx.x`

For the kernel `checkCollision`, there are multiple cases to consider when mapping a thread to two particles `pIdx` and `qIdx`.

- For each thread, compute the following temporary indices
  - `pTemp = blockIdx.x`
  - `qTemp = blockDim.x * blockIdx.y + threadIdx.x`
- Full row (odd N or even N but block has a `blockIdx.x != (N / 2) - 1`)
  - If `pTemp <= qTemp`, then it is a reflected computation
    - `pIdx = N - 2 - pIndex; qIdx = N - 1 - qIndex`
  - Otherwise, `pIdx = pTemp; qIdx = qTemp`
- Leftover row (only for even N): block has `blockIdx.x == (N / 2) - 1`
  - Only compute when `qTemp > pTemp` for $j > i$ since there are no reflected computations in this row

4 Discussion: Floating-point Precision

4.1 Observations

In our initial runs of the default testcase, we noticed that the CUDA implementation did not converge to the same final state as the OpenMP implementation. To investigate this anomaly, we re-ran the default testcase in `print` mode and observed that differences in the floating-point (FP) position of particles manifested as early as the first time step.

4.2 GPU Floating-point Computations

We read the report on *Floating Point and IEEE 754 Compliance for Nvidia GPUs*, published by Nvidia in 2011 [5]. Of particular significance are the following details on FP computations in general:

1. The IEEE 754 standard specifies a common encoding for basic FP formats for consistency across platforms. It also recommends extended FP formats, but does not specify any particular implementation.
   - Single-precision (FP32): 1-bit sign, 8-bit exponent, 23-bit significand
   - Double-precision (FP64): 1-bit sign, 12-bit exponent, 53-bit significand
2. Due to its finite precision, FP arithmetic violates the rules and properties of mathematical arithmetic. The order of execution of basic FP operations influences the accuracy of the final result, due to rounding of intermediate values to a specified precision.
   - Violation of **associativity**: Mathematically, $(A+B)+C = A+(B+C)$, but $rn(rn(A+B)+C) \neq rn(A+rn(B+C))$ for FP arithmetic (here, rn rounds the result)
3. The IEEE 754 standard also specifies a *fused-multiply add* (FMA) operation, which computes the result of $rn(A \times B + C)$ with only one rounding operation.
   - Using the FMA operation results in greater accuracy as compared to computing $rn(rn(A \times B) + C)$, which requires two rounding operations
4. **Subtractive cancellation**: Addition of two FP numbers with similar magnitudes but different signs result in a loss in precision, as many of the leading bits cancel, leaving fewer bits of precision in the final result. However, this can be avoided using the FMA operation.

   - The FMA operation first computes a double-width product $A \times B$ during the multiplication, i.e. if A, B are FP64 numbers, then $A \times B$ is a FP128 number
   - $A \times B$ thus has a 106-bit significand, whilst C has a 53-bit significand
   - Therefore, even if subtractive cancellation occurs during the addition of $(A \times B)$ and C, there remains at least 53 valid bits of precision, resulting in no loss of precision in the final result

5. Nvidia GPUs provide native hardware support for both FP32 and FP64 FMA operations, and using them is faster and more accurate than performing separate multiply and add operations.

6. By default, the Nvidia CUDA compiler (NVCC) optimises CUDA programs by merging successive FP multiply and add operations into FMA operations.
   - This FMA merging optimisation may be explicitly disabled by using the CUDA compiler flag `--fmad=false`

7. There is no standardised accuracy level for FP mathematical functions implemented by different libraries. Functions compiled for the host (`__host__` functions) use the host compiler's math library whereas functions compiled for the device, i.e. `__global__` and `__device__` functions, use the CUDA math library implementation.
   - Since we make use of the `sqrt` function when checking particle-particle collisions, this introduces deviations between our OpenMP and CUDA implementations

4.3 Floating-point Hardware Comparison

The hardware support for double-precision FP64 numbers and FP operations also differ between modern CPUs (that support the AMD64 ISA) and GPUs.

1. CPUs possess 64-bit GPRs, whereas many GPUs only possess 32-bit registers. Therefore, whilst CPUs store FP64 numbers in a single GPR, GPUs store FP64 numbers in a pair of adjacent registers.

2. CPUs have specialised FP units (FPUs) as coprocessors for performing scalar and vector FP64 computations natively.

- Scalar FP64 operations (specifically x87 instructions) are performed with an 80-bit extended precision format to minimise precision loss from overflow and rounding of intermediate values, before the results are rounded and stored in GPRs
- Vector FP64 operations are performed on extra-wide registers (albeit at lower precision than x87 instructions), via support for one or more of the following supplementary instruction sets: *Advanced Vector Extensions* (AVX) or *Streaming SIMD Extensions* (SSE)
  - These extensions also provide instructions for scalar FP64 operations on extra-wide XMM registers
- Some CPUs also possess hardware support for FMA operations, by providing an FMA instruction from the FMA instruction set to compilers

3. In contrast, GPUs emulate FP64 operations by decomposing them into a sequence of instructions (*subroutines*) on pairs of 32-bit registers, that are only performed by the FP64 units within each SM.

   - Consumer-grade Nvidia CPUs only possess two hardware FP64 units per SM, primarily to ensure that CUDA programs with FP64 code operates correctly (but not necessarily quickly) [2]
     - Thus, the throughput of FP64 instructions suffers heavily relative to FP32 instructions - for the Titan RTX and Tesla T4, the FP64 instruction throughput is $\frac{1}{32}$ that of its FP32 instruction throughput
   - Datacentre-grade Nvidia GPUs possess the full 32 hardware FP64 units per SM, allowing them to run CUDA programs with FP64 code with no performance loss
     - Since one FP64 number requires two registers to be stored, the final throughput of FP64 instructions is only half that of FP32 instructions
     - Amongst the GPU nodes in the Compute Cluster, only the Tesla V100 and Titan V have this 2:1 ratio of FP32 to FP64 TFLOPS throughput

4.4 Divergence Results

To investigate the rate at which the simulation state diverged, we ran the default testcase in `print` mode for both implementations. Additionally, the CUDA implementation was ran once with the flag `--fmad=true` and once with `--fmad=false`, to explicitly enable or disable the FMA merging optimisation respectively.

We used the simulation state of the OpenMP implementation as a reference, and considered the state of a particle to have *diverged* at a time step S if at least one of x, y, v_x or v_y differed at the 8^{th} decimal position.

Figure 7 below plots the number of diverged particles N_{div} against the step number S. Note that the horizontal axis only displays S to 180 instead of the full 1000 steps for the default testcase.

