OCCLUDE: An OpenCL N-body code for collisional dynamics

AST1500 Research Project Report

Trevor Vincent

Supervisor
Dr. Hanno Rein

University of Toronto

Astronomy and Astrophysics

August 30, 2014

x

Abstract

This project report presents OCCLUDE, an open-source collisional n-body OpenCL code which uses a Barnes-Hut octree [1] to accelerate gravity calculations and collision detection on the graphics card. OCCLUDE was designed primarily to run shearing-sheet simulations of Saturn's dense rings, as these are of theoretical interest in astrophysics. Preliminary speed tests have shown that OCCLUDE is at least 2.5 times faster than REBOUND (an open-source CPU N-body code written with similar algorithms) and with some tweaking we expect to see double digit speedups on higher-end graphics cards. As far as the author is aware, this is first publicly available OpenCL N-body code that uses trees and detects collisions between particles. OCCLUDE can be downloaded at the author's github `https://github.com/trevor-vincent/occlude`.

Contents

1 Introduction

OCCLUDE fills a vacancy in the REBOUND N-body code — a lack of support for graphics card acceleration. But what is the REBOUND code anyway? REBOUND is a publicly available collisional N-body code written by Hanno Rein, Shangfei Liu and David S. Spiegel which can be downloaded at `https://github.com/hannorein/rebound`. REBOUND is written in C and is highly modular, with separate modules for gravity, collisions, boundary conditions and integrators, allowing easy customization for any problem type. In its most recent version, REBOUND can compute anything from long-term symplectic orbit integrations to shearing-sheet simulations of Saturn's rings. REBOUND uses OpenMP and MPI to utilize the extra processing power on shared and distributed memory systems respectively. However, REBOUND currently does not have support for computation on graphics card processing units (GPU). This presents a problem because the average consumer-grade graphics card now contains hundreds of processing cores which can potentially speed up code by a substantial factor. The primary goal of OCCLUDE is to rewrite the REBOUND modules using the OpenCL (Open Computing Language) API, an open-source API that binds to the C language and allows one to write code for any set of CPUs or GPUs (Intel, AMD, NVIDIA, etc.).

With a code that has support for GPU acceleration, such as OCCLUDE, we can then run N-body codes with higher densities on normal desktop workstations. This means we can investigate the physical phenomena in the densest sections of Saturn's rings with sheering-sheet simulations. In Section 2 of this project report we will delve into the motivation and background of the project, this includes an introduction to the phenomena seen in Saturn's dense rings, the shearing-sheet model, collisional N-body codes and a brief introduction to graphics card programming. In Section **??** we describe the OCCLUDE code in detail along with the tests that were done to examine it's accuracy and speed. Finally, we list the full OCCLUDE OpenCL kernels in the Appendix for convenience.

2 Background

2.1 Saturn's Dense Rings

The rings around Saturn represent one of the most enduring symbols in astronomy. We now know that these planetary rings are not solid objects, but composed of countless icy particles with sizes ranging from specks of dust to small moons kilometers across. Furthermore, instead of being a homogeneous set of icy clumps ranging in size, we now know that each one of Saturn's rings has different attributes, such as optical depth (density), gaps, waves, moonlets, mass and radius distributions, etc. The richness of the features in these rings found by Cassini and earlier probes leaves researchers with many still unanswered questions.

Saturn's rings can be broadly grouped into two categories: the dense A, B and C rings and the tenuous D, E and G rings (see Figure (6)). The primary concern of this project is Saturn's dense rings, which have optical depths greater than 0.1 and are predominantly comprised of particles larger than 1 cm [2]. This is the regime that will be easier to investigate on a desktop workstation with a GPU-accelerated code capable of large particle simulations. For Saturn's dense rings the important physical processes are the self-gravity of the ring particles, the dissipative collisions between ring particles, the motion of the ring particles in Saturn's gravity and the gravitational interaction with Saturn's moons and moonlets. We will touch on the first three since these are important in the simulations we will be doing with OCCLUDE.

With higher densities, the gravitational interactions between the ring particles, the "self-gravity", becomes significant. Analytical models usually do not treat self-gravity or treat it in a very approximate manner, making it a feature that must be studied using numerical codes. Without self-gravity, an equilibrium state for the ring particles is quickly reached as the viscous heating due to the Keplerian shear is balanced by the energy dissipation due to inelastic collisions. With high density environments, such as those in rings with high optical depth, self-gravity is no longer neglibible and the gravitational instability caused by it leads to gravitational wakes (See Figure (7)). These wakes are formed by colliding particles slowing down and forming gravitational bound opaque clumps that then disrupt again due to the shear stress caused by the differential rotation. The Cassini probe has

detected these wakes nearly everywhere in the dense A and B rings and they trail by 20° to 25° from the tangential direction. The self-gravity wakes cause disturbances around them

The high density also influences the collision rate and hence the mean free path of a ring particle. This leads to a component of pressure and momentum transport that is not dependent upon the ring particles position — a non-local component and this determines the stability properties of the ring's flow. A strong nonlocal contribution to viscosity, common in dense rings, is suitable for viscous overstability, a condition where the system is so stable, it develops axisymmetric waves with wavelengths on the order of 100m.

Features such as the above in Saturn's dense rings can only be analyzed with dense particle simulations, since it is within this regime that such phenomena appear. While GPU-acceleration will no doubt help in bringing down the computation time in dense ring simulations, there is a further simplification we can make when studying the rings — the shearing sheet model.

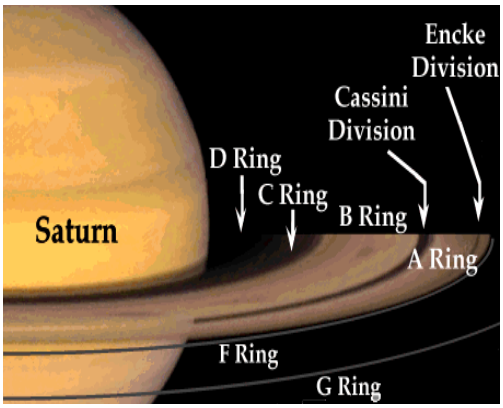

Figure 1: *A crude labelling of the Saturn ring system. This figure came from http://science.nasa.gov/science-news/science-at-nasa/2002/12feb_rings/*

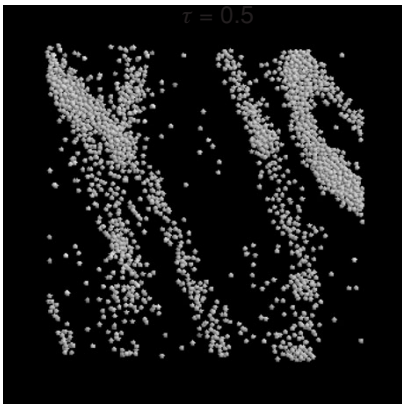

Figure 2: *Self-gravity wakes forming in N-body ring simulations. This figure came from Reference [2] .*

2.2 Shearing Sheet Model of Planetary Rings

The many unusual features of Saturn's ring call for explanatory models. The most straight forward way to explain such features would be to simulate the individual motions and interactions of each of the particles using Newton's

law's of motion on a computer. Unfortunately, the memory and speed of computers limits this approach to only considering a small number of ring particles, usually denoted N. These so-called N-body simulations, nowadays are limited to $N = 10^4$ to $N = 10^6$ particles depending on the computer system and the complexity of the interactions simulated. But this is only a small fraction of the number of ring particles in a planetary ring.

