Contents

Chapter 1

Introduction

1.1 Previous Work

1.2 Project Goals

Chapter 2

The Discrete Element Method

2.1 Forces

2.2 Collision Detection

It is most efficient to have the control volumes as small as possible because this reduces the number of particles in neighbouring control volumes. Control volumes must be at least as large as the largest particle to ensure that neighbouring control volumes contain all possible collisions. For monodisperse particle populations this means that the control volumes should be the same size as the particles. For polydisperse particle populations this means that the control volumes should be the size of the largest particle in the population. As mentioned in this can decrease efficiency for statistical distributions of particle sizes.

2.3 Implicit/Explicit

2.4 Rotation/Quaternions

Chapter 3

OpenCL and Graphics Processing Units

Chapter 4

Python Implementation

4.1 Overview

An initial implementation of the Discrete Element Method has been done in Python. The objective of this implementation is to gain an understanding of the DEM and any inherent computational difficulties. Python has been chosen as a testing environment for its simplicity and ease of development.

4.2 Element Types

Different element types are required for different types of geometry and particle. For the Python implementation the two simplest have been chosen, a spherical particle and an axis-aligned wall.

4.2.1 Particle

The basic particle element is a spherical particle with pre-determined properties. These properties include initial position, initial velocity, diameter, density, fluid viscosity, and functions for fluid velocity and gravity. All of these properties can be set upon instantiation of each particle object and so can be easily modified for a variety of different simulations.
There are two objects for particles, the main object, 'Particle', tracks a full particle state history which is very memory intensive and unnecessary for most applications. The second object, 'LowMemParticle', inherits from 'Particle' and only keeps track of the current state and, during iteration, one future state.
The particle is iterated using the function call 'Particle.iterate()'. This passes a Δt to the particle object and iterates the velocity and position. The methods used for integrating these properties are discussed in section **??**.

4.2.2 Axis-Aligned Simple Wall

The basic wall element is an axis-aligned simple wall. This object is defined by two points, minimum and maximum, that must lie in the same plane. From them a rectangle is formed. A normal is calculated for the wall and stored in the object to save time in collision calculations. The wall is treated as fixed, eliminating the need for complex material properties or calculation of motion.

4.3 Collisions

4.3.1 Collision Detection

Broad phase collision detection uses the simple spatial zoning technique described in Tuley[**?**]. This approach has been chosen because it is quick and simple to implement. Other options were considered for this implementation but the benefits of using them were far outweighed by the complexity that using them would add to the overall algorithm. Since the initial Python implementation will not be fast anyway it was not deemed necessary to implement optimised algorithms at this stage.
The domain is represented by a three dimensional array where each entry is a control volume. The control volume is a list of particles in its bounds. The list of particles is iterated over and each particle assigns itself to the correct control volume. This results in a three dimensional array where each control volume has all of the particles within its bounds as an array. Collision objects are then created for each pair of particles in the same,

or neighbouring, control volumes. This approach reduces the problem from $O(N^2)$ to almost $O(N)$ as shown in figure **??**.