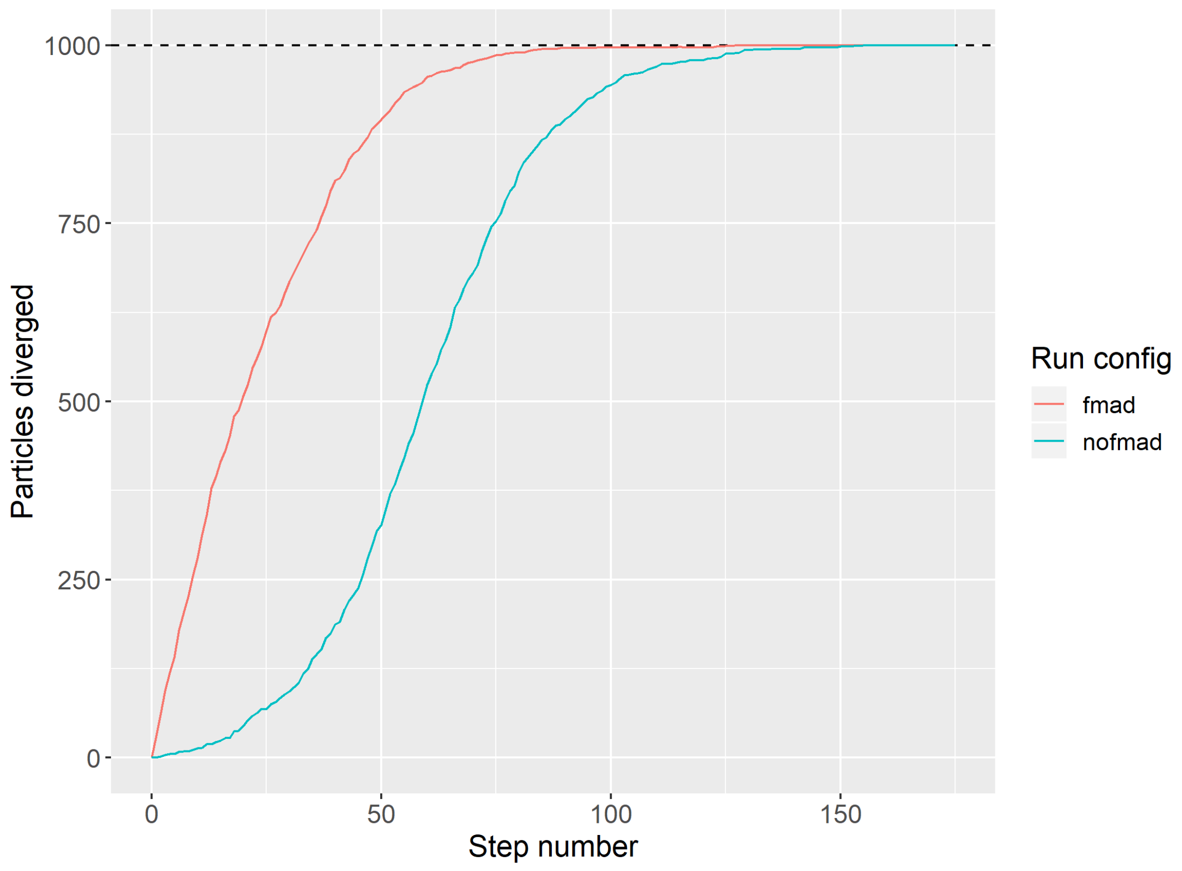


Figure 7: Plot of number of incorrect particles against time step

4.5 Correctness

Figure 7 shows that enabling the FMA merging optimisation leads to a faster divergence of the simulation state from the OpenMP implementation, with full divergence occurring at $S = 127$ as compared to $S = 155$. This demonstrates that discrete particle simulation is highly sensitive to small perturbations in the simulation state, with the large degree of chaos making it difficult to ascertain correctness.

Suppose we treat the simulation results from the OpenMP implementation as the "*correct*" result, then this runs counter to our expectation that the FMA operation is more numerically precise than separate FP multiply and add operations. Leaving the FMA optimisation enabled *should* result in better agreement with the intermediate simulation states of the OpenMP implementation, but this is not observed.

We inspected the assembly output of our OpenMP implementation, and found that the C compiler output scalar FP64 SSE2 instructions, without any higher-precision x87 or FMA instructions.

Reading the documentation of the GNU Compiler Collection (GCC) C compiler that is invoked by NVCC, we find that by default, the compiler **does not** generate vector or FMA instructions even if supported by the underlying hardware. Vector and FMA instructions must be specifically enabled with the sequence of compiler flags `-O2 -mavx2 -mfma`.

Despite re-compiling the OpenMP implementation with these flags and running the default testcase again, the simulations still exhibited rapid divergence that was complete at about $S = 150$. Given the numerous differences between the mode of FP computations between CPUs and GPUs, we conclude that there is no deterministic way to determine correctness of either implementation. We recommend that visual inspection of the simulation state at every step be employed instead.

5 Test Conditions and Testcases

5.1 Test Setup

For ease of deployment, we wrote a Bash script to automate the compilation, test case generation, profiling and transfer of results back to Sunfire with different sets of source code. The scripts can be found in the folder `/testDispatcher`.

Benchmarking of our previous OpenMP (parallel) implementation was done on the Intel Xeon Silver 4114 node, specifically soctf-pdc-010. We ran the program on 20 threads, which we found to produce the highest speedup with respect to our sequential implementation. [3]

Benchmarking of our CUDA implementations was done on the Compute Cluster nodes, specifically node xgpe1 with two Xeon Silver 4116 CPUs and a Nvidia Titan RTX GPU, and node xgpf1 with two Xeon Silver 4116 CPUs and a Nvidia Tesla T4 GPU.

The CUDA implementations were also benchmarked on the Jetson TX2 node, specifically jetsontx2-01. However, due to the excessively long runtime for small testcases, the larger testcases with $N > 6000$ were skipped.

A Python script was used to generate the input files for each of the implementations. Each testcase was run five times and the fastest execution time was retained as the datapoint for that testcase.

To replicate our results for a given implementation, run **`./test.sh`** in the particular implementation's folder. It is recommended to change the target folder for the **`scp`** command as it will automatically transfer all the results out when it is complete.

5.2 Random Testcases

For benchmarking, testcases ran in *perf* mode and the initial states of particles were not provided. Variables of the simulation were adjusted for each testcase.

The simulation parameters of the default testcase are

- $N = 1000, L = 20000, r = 1, S = 1000$

The testcases that were executed for each implementation are as follows.

1. CPU (parallel) implementation - 20 OpenMP threads
   - Varying N only: $N = 1k, 2k, 3k, 4k, 6k, 8k, 12k, 16k, 24k, 32k$
2. GPU implementation (Jetson TX2)
   - $F =$ `--fmad` compile flag
   - Varying both N and F together
     - $N = 1k, 2k, 3k, 4k, 6k$
     - $F =$ `true, false`
3. GPU implementation (Nvidia Titan RTX & Nvidia Tesla T4)
   - $F =$ `--fmad` compile flag
   - Varying both N and F together
     - $N = 1k, 2k, 3k, 4k, 6k, 8k, 12k, 16k, 24k, 32k$
     - $F =$ `true, false`

6 Execution Results

All plots were generated with the help of R. Some of the plots are not reproduced here for brevity.