Modern N-body simulations of planetary rings get around this hardware limit by considering a small co-rotating patch within the rings, with certain periodic boundary conditions. This is the called shearing sheet model. Consider a box of ring particles orbiting at a distance a_0 from the planet, the Kepler velocity is $\Omega_0 a_0$. In a co-rotating the frame, the acceleration of a particle in this box is given by the usual equation

$$\frac{d^2\boldsymbol{r}}{dt^2} = -\nabla\Phi - 2\boldsymbol{\Omega} \times \frac{d\boldsymbol{x}}{dt} - \boldsymbol{\Omega} \times (\boldsymbol{\Omega} \times \boldsymbol{x}) \ . \tag{1}$$

Now, the potential is a combination of the potential due to the other ring particles in the patch and the potential due to the massive body M the ring particles are circling. That is,

$$\Phi = \Phi_{rp} + \Phi_M \ . \tag{2}$$

However, since the patch is far away from M, we can Taylor expand Φ_M as follows

$$-\frac{\partial \Phi_M}{\partial x_j}(\boldsymbol{x}, t) = -\Phi_j^M - \sum_k \Phi_{jk}^M(t) x_k + O(|x|^2) \ , \tag{3}$$

where $\Phi_j^M = \frac{\partial \Phi_M}{\partial x_j}\Big|_{\boldsymbol{x}=0}$ and $\Phi_{jk}^M = \frac{\partial \Phi_M}{\partial x_j \partial x_k}\Big|_{\boldsymbol{x}=0}$. Since the Φ_j term in the expansion will add a constant acceleration boost to every particle, we may ignore it by jumping to a frame accelerated by exactly this amount, thus we will drop this term. Thus,

$$-\nabla\Phi = -\nabla\Phi_{rp} - \sum_k \Phi_{jk}^M(t) x_k \tag{4}$$

Since the massive planet M is spherical and centered at $\boldsymbol{R}(t) = (-a_0, 0, 0)$, we may write $\Phi(\boldsymbol{r}, t) = \Phi(|\boldsymbol{r} - \boldsymbol{R}|)$ and it is straightforward to derive that

$$\begin{aligned} \Phi^M_{xx} &= \Phi''_M(a_0) = \Omega_0^2 + 2a_0\Omega_0\Omega'(a_0) \\ \Phi^M_{yy} &= \Phi^M_{zz} = \frac{\Phi'_M(a_0)}{a_0} = \Omega_0^2 \\ \Phi^M_{xy} &= \Phi^M_{xz} = \Phi^M_{yz} = 0 \end{aligned} \tag{5}$$

Substituting Equation Set (5), along with Equation (4) and $\boldsymbol{\Omega} = \Omega_0\hat{\boldsymbol{z}}$ into Equation (1), we get

$$\begin{aligned} \ddot{x} &= 2\Omega_0\dot{y} - 2a_0\Omega_0\Omega'(a_0)x - \frac{\partial\Phi_s}{\partial x} \\ \ddot{y} &= -2\Omega_0\dot{x} - \frac{\partial\Phi_s}{\partial y} \\ \ddot{z} &= -\Omega_0^2 z - \frac{\partial\Phi_s}{\partial z} \ . \end{aligned} \tag{6}$$

Using $\Omega'(a_0) = -\frac{3}{2}\frac{\Omega_0^2}{a_0}$ and collapsing the three equations into one, we get

$$\ddot{\boldsymbol{r}} = -2\Omega_0\hat{\boldsymbol{z}} \times \dot{\boldsymbol{r}} + 3\Omega_0^2(\boldsymbol{r} \cdot \hat{\boldsymbol{x}})\hat{\boldsymbol{x}} - \Omega_0^2(\boldsymbol{r} \cdot \hat{\boldsymbol{z}})\hat{\boldsymbol{z}} - \nabla\Phi_s \ . \tag{7}$$

These are the sheared sheet equations of motion and the corresponding Hamiltonian is

$$\begin{aligned} H(\boldsymbol{r}, \boldsymbol{p}) &= \frac{1}{2}\boldsymbol{p}^2 + \Omega_0(\boldsymbol{p} \times \boldsymbol{r}) \cdot \hat{\boldsymbol{z}} + \frac{1}{2}\Omega_0^2\left[\boldsymbol{r}^2 - 3(\boldsymbol{r} \cdot \hat{\boldsymbol{x}})^2\right] + \Phi(\boldsymbol{r}) \ . \\ &\equiv H_0(\boldsymbol{r}, \boldsymbol{p}) + \Phi(\boldsymbol{r}) \end{aligned} \tag{8}$$

There is a useful set of solutions to these equations of motion when $\nabla\Phi_s = 0$, called the epicycle approximation, which is approximately valid when the ring self-gravity is negligible. The epicycle approximate solutions are

$$
\begin{aligned}
x &= x_g + X cos(\Omega_0 t + \alpha) \\
y &= y_g - \frac{3}{2}\Omega_0 x_g t + Y sin(\Omega_0 t + \alpha) \\
z &= Z cos(\Omega_0 t + \alpha_z) \ ,
\end{aligned} \tag{9}
$$

where $2X = Y$ and X, Z, x_g, y_g, α and α_z are arbitrary constants determined by the initial conditions. These equations will be used in Section (3.2).

The shearing sheet model usually assumes a certain type of periodic boundary condition, known as the shearing-periodic boundary conditions. This boundary condition is based off the fact that when $\Phi_s = 0$, Equation (7) is invariant under the transformation

$$
\boldsymbol{r} \rightarrow \boldsymbol{r} + c_x(\hat{\boldsymbol{x}} - \frac{3}{2}\Omega_0 t \hat{\boldsymbol{y}}) + c_y \hat{\boldsymbol{y}} \ , \tag{10}
$$

for arbitrary constants c_x and c_y. Equation (7) is also invariant under Equation (10) when Φ is invariant under the same transformation.