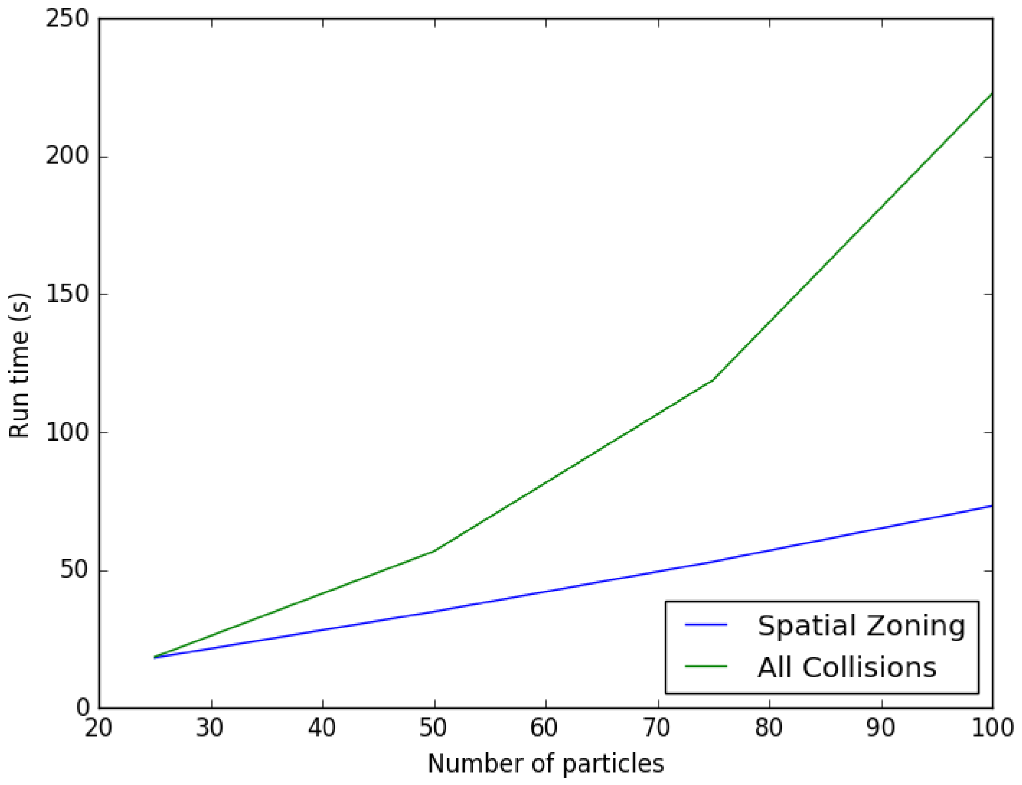

Figure 4.1: This graph shows that the simple spatial zoning technique reduces the problem from $O(N^2)$ down to almost $O(N)$.

4.3.2 Collision Resolution

After an array of collisions has been generated they are iterated over and each collision is resolved. First, the distance between particles is calculated to determine if they are in contact. Often this reveals that they are not in contact and the calculation ends there. If particles are in contact then collision forces are determined.
In this initial Python implementation only the simple normal and tangential contact forces are calculated. These are enough to run sufficient initial test cases.

4.4 Calculating Forces

There are three categories of forces used in this implementation. These are drag (particle-fluid interaction), gravity, and DEM forces.

4.4.1 Drag

The drag forces are determined using Stokes' law as discussed in This is calculated using a flow field calculation function that is passed into the Particle object upon instantiation. The particle then calls this function as part of its get_accel() function. This allows a variety of flow field functions to be used without modifying the Particle object code. The default for this function is a perfectly stationary flow.

4.4.2 Gravity

Gravity is treated in a similar manner to drag. A gravity function can be passed into the Particle object upon instantiation. Although this defaults to a simple -9.81ms^{-1}, it can be chosen to simulate a rotating frame of reference or other complex configurations. A rotating frame of reference has been implemented in the "gravity_shift_closed_box" example simulation.

4.4.3 DEM Forces

The DEM forces that are calculated in collisions are stored in an array within the Particle object. When the particle is iterated the array is added together and used in the integration calculation. After this calculation the array is cleared so that forces don't get incorrectly added multiple times. This configuration makes it simple to add and remove forces to the simulation whenever necessary and could also be used in general to add any force to the particle.

4.5 Numerical Integration

4.5.1 Velocity

Velocity is iterated with equation **??** where $\dot{u}$ is the acceleration obtained from the function call 'Particle.get_accel()'.

$$u_{n+1} = u_n + \dot{u}\Delta t \tag{4.1}$$

4.5.2 Position

Position is iterated with equation **??**.

$$x_{n+1} = x_n + \frac{u_{n+1} + u_n}{2}\Delta t \tag{4.2}$$

4.5.3 Method of Integrating Drag

There are three different methods of integrating drag. Firstly, there is the analytical solution, this is the exact solution to the model but cannot be done easily computationally. Secondly, there is the explicit numerical solution, this takes the current state and estimates the future state. Thirdly, there is the implicit numerical solution, this assumes the future state and integrates accordingly. For a simple system with constant flow speed and no gravity, the system acceleration is described by equation **??**.