The processed data is available in the submission as `.csv` files in `/data/gpu`. Raw data files from the **`time`** and **`nvprof`** command are available in `/data/gpu/<GPUtype>` as text files with `.time` extension and no extension respectively. The **`time`** command was used in place of **`perf stat`** on the Compute Cluster nodes since it was not available - however, the wall-clock time reported by both do agree.

All mentions of *speedup* of the CUDA implementation in this section is made with reference to the execution time of the OpenMP implementation with 20 threads, since running the large testcases on the purely sequential implementation would take too long.

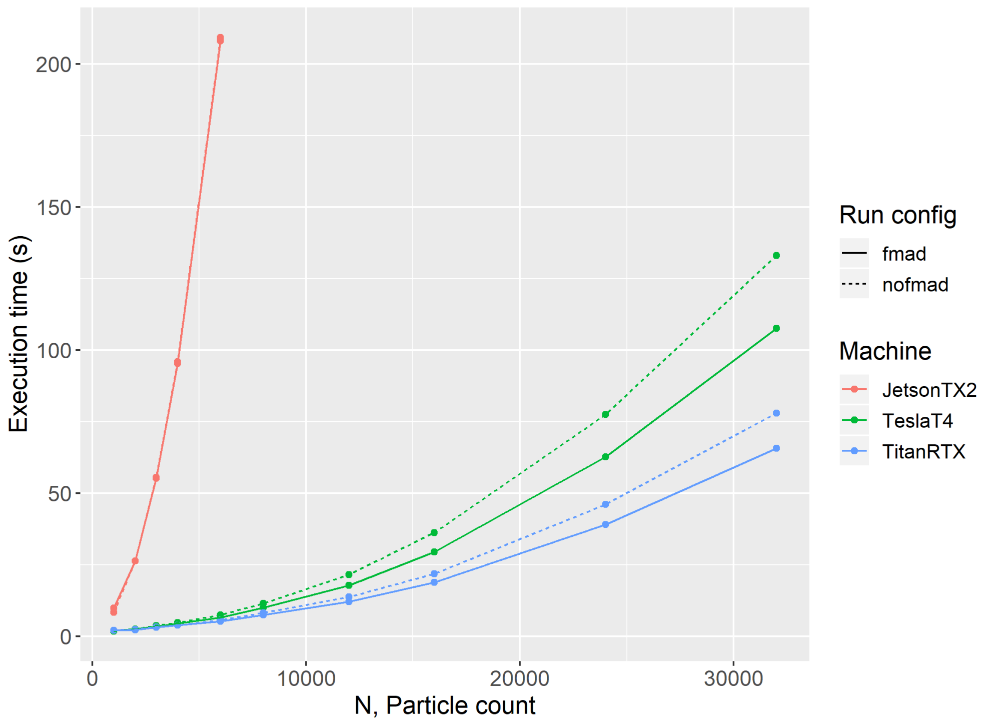


Figure 8: Plot of execution time against particle count, N (including Jetson-TX2)