To implement this boundary condition in a computer simulation, eight ghost patches surround the main patch of ring particles as shown in Figure (**??**). Each particle crossing a boundary is replaced by another particle with a position given by Equation (10) and a velocity shifted by $\delta v_y = \pm\frac{3}{2}\Omega L_X$ (which comes from the fact that the $\frac{d\Omega}{dx} = -\frac{3}{2}\frac{\Omega}{x}$.

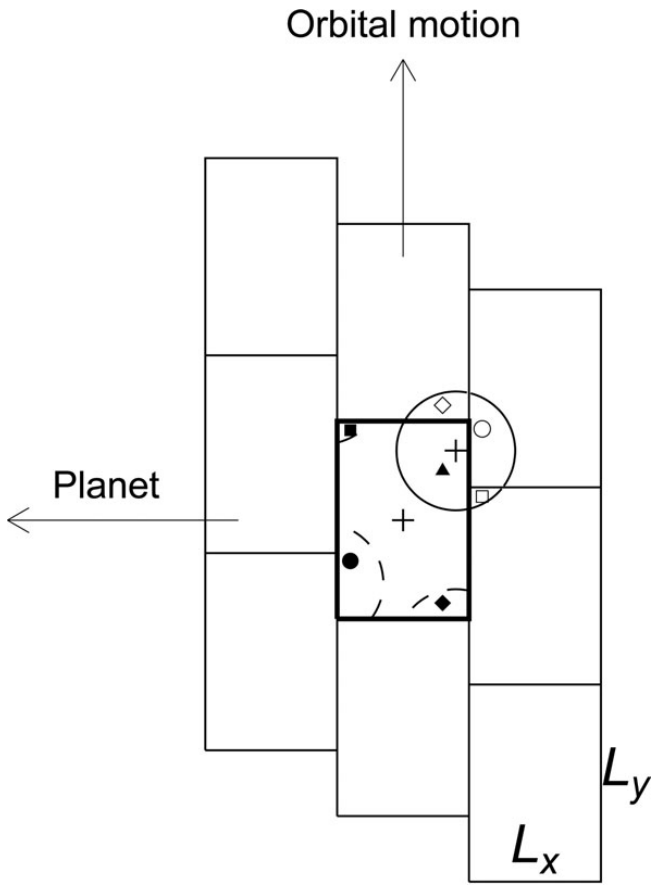

Figure 3: *In the shearing-sheet model, the main simulation box is surrounded by eight ghost boxes. Only the nearest particles within a small circle surrounding each point on the boundary exert a considerable influence on the main box. This figure is from Reference [3]*

The particles in the eight ghost boxes also contribute to the self-gravity calculations.

The shearing sheet model is usually coupled with a collisions model, which generally assume instantaneous impacts between hard spheres. Linear and angular momentum is conserved and the dissipative nature of the collisions is modelled by a coefficient of restitution. The adjacent ghost boxes must be checked for collisions at the boundary.

As might be guessed, the shearing sheet model is most easily implemented

in an N-body simulation. We will discuss these next, in particular, the REBOUND n-body code in detail.

2.3 Collisional N-body Codes (REBOUND)

Our understanding of astrophysical systems has been greatly advanced by N-body codes, computer software that follow the motion of a large number of masses under their mutual gravitational attraction. N-body codes usually come in two flavours: collisional and collision-less. Collisional N-body codes simulate the evolution of a system with N bodies by numerically integrating the equations of motion for exactly those N bodies. Collision-less N-body codes simulate the evolution of a system with N bodies by following the motion of $N^* < N$ particles, which represent Monte Carlo walkers as opposed to actual physical bodies. REBOUND is the only publicly available multi-purpose collisional N-body code, which means it tracks the evolution of every particle in astrophysical systems. It does this through four main modules.

The first module is the gravity module. The gravitational interaction between particles is traditionally computed using direct summation:

$$\boldsymbol{a}_i = \sum_{j \neq i} \frac{G m_j \left(\boldsymbol{r}_j - \boldsymbol{r}_i\right)}{|\boldsymbol{r}_j - \boldsymbol{r}_i|^3} \,. \tag{11}$$

However, this method of computing the gravitational force must iterate through each pair of particles, which is an $\mathcal{O}(N^2)$ process. A more popular $\mathcal{O}(N \log N)$ approximate method used to calculate the gravitational force was proposed by Barnes and Hut (hereafter BH) in 1986 [1]. The BH algorithm is based on the idea that distant particles contribute to the gravitational force less than those nearby. The BH algorithm starts by placing an imaginary cube around the simulation space and dividing this cube into eight sub-cubes. If any sub-cube contains more than one particle, we divide it into eight equal sub-cubes again. This hierarchy of cubes forms an octree and the original cube encapsulating the entire simulation space is called the root of the tree. Each cube after the root has one parent, seven siblings and potentially eight children. At the base of the tree is the particles, which are called leaves. Once this tree is constructed we can then approximate the total force on a particle

as follows. If a given cube of particles is too far away from our particle, we calculate it's total mass and center of mass and use these as an approximate method of calculating the force of this group of particles on our particle using Newton's gravitational law. Contributions from individual particles are only considered when they are nearby. In essence, if a group of particles is too far away from the current particle we are calculating the total force of this group on our particle using a truncated multipole expansion, otherwise we just use Equation (18). For reference in later sections, the condition that is used to determine if a cell is too far way is

$$|\Delta \boldsymbol{r}| > \frac{w}{\theta} \ , \tag{12}$$

where $\Delta \boldsymbol{r}$ is the relative position between the particle and the cell's center of mass, w is the width of the cell and θ is a free parameter called the opening angle. Varying the opening angle determines how accurate or how fast the gravity calculation with be using the BH tree. A smaller opening angle means a more accurate force calculation, but a slower computation time.

Once the total force has been calculated for each of the N bodies, we then need to integrate their positions and momenta forward in time. This is done in REBOUND using any one of three symplectic integrators: leapfrog, Wisdom-Holman, and the symplectic epicycle integrator (SEI). Each one of these integrators steps the positions and momenta forward in time by integrating Hamilton's equation. The Hamiltonian for an N-body system can generally be split into two parts: $H = H_1 + H_2$. Usually H_1 might be the kinetic term, $\frac{1}{2}\boldsymbol{p}^2$, where p is the generalized momentum vector and H_2 might be the potential term, $\phi(\boldsymbol{q})$, where $\boldsymbol{q}$ is the generalized position coordinate vector. The symplectic integrator works by integrating Hamilton's equations of motion with H_1 for half a time-step, $\mathrm{d}t/2$, known as a drift step, followed by integrating Hamilton's equations of motion for a full time-step with just H_2, known as a kick step, finally ending with one last drift step for $\mathrm{d}t/2$. The integrator is labelled symplectic because it maps the initial generalized coordinates $(\boldsymbol{q}_i, \boldsymbol{p}_i)$ to the final generalized coordinates $(\boldsymbol{q}_f, \boldsymbol{p}_f)$ through an operation that conserves what's known as a Poincare invariant. This conservation property tends to lessen the numerical dissipation often associated with integrators [4].