$$\dot{u} = \frac{v - u}{\tau} \tag{4.3}$$

Equation **??** can be solved to show that the analytical solution for the particle speed, u, is that in equation **??**.

$$u(t) = v(1 - e^{-\frac{t}{\tau}}) \tag{4.4}$$

The explicit numerical integration method for equation **??** is that in equation **??**. Where u_n is the current speed and u_{n+1} is the speed after timestep Δt.

$$\frac{u_{n+1} - u_n}{\Delta t} = \frac{v - u_n}{\tau} \tag{4.5}$$

The implicit numerical integration method for equation **??** is that in equation **??**. Where u_n is the current speed and u_{n+1} is the speed after timestep Δt.

$$\dot{u} = \frac{u_{n+1} - u_n}{\Delta t} = \frac{v - u_{n+1}}{\tau} \tag{4.6}$$

Equation **??** can be rearranged to get an equation of the form $\dot{u} = f(u_n)$ as shown in equation **??**.

$$\dot{u} = \frac{v - u_n}{\tau + \Delta t} \tag{4.7}$$

When these three methods are applied to the system they produce the results in figure **??**.
Comparing the explicit and implicit numerical integration to the analytical solution shows that the explicit method has an average percentage difference of 2.05% and the implicit method has an average percentage difference of 1.85% when the timestep is $0.1s$.
The average percentage difference can be compared between the two methods with varying timesteps as shown in figure **??**. This graph shows that the explicit method increases its average percentage difference approximately linearly with increasing timestep. The implicit method increases its average percentage difference non-linearly at a slower rate than the explicit method. This implies that the implicit method is more accurate than the explicit method, especially for higher timesteps.

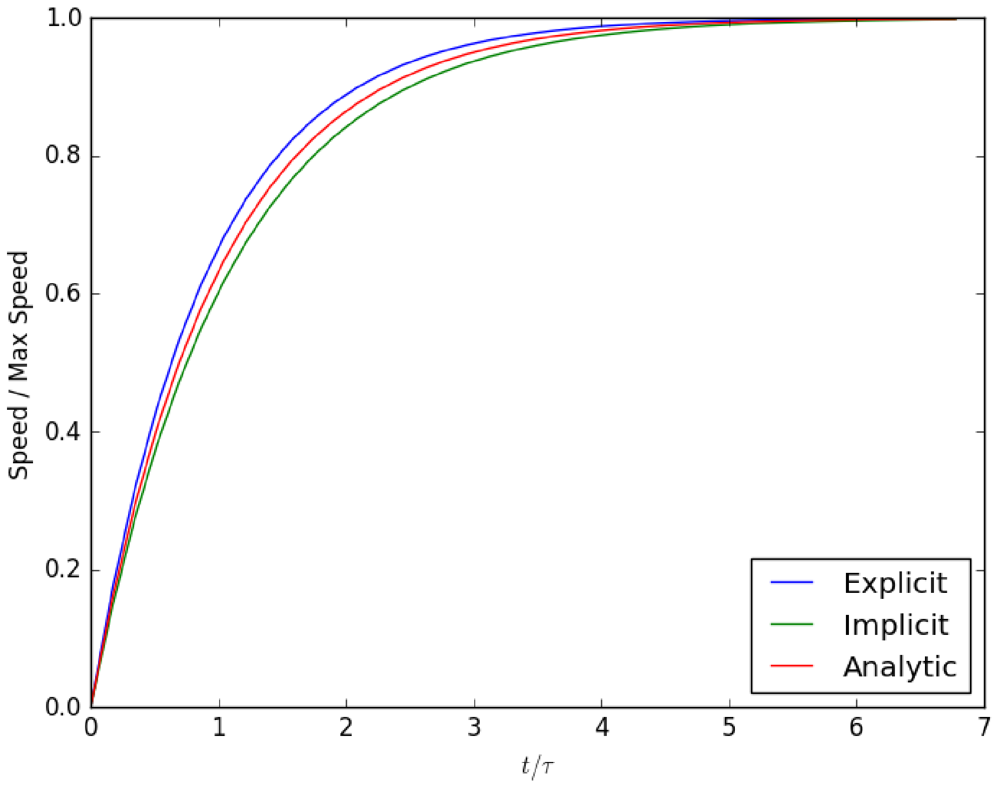

Figure 4.2: A graph of particle speed against time for the three methods of integration.