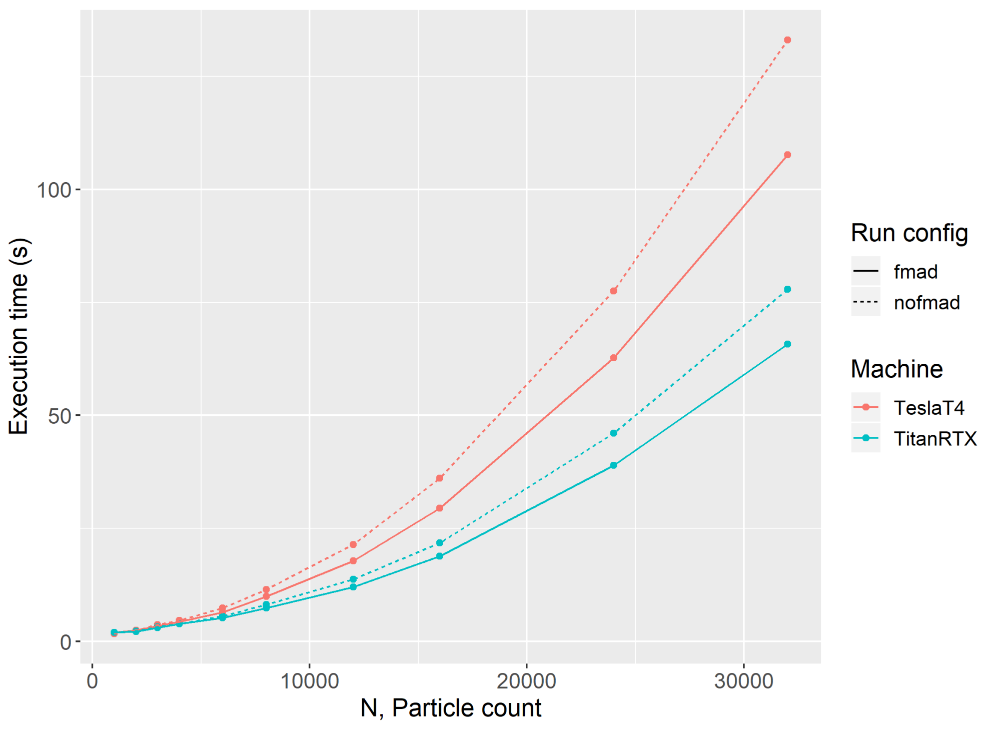


Figure 9: Plot of execution time against particle count, N

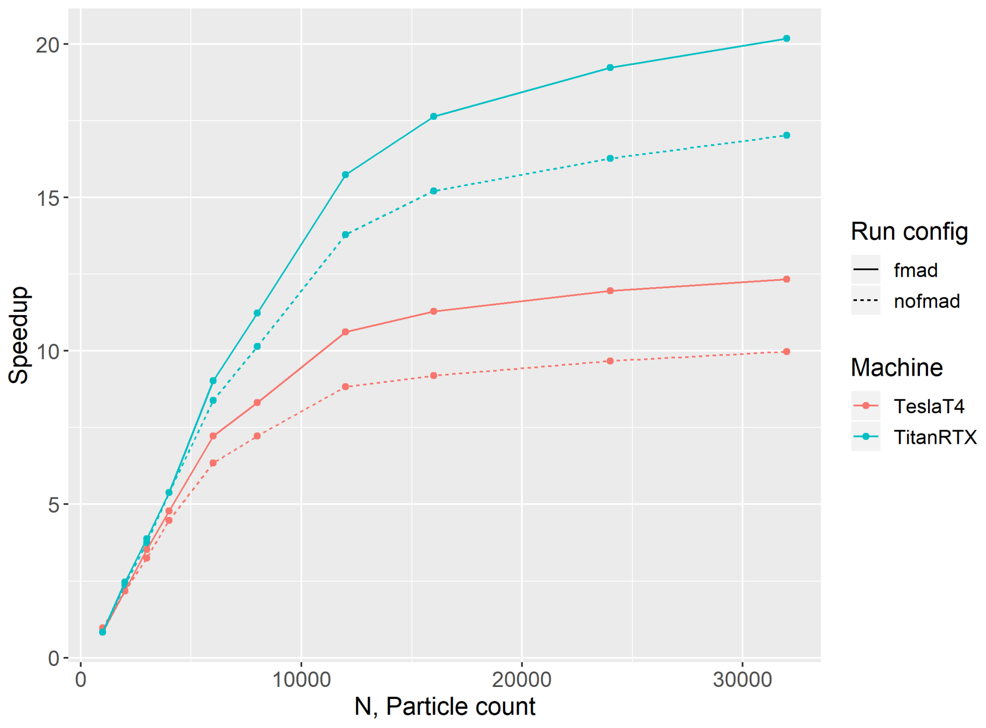


Figure 10: Plot of speedup against particle count, N

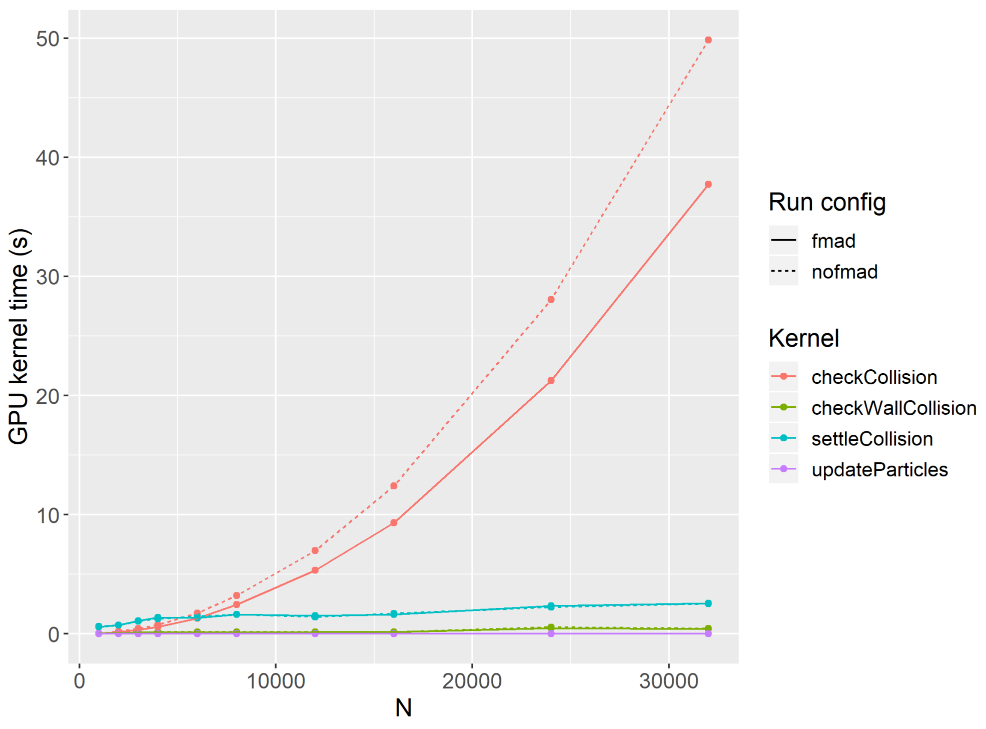


Figure 11: Plot of kernel time against N (all kernels)

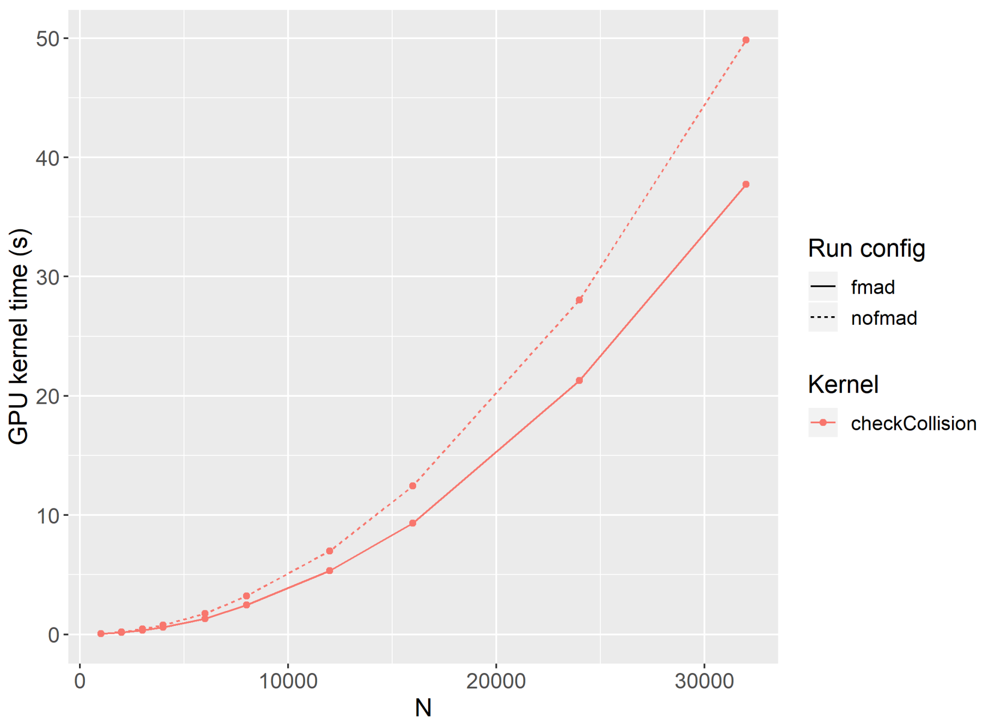


Figure 12: Plot of kernel time against N (only `checkCollision`)

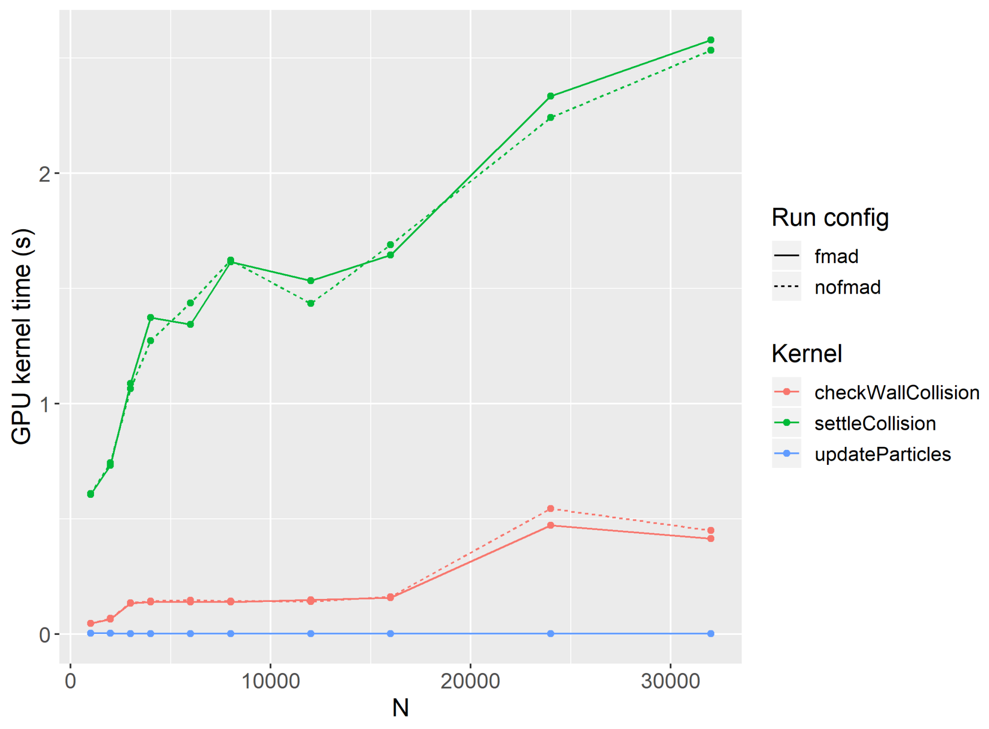


Figure 13: Plot of kernel time against N (all other kernels)