The third important module of REBOUND computes collisions. These mod-

ules use a hard-sphere model and resolve collisions through momentum and energy conservation. A constant or an arbitrary velocity dependent coefficient of restitution ϵ can be specified to model inelastic collisions in REBOUND. The relative velocity after the collision is then given by

$$\begin{aligned} v_n' &= -\epsilon v_n \\ v_t' &= v_t \,, \end{aligned} \tag{13}$$

where v_n and v_t are the relative normal and tangential velocities before the collision. Like the gravity module, we can compute the momentum contributions from the collisions for each particles by iterating through all other active particles and determining if they have collided. This direct approach scales as $\mathcal{O}(N^2)$. Once again, REBOUND uses a BH octree to speed up the collision detection. To determine if two particles collided, one starts at the root of the tree and descends into each daughter cell as long as the distance of the particle to the cell center is smaller than some critical threshold value (discussed in Section (3.8)). If a potential collision is detected, this particle pair is added to a collision queue which will later add the momentum contributions from the collision.

The last important module in REBOUND implements the boundary conditions. REBOUND currently supports open (no boundaries), periodic boundaries which make use of ghost box cells, and shearing-periodic boundaries [5].

One thing REBOUND doesn't currently support, is GPU acceleration. We will discuss this next.

2.4 The GPU and OpenCL

For high resolution N-body simulations of planetary rings, we will need all of the processing power we can get — one CPU is usually not sufficient. In every laptop and desktop computer there is a graphics card which uses hundreds of its processing cores to compute millions of pixel colours each second in parallel. Before 2008, general purpose scientific computation on the GPU was essentially non-existent. Then, around 2008, the Kronos group, formed by Apple and a few other companies, released the first

version of OpenCL (https://www.khronos.org/opencl/). The advantage of the OpenCL API over its competitor, the CUDA API (http://www.nvidia.ca/object/cuda_home_new.html), which was released about the same time, is that OpenCL can be used to program for any GPU hardware (or CPU hardware), not just NVIDIA cards, which the NVIDIA CUDA API is limited to. Furthermore, OpenCL can also be used to talk to systems with both CPUs and GPUs available for computation, that is, heterogeneous systems. Finally, OpenCL is open source and supported by an active development team.

Classical N-body codes have already been successfully sped up on GPU hardware, see for example the CUDA implementation described in Reference [6]. But these implementations use direct summation and do not include the BH tree algorithm to calculate force or to detect collisions (collisions are ignored). Parallelizing the Barnes Hut tree code presents one of the biggest problems in this project, since the Barnes Hut algorithm repeatedly builds and traverses an irregular tree-based data structure multiples times per simulation. Despite the particular non-parallel nature of the BH tree, it has been shown that the Barnes Hut algorithm can be sped up on GPU hardware by 74 times over a serial implementation (see Reference [7]) and another set of authors found that their N-body code with a BH tree for gravity calculations ran 20 times faster using the GPU and CUDA API then an equivalent serial implementation [8].

In order to understand the OCCLUDE OpenCL code which is to follow, we must delve a bit into the OpenCL model and how to write code, called kernels, which run on the GPU. This will not be a deep overview of the API or GPU programming, for that see [9]

The OpenCL API models all hardware used for computation (this could be a GPU, CPU, or something else, so the hardware is usually just called the "**device**") in the same way. This OpenCL hardware model is displayed in Figure (4). The hardware model has one big storage facility called **global memory** which is located off-chip. All variables and arrays stored in global memory are tagged with a "**__global**" prefix before their type identifier. This global memory storage may have a subsection used for read-only memory storage which can be faster to access. All variables and arrays stored in this read-only or constant memory are tagged with a "**__constant**" prefix before their type identifier. On the actual chip, there are devices the OpenCL

model calls **compute units**. These are independent processing devices that have their own set of local memory storage facilities, fittingly called **local memory**. Variables in local memory are tagged with the "**__local**" prefix and are shared amongst all cores in a compute unit. The compute units also contain processing cores which do the actual arithmetic. The coupled package of instructions and the local memory associated with those instructions to be executed on the cores in a compute unit are called **work-items** or threads, we will use both terms throughout. Each thread also has some **private memory** only accessible by it, which is located in the local memory storage on the compute unit. Variables stored in private memory have no prefix before their type identifier and are declared like they are in a C function (e.g. int a = 5 stores a value of 5 for a in private memory). The work-items are packaged in groups called **work-groups** which are of user-defined size, but are typically multiples of the number of processing cores in the compute unit. Finally, on a lot of hardware, such as NVIDIA and AMD GPUs, the cores in the compute unit can only compute a certain number of work-items in lockstep at a given time. This number of work-items is called a **wavefront** and will play a large role in the OCCLUDE code.

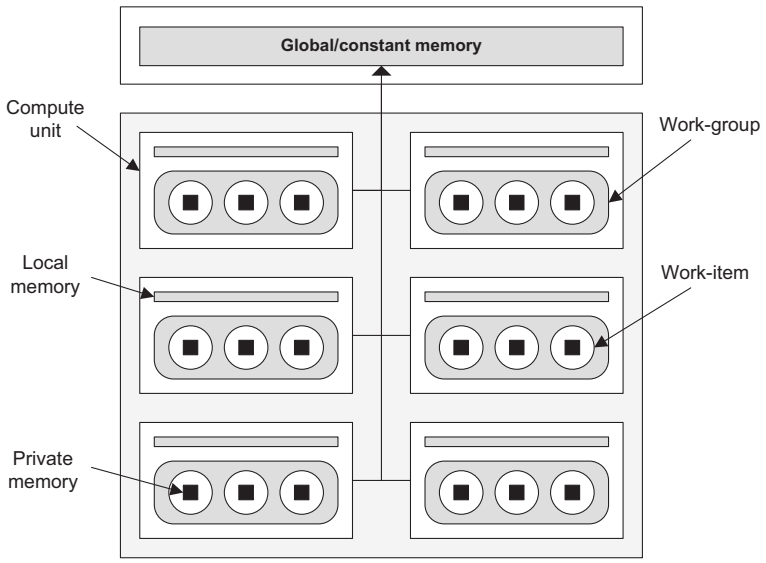

Figure 4: *The OpenCL model for the device hardware. This could be an AMD GPU or an NVIDIA GPU, or any other computational hardware.*