Unlike the explicit method, the implicit method depends on the other accelerations in the system. Equation **??** can be redefined to include these other accelerations as an extra $\dot{u}_e$ term as shown in equation **??**.

$$\dot{u} = \frac{u_{n+1} - u_n}{\Delta t} = \frac{v - u_{n+1}}{\tau} + \dot{u}_e \tag{4.8}$$

As before, equation **??** can be rearranged to be in the form $\dot{u} = f(u_n)$ as shown in equation **??**. The derivation of this can be found in appendix **??**.

$$\dot{u} = \frac{v - u_n + \tau\dot{u}_e}{\tau + \Delta t} \tag{4.9}$$

Equation **??** can be applied to a particle falling under the effect of gravity through a stationary fluid. The results are shown in figure **??**. As with the previous system, the explicit method has an average percentage difference of 2.05% and the implicit method has an average percentage difference of 1.85% when the timestep is $0.1s$. This shows that the accuracy of the modified equation is consistent with that of equation **??**.

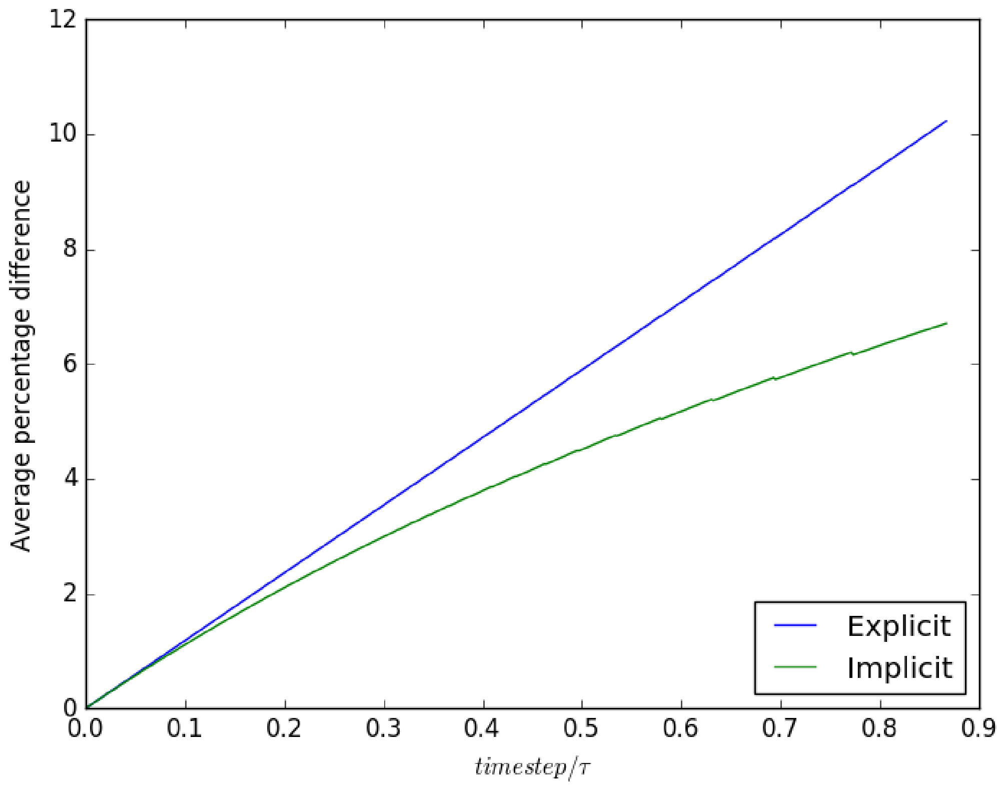

Figure 4.3: A graph of average percentage difference between the numerical method and analytical solution against varying timestep. The jaggedness of the lines is due to the discrete changes in timestep causing small rounding errors.