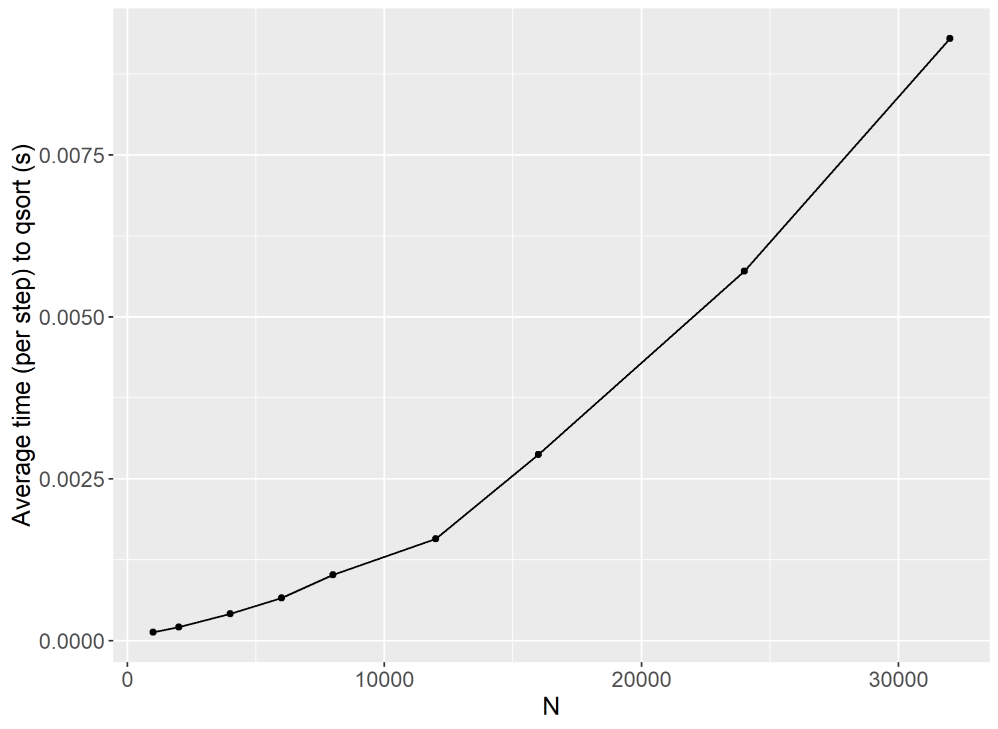


Figure 14: Plot of average time of `qsort` and filtering per step against N

7 Discussion: CUDA Implementation

7.1 Expectations

We did not expect our CUDA implementation to perform significantly better than the OpenMP implementation, due to the large degree of warp divergence from the many conditional statements required in a discrete particle simulator. In theory, such a computation should be very inefficient on the GPU since divergent branches are executed serially by threads in a warp.

7.2 Observations and Comparison with OpenMP Implementation

From Figure 8, we see that the execution time of the CUDA implementation maintains the same $\Theta(N^2)$ order-of-growth as the OpenMP implementation, for all GPUs and run configurations. This is expected, since we expect the runtime to be dominated by the cost of checking $\Theta(N^2)$ particle-particle collision pairs.

Figure 10 shows that the CUDA implementation still exhibits a significant speedup over the OpenMP implementation with 20 threads. We observe that the speedup initially grows linearly with N up to some value N_{crit}, beyond which it quickly tapers off and displays a logarithmic nature beyond that. This is true for both run configurations, and is a clear demonstration of the rapid diminishing returns predicted by *Amdahl's law*. Additionally, N_{crit} was larger for the Titan RTX, which is expected as the Titan RTX has 72 SMs (4608 CUDA cores) as oppposed to the 40 SMs (2560 CUDA cores) of the Tesla T4.

Comparing run configurations (value of the `fmad` compiler flag), Figures 9 and 10 clearly demonstrate that the (default) `fmad` program also seems to outperform the non-`fmad` variant. This behaviour occurs as the merging of successive FP multiply and add instructions into FMA instructions is an optimisation employed by NVCC to improve the precision and speed of the program. See Section 4.2 for more details on precision. FMA instructions coalesce a FP multiply and an add instruction into a single instruction, saving clock cycles in the process.

Analysing the time taken by each kernel in Figure 11, we see that the `checkCollision` kernel exhibits the expected $\Theta(N^2)$ order-of-growth of runtime from having to compute $O(N^2)$ collisions. The `checkCollision` and `settleCollision` kernels exhibit some growth with increasing N (albeit not linearly as expected), whereas the trend for `updateParticles` was not discernible.

Interestingly, we note that the `--fmad` flag only had an effect on the execution time of the `checkCollision` kernel, and did not meaningfully affect the execution time of the other three kernels. We propose two reasons for this:

- The quadratic order-of-growth of runtime of this kernel makes the effect of the FMA optimisation easier to observe at large N, and
- the `checkCollision` kernel is primarily computation with a smaller proportion of branching statements, relative to the other kernels.