The usual pipeline for an OpenCL program is as follows: first the data is transferred to the device (GPU), then groups of instructions written by the programmer as kernels are sent to each of the compute units, which then run these instructions. The data is then transferred back to the GPU. Finally, to give an example of how to write a kernel to run on the device, a simple kernel that adds the elements in two arrays stored in global memory and stores the result in an array in global memory would be written as

```
__kernel void add_arrays(__global *int a,__global *int b,__global *int c)
{
      //get work-item id
      int i = get_global_id(0);
      c[i] = a[i] + b[i];
}
```

Figure 5: *A simple kernel that adds two arrays a and b in global memory and stores the result in a third array c located in global memory.*

Notice that there are no for-loops, because each work-item, identified by it's work-item id, handles one of the elements in the array. Finally, before we move on, a common bottleneck that props up in GPU coding is called **thread divergence**. On GPUs each compute unit can only compute one wavefront at a time and the threads in this wavefront are executed in lockstep. This means that if there is any data dependent condition or branching of the thread instructions, every thread in the wavefront will be forced through all instruction paths, which can lead to inefficient computation. In Section (3), we will describe how we overcome such problems when traversing the Barnes-Hut tree on the GPU.

3 OCCLUDE

3.1 Main Overview

OCCLUDE can be downloaded from my Github at `https://github.com/trevor-vincent/occlude.git`. It consists of eleven different OpenCL kernels, which utilize an octree stored in an array to speed up gravity

calculation and collision detection. A comparison of the REBOUND pipeline and the OCCLUDE pipeline is shown in Figures (6) and (7). I attempted to keep the OCCLUDE pipeline very similar to REBOUND and naming conventions for the OCCLUDE kernels are also very similar to the naming conventions used by REBOUND functions. While in it's current state OCCLUDE can only run shearing-sheet model simulations, it was designed in the same manner as REBOUND to be easily modifiable for other types of simulations.

The BH algorithm underlying OCCLUDE is based primarily off of a CUDA API algorithm proposed by Burtscher et al. [7]. The main idea of the algorithm, and we will see this in both the gravity calculation and collision search, is that instead of assigning one GPU thread per particle and descending the tree to accelerate the calculations for each thread, we assign one particle to a thread and the entire wavefront descends the tree together and votes on whether a certain tree cell is too far or not, I call this wavefront voting. This voting approach eliminates a type of GPU inefficiency known as thread divergence which we discussed in Section (2.4).

In the following sections, I will delve into the entire pipeline kernel by kernel. For a complete listing of the OpenCL kernel code see the Appendix. Finally, In Section (3.10) I will describe tests that have been completed so far to assess OCCLUDEs accuracy and speed.

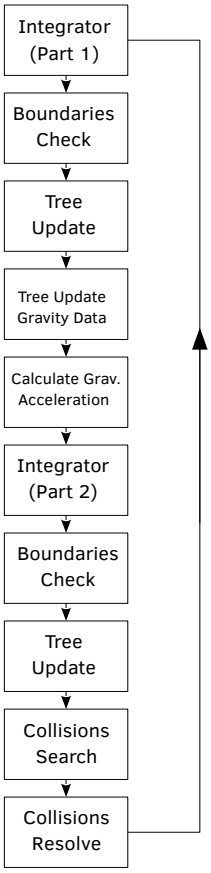

Figure 6: *A simplified sketch of the REBOUND pipeline.*

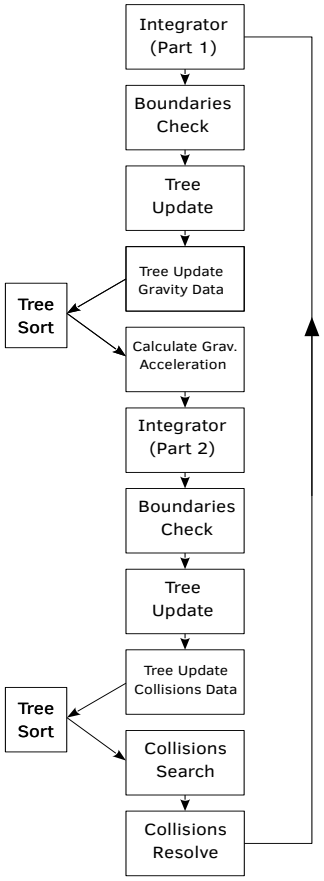

Figure 7: *A simplified sketch of the OCCLUDE pipeline. Each Box corresponds to a OpenCL kernel.*

3.2 Integrator Kernel

To evolve the motion of the bodies in the shearing sheet simulation forward in time we use the symplectic epicycle integrator (SEI) introduced by Rein and Tremaine [10]. This is a mixed variable symplectic (MVS) integrator that is second order and time reversible. MVS integrators split the Hamiltonian into two integrable parts $H(\boldsymbol{r}, \boldsymbol{p}) = H_A(\boldsymbol{r}, \boldsymbol{p}) + H_B(\boldsymbol{r}, \boldsymbol{p})$, with $|H_B| << |H_A|$. First the trajectory is advanced under the influence of H_A for half a time-step, then $|H_B|$ for a full timestep, then H_A again for half a time-step. If we assume that the gravitational forces from the other ring particles are much

smaller than the Keplerian force, then, from the Hamiltonian in Equation (8), we have

$$\begin{aligned} H_A &= H_0(\boldsymbol{r}, \boldsymbol{p}) \ , \\ H_B &= \Phi \ . \end{aligned} \tag{14}$$

The convenient attribute of this split is the fact that we already have the analytic solution to the equations of motion governed by the H_A Hamiltonian — the epicycle approximation solutions (See Equation Set (9)), but this time they are exact analytical solutions (since H_A does not include the potential Φ). If the initial position the ring particle in phase space is $(x_0, y_0, z_0, v_{x0}, v_{y0}, v_{z0})$, we can rewrite the epicycle solutions given by Equation Set (9) as

$$\begin{aligned} x(t) &= x_g + (x_0 - x_g)\cos(\Omega_0 t) + \frac{1}{2}(y_0 - y_g)\sin(\Omega_0 t) \\ y(t) &= y_g - \frac{3}{2}\Omega_0 x_g t + (y - y_g)\cos(\Omega_0 t) - 2(x_0 - x_g)\sin(\Omega_0 t) \\ z(t) &= z_0 \cos(\Omega_0 t) + v_0 \Omega_0^{-1} \sin(\Omega_0 t) \ , \end{aligned} \tag{15}$$

where $(x_g, y_g) = (2v_{y0}\Omega_0^{-1} + 4x_0,\, y_0 - 2v_{x0}\Omega_0^{-1})$ is the centre of epicyclic motion. The symplectic integrator integrates these equations forward in time for half a timestep, then it integrates the other Hamiltonian $H_b = \Phi$ for a full timestep, which amounts to the update

$$\boldsymbol{v}^{n+1} = \boldsymbol{v}^n - \delta t \nabla \Phi(\boldsymbol{r}^{n+1/2}) \ . \tag{16}$$