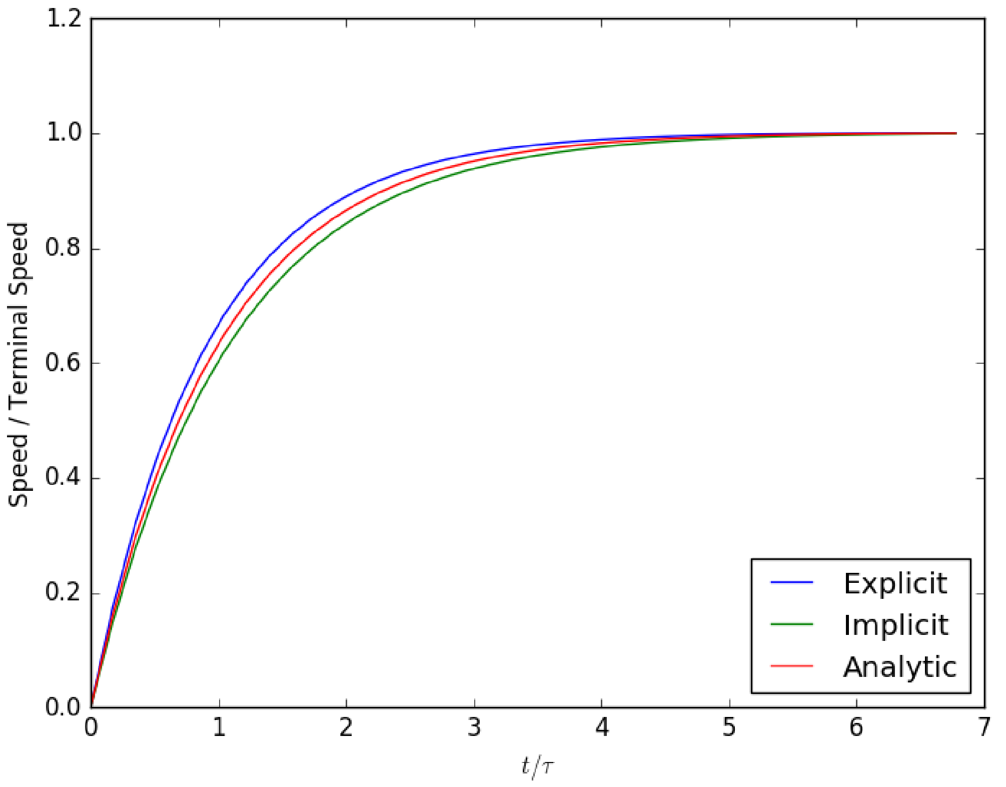

Figure 4.4: A graph of particle speed against time for each method of integration using equation **??**.

4.6 Verification

To asses the accuracy of the implementation a series of cases have been tested and compared to analytical solutions of the model.

4.6.1 Settling Overlap

In this case two particles are used. The first particle is an ordinary particle acting under the effects of gravity. The second particle, placed below the first particle, is a particle with quasi-infinite density without being affected by gravity. As time increases the first particle bounces on the second particle until it eventually comes to rest with some overlap with the second particle.

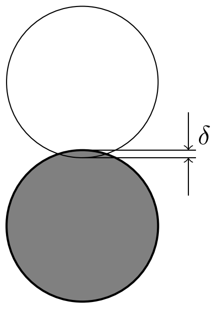

Figure 4.5: A particle, under the effect of gravity, resting upon a second particle of infinite density, unaffected by gravity. Overlap, δ, is labelled and exaggerated for clarity.