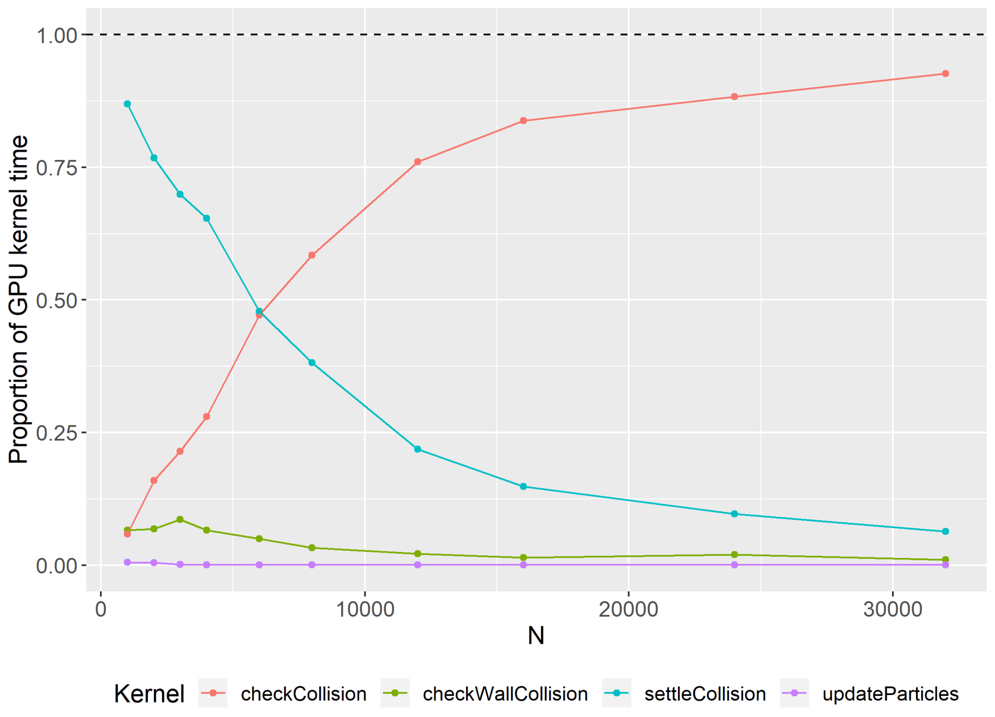


Figure 15: Plot of proportion of GPU runtime against N for the four different kernels

Comparing the fraction of total GPU time spent executing each kernel from Figure 15, we see that the `checkCollision` kernel dominates quickly with increasing N, in line with expectations. More interestingly, the `settleCollision` kernel dominates the overall execution time for smaller N, reaching parity with with `checkCollision` at about $N = 6000$ and decreasing in contribution thereafter. This behaviour is likely due to the amount of computation required to resolve a collision being larger (more conditional statements) than that required to check a potential particle-particle collision. The remaining kernels perform relatively modest work and are completed relatively quickly regardless of N.

Of particular note is the `qsort` and filtering that we did not parallelise, opting for it to be performed on the host instead. We see that the time taken to `qsort` and filter appears to grow quadratically with N; the impact on the entire algorithm cannot be understated. For $N = 32000$ on the Titan RTX, it took on average 8ms for the host to complete the task - or a total of 8s overall throughout the entire simulation, which is almost one-sixth the total execution time.

This behaviour can be explained. As the box size remains constant but N grows, we expect the number of particle-particle collisions (of which there are $\Theta(N^2)$) to increasingly dominate over the number of particle-wall collisions (only of $\Theta(N)$), since each particle now intersects the trajectory of more particles. This is empirically confirmed by Figure 16 below, which plots the ratio of p-p to p-w collision in the collision candidates array.

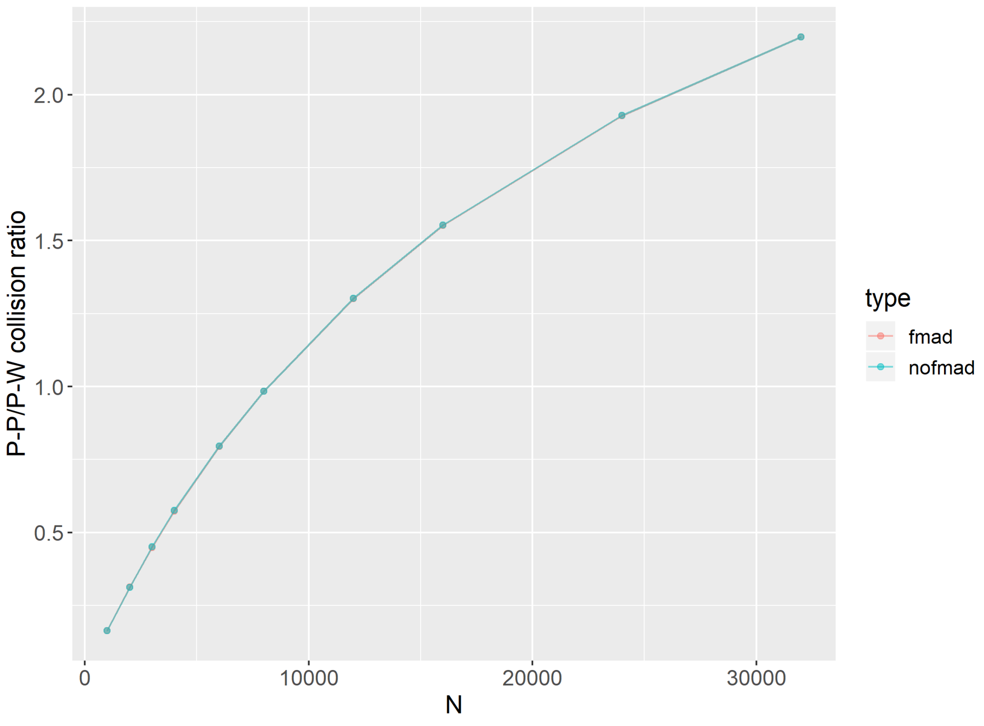


Figure 16: Plot of ratio of particle-particle collisions to particle-wall collisions against N

Therefore, we expect the number of total collision candidates N_{cand} to grow approximately quadratically with N, i.e. for large N, $N_{cand} \approx kN^2$. Since the runtime of quicksort is $\Theta(MlgM)$ to sort M items, the final time complexity is $\Theta(N_{cand}lgN_{cand}) = \Theta(N^2lgN)$.

7.3 GPU Comparison

The Jetson-TX2 was computing at an extremely slow rate and the test cases past $N = 6000$ were skipped in the interest of time. While the Jetson-TX2 has an extremely unique architecture of having a shared device and host memory, it is not scalable to large problem sizes and its benefits, if any, were not demonstrated for this discrete particle simulation.

Considering the two main GPUs, the Titan RTX significantly outperforms the Tesla, which is consistent with the computational capability of the GPUs. The Titan RTX has a total of 4608 CUDA cores across 72 SMs as opposed to the Tesla T4's 2560 CUDA cores with only 40 SMs, for a total of 80 more CUDA cores.

Coupled with the higher base and boost clocks of the Titan RTX, it can perform significantly more computations per second as compared to the Tesla T4, with the rated FP64 performance of 509.8 TFLOPS being almost twice that of the Tesla T4's 254.4 TFLOPS. This behaviour is observed in Figure 9, where the delta between the execution time of both GPUs increases with increasing N, with the Titan RTX being almost twice as fast when $N = 32000$.

8 Discussion: FP32 Variant

8.1 Expectations

Following words of wisdom, we attempted to "trade precision for speed" [1]. We expect the kernels to execute much faster given that the GPUs provided to us are optimised to perform single-precision FP32 operations (32 times more hardware FP32 than FP64 units) instead of double-precision FP64 operations. Whilst we understand that this will decrease the overall accuracy of the simulation and that the much larger magnitude of FP errors will snowball in this inherently chaotic system, it may be a viable alternative in the real world to produce an algorithm that is "nearly there".