The SEI integator can be written as a set of three consecutive integration operations:

$$\hat{H}_{SEI}(\delta t) = \hat{H}_0\left(\frac{1}{2}\delta t\right)\hat{\Phi}(\delta t)\hat{H}_0\left(\frac{1}{2}\delta t\right) \ . \tag{17}$$

The SEI is split into two kernels, one that handles the drift step (part 1) and the other handles the kick-drift step (part 2). We assign one thread to

each particle in either kernel and each thread applies the operator given by Equation (17).

3.3 Boundary Check Kernel

The boundary check kernel applies the periodic shearing-sheet boundary conditions to each particle. No GPU optimizations are required for this kernel because the boundary check for a particle can be handled independently by one thread and thus the kernel is pretty much identical to it's CPU counterpart in REBOUND.

3.4 Tree Kernel

In REBOUND, the Barnes-Hut tree is traversed using C pointers. Each node in the tree is represented by a cell struct which points to it's eight children. See Figure (8) for the definition of this structure.

```
struct cell {
    double x; /**< The x position of the center of a cell */
    double y; /**< The y position of the center of a cell */
    double z; /**< The z position of the center of a cell */
    double w; /**< The width of a cell */

    double m; /**< The total mass of a cell */
    double mx; /**< The x position of the center of mass of a cell */
    double my; /**< The y position of the center of mass of a cell */
    double mz; /**< The z position of the center of mass of a cell */

    struct cell *oct[8]; /**< The pointer array to the octants of a cell */
    int pt;        /**< It has double usages: in a leaf node, it stores the ↩
        index
                    * of a particle; in a non-leaf node, it equals to (-1)*↩
                        Total
                    * Number of particles within that cell. */
};
```

Figure 8: *Definition of the struct cell. See rebound/src/tree.h in the REBOUND Github repository.*

To add a particle to the tree, REBOUND starts at the root cell and determines which of the root children cells the particle is in. If the cell is empty (the pointer to this child is NULL), then a new cell is constructed with eight children and the NULL pointer from the root cell is now set to point to this new child node. This new child then carries an integer field called "pt" to hold the particle index of the particle that is contained within it. This process is then repeated to add each of the subsequent particles. If one of the subsequent particles is to be added to a cell which contains another particle, REBOUND finds the child subcell each should be in and calls the function again (recursion) with these octants as the parents. The tree creation is done in the function **tree_add_particle_to_cell()** defined in rebound/src/tree.c (See the REBOUND Github repository).

There are two issues with a straight conversion of this tree algorithm to OpenCL code which is to be executed on the graphics processing unit (GPU). Firstly, OpenCL does not support recursion. Kernels cannot call themselves. Secondly, the cell structure and the pointer traversing operations are complex memory operations, which generally lead to slowdowns on the GPU [7].

Instead, a possible method for storing the tree on the GPU is to store the tree node ids in an array. In order to store the tree in an array, there must be a pre-determined maximum number of nodes, N_{nodes}. The total size of the array that will hold the tree will then be $N_{nodes} * 8$ because each of the nodes has eight children. We will store the eight children of the root node at the back of the array and each new node that is added to the tree will be placed in the eight elements before the last. There are three values that can be stored at any of the eight children of the array. These values are

1. val $<$ number of bodies $\rightarrow$ the child is a leaf.
2. val $\geq$ number of bodies $\rightarrow$ the child is a non-leaf.
3. val = -1 $\rightarrow$ the child is empty, or NULL.

At the start of our tree creation kernel, each OpenCL thread will be given a particle to store in this tree array. The particle id is "i" (see Figure (9)). The thread will start at the back of the array, where the eight children of the root cell lie (or descend down the tree until it reaches a null child or leaf). It will then lock this element by writing a value of "-2" to the array index. This is so that another thread cannot write to this branch of the tree while our thread is accessing it, which could lead to race conditions and inaccurate results.

The beginning of the kernel is shown in Figure (9). In this code snippet, the root child is first determined and snatched from the tree array, called **children_dev[]**. If the child id at this array location is not leaf and is not null, we will descend down the tree as shown in the nested while loop.

```
__kernel void cl_tree_add_particles_to_tree(
          __global float* x_dev,
          __global float* y_dev,
          __global float* z_dev,
          __global float* mass_dev,
          __global int* start_dev,
          __global int* children_dev,
          __global int* maxdepth_dev,
          __global int* bottom_node_dev,
          __constant float* boxsize_dev,
          __constant float* rootx_dev,
          __constant float* rooty_dev,
          __constant float* rootz_dev,
          __constant int* num_nodes_dev,
          __constant int* num_bodies_dev
          )
{

  int i, j, k, parent_is_null, inc;
  int child, node, cell, locked, patch, maxdepth_thread, depth;
  float body_x,body_y,body_z,r, cell_x, cell_y, cell_z;
  __local float root_cell_radius, root_cell_x, root_cell_y, root_cell_z;

  i = get_local_id(0);

  //the first thread in the work group initializes the data
  if (i == 0){
    root_cell_radius = *boxsize_dev/2.0f;
    root_cell_x = x_dev[*num_nodes_dev] = *rootx_dev;
    root_cell_y = y_dev[*num_nodes_dev] = *rooty_dev;
    root_cell_z = z_dev[*num_nodes_dev] = *rootz_dev;
    mass_dev[*num_nodes_dev] = 1.0f;
    start_dev[*num_nodes_dev] = 0;
    *bottom_node_dev = *num_nodes_dev;
    *maxdepth_dev = 1;
    //set root children to NULL
    for (k = 0; k < 8; k++)
      children_dev[*num_nodes_dev*8 + k] = 1;
  }
  barrier(CLK_LOCAL_MEM_FENCE);

  maxdepth_thread = 1;
  parent_is_null = 1;
  inc = get_global_size(0);
  i += get_group_id(0)*get_local_size(0);

  //..... see code in figure below
```