For the particle to be at rest the particle's gravity must be equal to the normal force from the DEM.

$$F_g = F_n \tag{4.10}$$

$$mg = k_e\delta - \eta u \tag{4.11}$$

To get an equation for overlap, δ, the equation is simplified and rearranged. The speed, u, is 0 at rest.

$$\delta = \frac{mg}{k_e} \tag{4.12}$$

Taking particle diameter to be 0.1 m and particle density to be 2000 kgm^{-3}, the particle mass is 1.047 kg. Gravity is taken to be 9.81 ms^{-2} and model spring stiffness is 10^5.

$$\delta = \frac{1.047 * 9.81}{10^5} = 1.027 * 10^{-4} \tag{4.13}$$

Running a simulation with these parameters also yields an overlap of $1.027*10^{-4}$. Comparing simulation results with high precision overlap prediction shows that the simulation result is within $2.1 * 10^{-11}$% of the predicted value.

Timestep stability

This result can also be compared for varying timesteps to asses the stability of the implementation. Figure **??** shows very stable results up until a timestep of approximately 0.00097.

4.6.2 Timestep Stability

Collisions go boom if timestep is too high etc. The stiffer the collisions the smaller the timestep must be to maintain physical results. This can be spotted and logged if E_k is higher after collision than before.

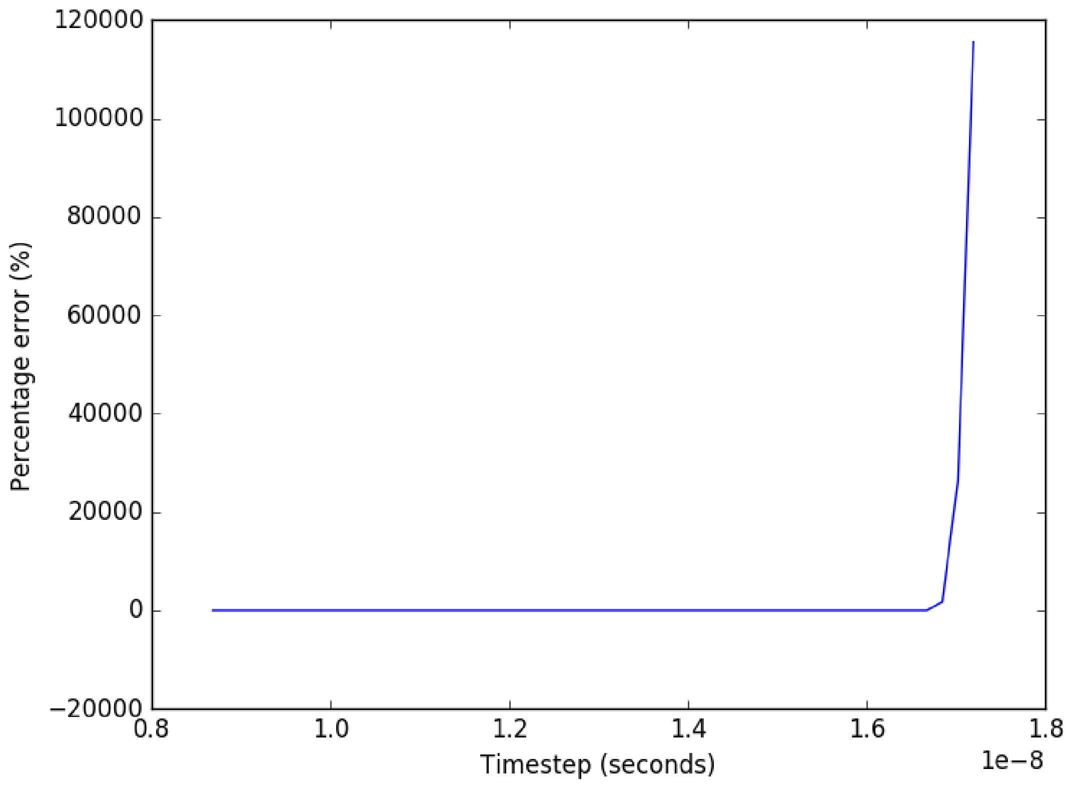

Figure 4.6: A graph of overlap percentage error against time step.

Chapter 5

OpenCL Implementation

5.1 Data Structures

As the C language does not support objects, the objects used in the Python implementation must be simplified into data structures and separate functions. Most of the calculations that were done in the object functions will now be done within the kernels.