8.2 Implementation and Testcases

The source files were modified in that all occurrences of '`double`' in the source files were replaced by '`float`'. The test files remained unchanged from Section 5.2.

8.3 Results

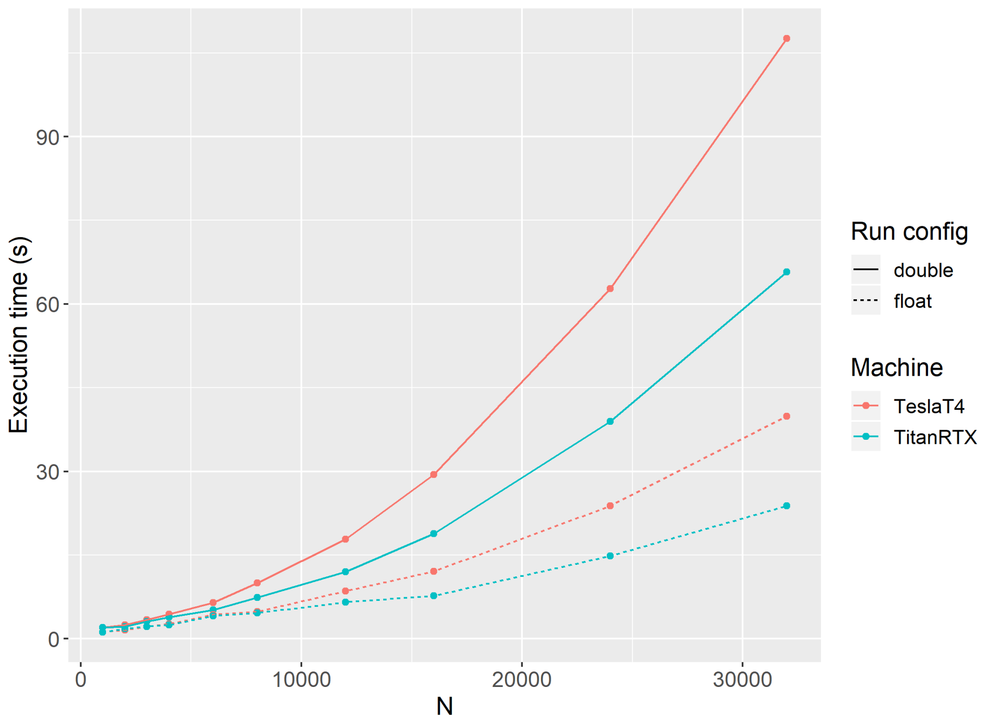


Figure 17: Plot of runtime (s) against N for `float` and `double` programs (`--fmad=true`)

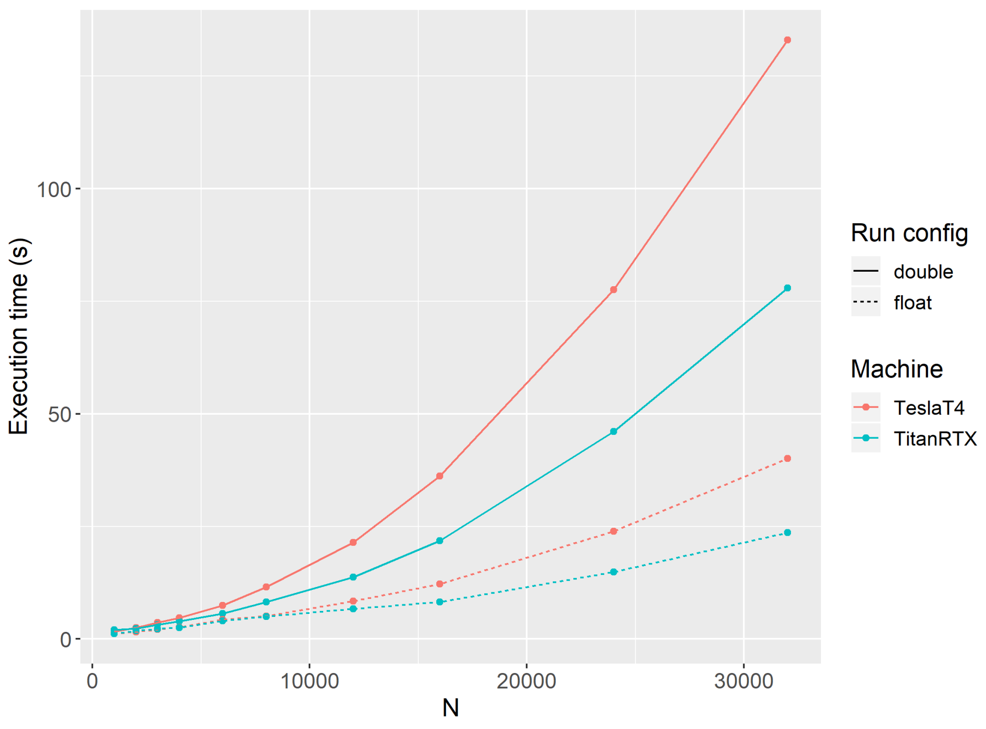


Figure 18: Plot of runtime (s) against N for `float` and `double` programs (`--fmad=false`)

While it is clear that the implementation using `floats` outperformed the one using `doubles`, the measured speedup fails to reach the factor of 32 that we had expected (from comparing the rated FP32 to FP64 throughput in TFLOPS).

Instead, the change only resulted in a speedup of between $2-3$. We propose two reasons for this deviation:

- The overhead from synchronisation required to add new collision candidates becomes significant for large N, since many more threads are attempting to increment the global counter `numCollisions` with `atomicAdd`
  - Multiple threads (within and across warps) contending to perform the same operation on a single memory address leads to serialisation of the operations, exacerbating the delay of atomic operations
- The GPU memory subsystem may become a bottleneck for FP32 computations since we opted to use managed memory - most of the data (particle states and collision candidates) resides in the GPU's global memory, which is slow to access and becomes increasingly poorly cached as N increases

9 Discussion: Chunk-size Variants

We also tried varying the chunk sizes (number of threads in each block) of the various different kernels in the program. We varied the parameters separately for the 1D kernels (`checkWallCollision`, `settleCollision` and `updateParticles`) and the 2D kernels (`checkCollision`).

9.1 Testcases

The simulation parameters of the modified default testcase are

- Machine: Titan RTX (`xgpe1`)
- $\mathbf{N = 2000}, L = 20000, r = 1, S = 1000$

1. GPU implementation
   - C_1 = chunk size (threads per block) of 1D kernels
   - C_2 = chunk size (threads per block) of 2D kernels
   - F = `--fmad` compile flag
   - Varying both N, C_1, C_2 and F together
     - $N = 2k, 3k, 4k, 6k, 8k, 12k, 16k, 24k, 32k$
     - $C_1 = 32, 64, 96, 128, 192, 256$
     - $C_2 = 32, 64, 96, 128, 192, 256$
     - $F =$ `true, false`

9.2 Results

The varying of chunk sizes yielded mostly disappointing results, with little to no variance (within the standard error) across runs. The execution times of runs with varying C_1 and constants $N = 32000, C_2 = 64, F =$ `true`, as well as varying C_2 and constants $N = 32000, C_1 = 64, F =$ `true` are illustrated in Figures 19 and 20.