Figure 9: *Part 1 of the tree kernel, data is initialized by the first thread in each work_group. The index of the particle handled be each thread is i. See occlude/src/cl_tree.cl on the Github repository or see the Appendix for the full kernel.*

```
  // ... see code in figure above

while (i < *num_bodies_dev){
 //insert new body at root
  if(parent_is_null == 1){
    parent_is_null = 0;
    body_x = x_dev[i];
    body_y = y_dev[i];
    body_z = z_dev[i];
    //root node
    node = *num_nodes_dev;
    depth = 1;
    r = root_cell_radius;
    j = 0;
    if (root_cell_x < body_x) j = 1;
    if (root_cell_y < body_y) j += 2;
    if (root_cell_z < body_z) j += 4;
  }
  child = children_dev[node*8 + j];
  //if child is not a leaf or is not null, we descend the ↩
      tree
  while (child >= *num_bodies_dev){
    node = child;
    depth++;
    r *= 0.5f;
    j = 0;
    if (x_dev[node] < body_x) j = 1;
    if (y_dev[node] < body_y) j += 2;
    if (z_dev[node] < body_z) j += 4;
    child = children_dev[node*8 + j];
  }

 //..... see code in figure below
```

Figure 10: *Part 2 of the tree kernel, each thread descends the tree until it hits a leaf or a NULL node. See occlude/src/cl_tree.cl on the Github repository or see the Appendix for the full kernel.*

Once a child cell in **children_dev[]**, which is either a leaf ($<$ number of bodies) or NULL ($= -1$), has been retrieved, we then lock it by storing a value of -2 (unless of course it is already locked, in which case we wait until it's free and then lock) using an atomic operation (an atomic operation is an operation executed by a single thread on a memory location which is locked by the hardware for that thread during the operation). If the locked cell is NULL, we insert the particle here, if it's a non-empty leaf, we create a sub-tree by adding a new node at the front of the array, which is labelled by the integer **bottom_node_dev**.

```
// ... see code in figure above
//if child is not locked
  if (child != -2){
    locked = node*8 + j;
    if (child == atom_cmpxchg(&children_dev[locked], child, -2)){ //try to ↩
        lock chil
    //if NULL, insert body
    if(child == -1){
      children_dev[locked] = i;
    }
    //create new subtree by moving *bottom_node_dev down one node
    else {
      patch = -1;
      do {
      depth++;
        cell = atomic_sub(bottom_node_dev,1) - 1;
        patch = max(patch,cell) ;
        cell_x = (j & 1) * r;
        cell_y = ((j >> 1) & 1) * r;
        cell_z = ((j >> 2) & 1) * r;
        r *= 0.5f;
        mass_dev[cell] = -1.0f;
      start_dev[cell] = -1;
        cell_x = x_dev[cell] = x_dev[node] -r + cell_x;
        cell_y = y_dev[cell] = y_dev[node] -r + cell_y;
        cell_z = z_dev[cell] = z_dev[node] -r + cell_z;
        for (k = 0; k < 8; k++)
    children_dev[cell*8 + k] = -1;
        if (patch != cell)
    children_dev[node*8 + j] = cell;
        j = 0;
        if (cell_x < x_dev[child]) j = 1;
        if (cell_y < y_dev[child]) j += 2;
        if (cell_z < z_dev[child]) j += 4;
        children_dev[cell*8+j] = child;
        node = cell;
        j = 0;
        if (cell_x < body_x) j = 1;
        if (cell_y < body_y) j += 2;
        if (cell_z < body_z) j += 4;
        child = children_dev[node*8 + j];
      } while ( child >= 0 );

      children_dev[node*8 + j] = i;
      // after mem_fence, all work items now see the added sub-tree
      mem_fence(CLK_GLOBAL_MEM_FENCE);
    children_dev[locked] = patch;
    }
  maxdepth_thread = max(depth, maxdepth_thread);
    i += inc;
    parent_is_null = 1;
  }
  }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
atomic_max(maxdepth_dev, maxdepth_thread);
}
```


Figure 11: *Part 3 of the tree kernel, the node we descended to is locked and if it is NULL we write the new node location to it and if it contains a child we create a sub-tree. See occlude/src/cl_tree.cl on the Github repository or see the Appendix for the full kernel.*

The kernel completes when every particle has been accounted for in the tree and the maximum tree depth each thread reach is compared and stored in **maxdepth_dev** using an atomic operation.

3.5 Update Tree Gravity Data Kernel

Once we have created the Barnes-Hut tree and it is stored in the tree array on the GPU, we must traverse this array and compute the total mass and the center of mass for each of the tree cells. We do this in the kernel **cl_tree_update_tree_gravity_data()**. As a secondary calculation, this kernel also does some mini stream compaction by pushing the null children to the back of each row of the eight and the non-empty children to the front. This will speed up future calculations of the acceleration, as we'll see in Section 3.7.

This kernel assigns each thread to a node in the tree array and the OpenCL thread will calculate the assigned node's total mass, center of mass and stream-compact it's eight children. We will assign nodes to each thread by beginning at the bottom of the tree (which corresponds to the front of the array, pointed to by the integer **bottom_node_dev**) and working upward towards the root cell. Once the masses of the children cells of some parent cell have been calculated, that parent cell is then ready for calculation and the thread handling it can then start calculating the parent's mass. Thus, some of the threads which are given cells further up the tree must wait until the other threads calculate the masses of it's children.

As shown in Figure (12), the kernel starts by initializing the memory for each thread and assigning each thread a node with id given by "k".

```
__kernel void cl_tree_update_tree_gravity_data(
                    __global float* x_dev,
                __global float* y_dev,
                __global float* z_dev,
                __global float* mass_dev,
                __global int* children_dev,
                __global int* count_dev,
                __global int* bottom_node_dev,
                __constant int* num_nodes_dev,
                __constant int* num_bodies_dev,
                __local int* children_local
                 )
{
  int i, num_children_processed, k, child, inc, num_children_notcalculated, count, ↩
       num_group_threads, local_id;
  float child_mass, cell_mass, cell_x, cell_y, cell_z;
  __local int bottom_node;

  local_id = get_local_id(0);

  if (local_id == 0){
    bottom_node = *bottom_node_dev;
  }
  barrier(CLK_LOCAL_MEM_FENCE);

  inc = get_global_size(0);
  num_group_threads = get_local_size(0);
  num_children_notcalculated = 0;

  // ... see code in figure below
```

Figure 12: *Part 1 of the tree update gravity kernel, this is the initial preramble where the variables are declared and initialized. See occlude/src/cl_tree.cl on the Github repository or see the Appendix.*