5.1.1 Particle

5.1.2 Collision

5.1.3 Buffers

To access the data from the device it must be passed into a buffer. None of the structure data is passed into the buffer so the device must have the same definition of the structure as the host. This is problematic as the host and device have different compilers. To work around this problem the host and device structures are written with members in descending order of size. In addition padding has been added to the host code since the alignment specifier is apparently insufficient to ensure correct alignment. The alignment attribute specifier is also not the same between different compilers so "if defined" statements are implemented for both Visual Studio and GCC compilers. The code for these definitions is shown in figure **??**.

Host:

```
typedef struct {
    cl_float3 pos;
    cl_float3 vel;
    cl_float3 forces;
    cl_ulong particle_id;
    cl_float particle_diameter;
    cl_float density;
    cl_float fluid_viscosity;
    cl_char padding[56];
// Structure memory alignment for Visual Studio and GCC compilers.
#if defined(_MSC_VER)
} __declspec(align(128)) particle;
#elif defined(__GNUC) || defined(__GNUG__)
} __attribute__((aligned (128))) particle;
#endif
```

Device:

```
typedef struct {
    float3 pos;
    float3 vel;
    float3 forces;
    ulong particle_id;
    float particle_diameter;
    float density;
    float fluid_viscosity;
} __attribute__ ((aligned (128))) particle;
```

Figure 5.1: Host and device definitions of the particle structure.

5.2 Kernels

The main calculations for this implementation are performed on the device. This means that the program must be separated into kernels to be passed over sets of data.

5.2.1 Collision Detection

To improve efficiency, in both speed and resource usage, performing naive collision detection is not viable for large numbers of particles. To improve on this the spatial zoning technique is used similar to the initial Python implementation. However, C does not make arrays of varying sizes easy or efficient so the data structures used and the algorithm implementation must be significantly different.
The basic problem is how to store control volumes as lists of references to particles. In Python this was easy, a simple 3D array of control volumes with lists of particle objects inside was sufficient. Various approaches to solving this problem were considered. One approach was to encode particle IDs (equal to the index of a particle in the particles array) with a hashing function into a single number that could be turned back into particle IDs on the device. However, this approach was infeasible because the numbers would get so large that they could not be stored accurately or efficiently.
The approach used is to have multiple passes of assignment of particles to control volumes. The first pass simply counts how many particles are in each control volume. This is stored in a single dimensional array of control volumes represented by integers of how many particles each contains. From this array another array is created. This array is of all the particle IDs but sorted into control volumes. The control volumes are of lengths defined by the count array and start at indexes stored in a third array. If a control volume has no particles, the start index of the array is set to -1. For the unsigned long data type this overflows to the maximum value (approximately 4.3 billion) which will likely never be used to index particles. When creating collisions this assumption will be checked and a warning issued. This arrangement of arrays is shown in figure **??**.
This approach is somewhat similar to how memory is handled on a computer, but using indexes instead of pointers. For an entirely host-side method an array of pointers to arrays of particles could be used, but this would not be sensible when dealing with device memory as each array would need to be moved to the device before use. Having three arrays that hold all the necessary properties simplifies the memory buffer process significantly. The maths required for turning positions and control volume coordinates into indexes in these

arrays are contained in cvUtils.c and kernelUtils.cl for host and device, respectively.