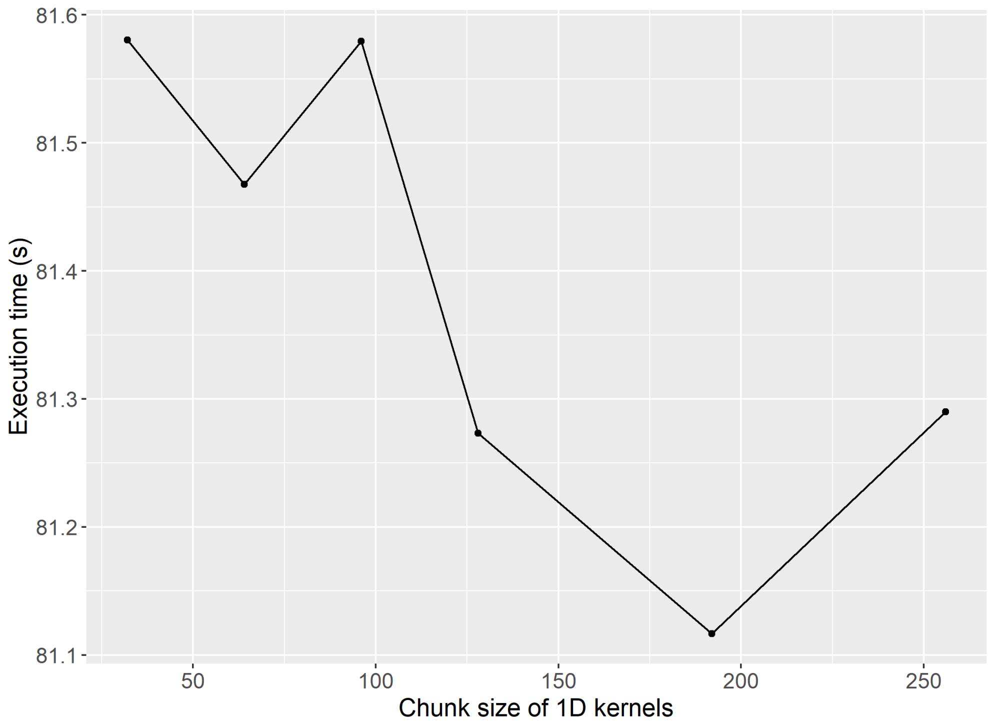


Figure 19: Plot of runtime (s) against C_1, 1D kernel chunk size ($N = 32000, C_2 = 64, F =$ `true`)

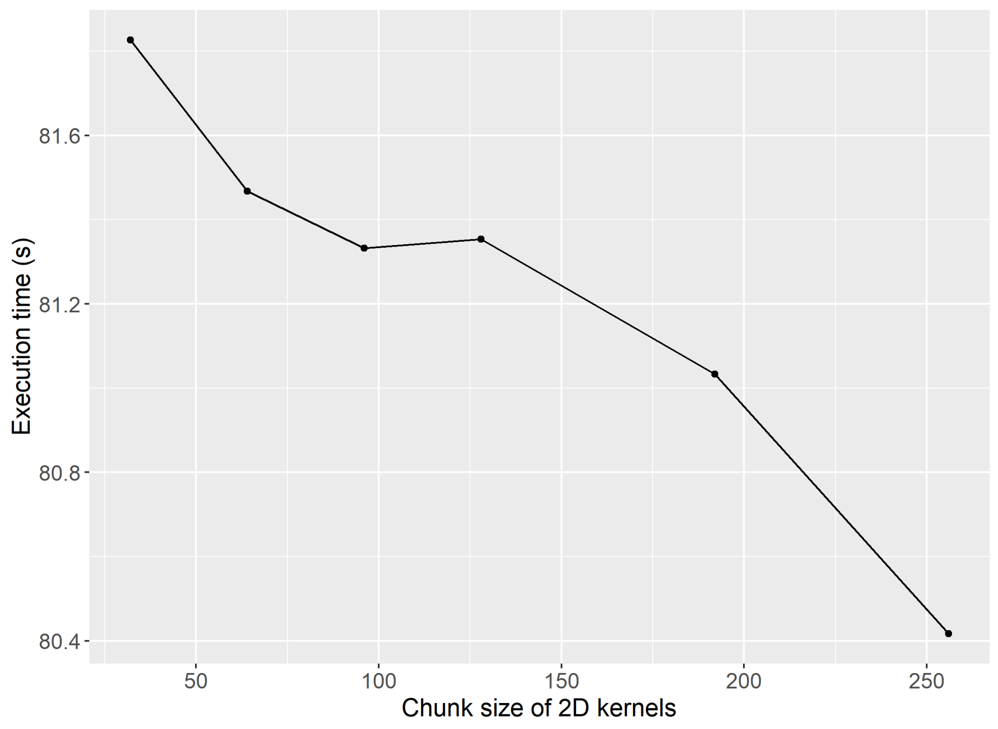


Figure 20: Plot of runtime (s) against C_2, 2D kernel chunk size ($N = 32000, C_1 = 64, F = \texttt{true}$)

A decreasing trend is observed in runtime against increasing C_2 but no discernable pattern is observed for runtime against increasing C_1. The performance increase with increasing C_2, however, is very small as compared to the total runtime and we cannot conclude that it has significantly sped up the program.

10 Other Minor Optimisations

10.1 Cache Preference Optimisation

Both the Titan RTX and Tesla T4 are built on Nvidia's Turing microarchitecture [2], where the 64KB of fast memory on each SM is split between the L1 cache and shared memory. Since our CUDA implementation does not make use of shared memory in any meaningful way, it was possible through the CUDA Runtime API to prioritise this fast memory for the L1 cache over shared memory use [4].

We modified our program to insert one line in the host function `__host__ simulate` at line 131 (before any kernels are launched):

- `cudaFuncSetCacheConfig(cudaFuncCachePreferL1);`

Light testing of the default testcase with N modified to $8000, 16000$ and 32000 showed that the modified implementation ran about $1 - 2\%$ faster, which was notable but not statistically significant due to run-to-run variance.

11 References

[1] C. Carbunaru. CS3210 (AY19/20 Sem 1) Lecture Slides, L06-CUDA. "Maximise Instruction Throughput" (Slide 62).

[2] E. Kilgariff, H. Moreton, N. Stam, and B. Bell. NVIDIA Turing Architecture In-Depth. https://devblogs.nvidia.com/nvidia-turing-architecture-in-depth/. Accessed: 2019-10-26.

[3] K. Loo and R. Lee. CS3210 Assignment 1 - Particle Movement Simulator (Part 1), 2019.

[4] NVIDIA. CUDA Execution Control. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__EXECUTION.html. Accessed: 2019-10-26.

[5] N. Whitehead and A. Florea. Floating point and IEEE-754 Compliance for NVIDIA GPUs, 2017.