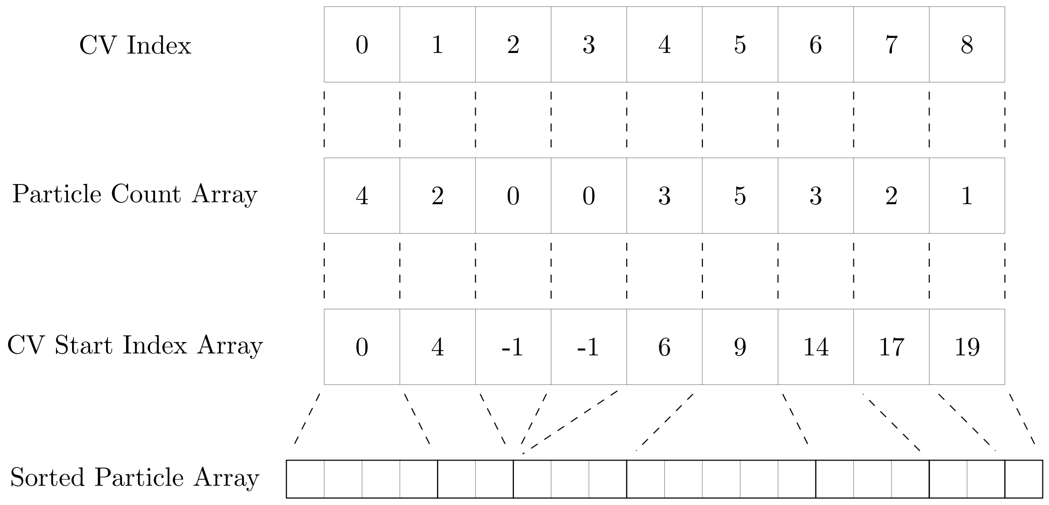

Figure 5.2: Diagram showing the structure and relationship between arrays representing Control Volumes.

5.2.2 Collision Resolution

The collision resolution kernel is actually separated into multiple kernels for different types of collision (particle-particle and particle-wall) however the behaviour of these kernels is almost identical. For this discussion we will use the particle-particle kernel as an example. The kernel takes a pointer to an array of collision structures and a pointer to the array of particles. The DEM collision force calculations are run for each collision and the forces are added to the DEM forces vector in the relevant particle structures. This approach has been chosen because it is easier to sum the forces as they are calculated rather than attempt to predict the length of the necessary force array to store each force separately as in the Python implementation.
This approach causes a serious problem as it is possible that multiple collision kernels will need to write data to the same particle at the same time. The solution to this is to use the atomic operations available in OpenCL. Unfortunately, OpenCL only natively supports full atomic operations for int and unsigned int data types. OpenCL does support an exchange atomic operator for single precision floats, but this is not the best approach for doing atomic arithmetic for floats.
An approach for doing atomic addition (as is necessary in this case) is presented in The code is shown in figure **??**. This approach uses the comparison exchange atomic operator by creating a union of the floats with unsigned ints. This works well because the bits being exchanged are the same for the float and unsigned int and actual atomic arithmetic is not necessary so the difference between the data types doesn't matter. Thus the new value is calculated and if the value used to calculate it has changed in that time the calculation is repeated with the updated value.

```
_INLINE_ void atomicAdd_g_f(volatile __global float *addr, float val)
  {
      union{
          unsigned int u32;
          float        f32;
      } next, expected, current;
       current.f32    = *addr;
      do{
          expected.f32 = current.f32;
          next.f32     = expected.f32 + val;
               current.u32  = atomic_cmpxchg( (volatile __global unsigned int *)addr,
                              expected.u32, next.u32);
      } while( current.u32 != expected.u32 );
  }
```

Figure 5.3: Kernel code for a solution to atomic addition with floats.

5.2.3 Particle Iteration

Chapter 6

Results

6.1 Comparison between CPU and GPU

Chapter 7

Conclusion

7.1 Further Work

Appendix A

Derivations

A.1 Equation ??

$$\dot{u} = \frac{v-u}{\tau} + \dot{u}_e \tag{A.1}$$

$$\frac{u_{n+1}-u_n}{\Delta t} = \frac{v-u_{n+1}}{\tau} + \dot{u}_e \tag{A.2}$$

$$u_{n+1} = \frac{\dfrac{v}{\tau} + \dfrac{u_n}{\Delta t} + \dot{u}_e}{\dfrac{1}{\Delta t} + \dfrac{1}{\tau}} \tag{A.3}$$

$$\frac{u_{n+1}-u_n}{\Delta t} = \frac{\left(\dfrac{\dfrac{v}{\tau} + \dfrac{u_n}{\Delta t} + \dot{u}_e}{\dfrac{1}{\Delta t} + \dfrac{1}{\tau}}\right) - u_n}{\Delta t} \tag{A.4}$$

$$\dot{u} = \frac{v - u_n + \tau\dot{u}_e}{\tau + \Delta t} \tag{A.5}$$