Each thread then starts processing the children of the node it was given. If the children have already been processed by other threads then they are dealt with and num_children_processed is incremented. If the children have not been dealt with by other nodes then num_children_notcalculated is incremented and the thread continues on in the kernel (See Figure (13)).

```
//... see code in figure above
while (k <= *num_nodes_dev){
  if (num_children_notcalculated == 0){
    cell_mass = 0.f;
    cell_x = 0.f;
    cell_y = 0.f;
    cell_z = 0.f;
    count = 0;
    num_children_processed = 0;

    for (i = 0; i < 8; i++){
  child = children_dev[k*8 + i];
  if (child >= 0){
      //stream compaction
    if (i != num_children_processed){
      //move child to the front
      children_dev[k*8+i] =  1;
      children_dev[k*8+num_children_processed] = child;
    }
    children_local[num_children_notcalculated*num_group_threads + local_id] = child;
    child_mass = mass_dev[child];
    num_children_notcalculated++;

    if(child_mass >= 0.f){
      //cell is ready to be calculated
      num_children_notcalculated  ;
      if (child >= *num_bodies_dev){
        count += count_dev[child]   1; //subtract one
      }
      cell_mass += child_mass;
      cell_x += x_dev[child]*child_mass;
      cell_y += y_dev[child]*child_mass;
      cell_z += z_dev[child]*child_mass;
    }
    num_children_processed++;
  }
    }
    count += num_children_processed;
  }
  //... see code in figure below
```

Figure 13: *Part 2 of the tree update gravity kernel, each thread is given a node and the children are pushed to the front and processed if they are ready. If they are not ready then the integer variable num_children_notcalculated is incremented. See occlude/src/cl_tree.cl on the Github repository or see the Appendix for the full kernel.*

If there are any children not processed (num_children_notcalculated != 0), and they are still not ready by the time we get to this part in the kernel, then we wait until they are ready. Once all of the children have been processed we add up the needed quantities — total mass, center of mass and number of particles in that node.

```
    //... see code in figure above
    //if there are some children not ready
    if (num_children_notcalculated != 0){
      do {
    child = children_local[(num_children_notcalculated   1)*num_group_threads + ↩
        get_local_id(0)];
    child_mass = mass_dev[child];
  //child is ready
  if (child_mass >= 0.f){
      num_children_notcalculated  ;
      if (child >= *num_bodies_dev){
        count += count_dev[child]   1; //we subtract one b/c num_children_processed ↩
            counts the cell
      }
      cell_mass += child_mass;
      cell_x += x_dev[child] * child_mass;
      cell_y += y_dev[child] * child_mass;
      cell_z += z_dev[child] * child_mass;
    }
      } while ( (child_mass >= 0.f) && (num_children_notcalculated != 0) );
    }

    //all children and subchildren of this node have been accounted for
    //it's time to add everything up for node k
    if (num_children_notcalculated == 0){
      count_dev[k] = count;
      //child_mass is used as a temporary storage device here
      //for the inverse mass
      child_mass = 1.0f/cell_mass;
      x_dev[k]=cell_x*child_mass;
      y_dev[k]=cell_y*child_mass;
      z_dev[k]=cell_z*child_mass;
      //before freeing up this cell by setting a nonzero mass_dev, we must first make ↩
          sure
      //the cell information is loaded into memory first, by using a MEM_FENCE
      mem_fence(CLK_GLOBAL_MEM_FENCE);
      mass_dev[k] = cell_mass;
      k += inc;
    }
  }
}
```

Figure 14: *Part 3 of the tree update gravity kernel, for each of the children nodes that weren't ready to be processed, we wiat until they are. Once they are ready and dealt with we then add up everything for the node that thread is handling. See occlude/src/cl_tree.cl on the Github repository or see the Appendix for the full kernel.*

3.6 Tree Sort Kernel

In the last kernel, we calculated the masses and center of masses of each node. But along-side these calculations, we also added up the number of particles in each node and stored it in the array **count_dev[]** which has the usual size given by **num_nodes_dev**. This array is used, along with the tree we created, to help sort the particles so that the leaf ids for each particle are aligned in the order they fall in the tree. So, for example, all the particles in the first child node of the root cell should be all together in the same

location of the array and the particles in the second child node should come right after, and similarly for the children's children. This pattern should continue all the way down to furthest depth of the tree. This sorted leaf id array will be of great use, as we will see, in the next kernel for calculating the accelerations due to gravitational interactions.

For this sort kernel, we continue the usual method of assigning one thread to each node of the tree. Unlike the last kernel, where the threads with nodes at the bottom of the tree get to execute first, the threads at the top of the tree (near the root cell) get to act first this time.

The kernel begins in the usual manner by having the first thread of a work-group load in the node id of the bottom node in the tree to local memory. Then each thread is prescribed a node with id "k".

```
__kernel void cl_tree_sort_particles(
                                        __global int* children_dev,
                                        __global int* count_dev,
                                        __global int* start_dev,
                                        __global int* sort_dev,
                                        __global int* bottom_node_dev,
                                        __constant int* num_nodes_dev,
                                        __constant int* num_bodies_dev
                                    ){

  int i,k, child, dec, start, bottom;
  __local int bottoms;

  if (get_local_id(0) == 0){
    bottoms = *bottom_node_dev;
  }
  barrier(CLK_LOCAL_MEM_FENCE);
  bottom = bottoms;

  dec = get_global_size(0);
  k = *num_nodes_dev + 1 - dec + get_local_id(0) + get_group_id(0)*get_local_size↩
      (0);

// ... see code in figure below
```

Figure 15: *Part 1 of the tree sort kernel, this is the initial preramble where the variables are declared and initialized. See occlude/src/cl_tree.cl on the Github repository or see the Appendix.*

Once this is done, the node handling the root cell will begin processing the children. For each of the children it will set the starting place for where these

children cells will write their leaf ids in the sorted array. These starting locations are written in **start_dev[]**. Once these starting locations have been written, the threads handling the children nodes will then parse their children and write their starting locations to **start_dev[]**. Only when a thread reaches a leaf, will it's id be written to the sorted index array **start_dev[]** using the starting offset given by **start_dev[]**.

```
  // ... see code in figure above

while (k >= bottom){
  start = start_dev[k];
  //when the node is ready
  if (start >= 0){
    for (i = 0; i < 8; i++){
      child = children_dev[k*8 + i];
      //non-leaf - set starting place to write in sorted ↩
          array
      if (child >= *num_bodies_dev){
        start_dev[child] = start;
        start += count_dev[child];
      }
      //leaf - write to sorted array
      else if (child >= 0) {
        sort_dev[start] = child;
        start++;
      }
    }
    k -= dec;
  }
}
```

Figure 16: *Part 2 of the tree sort kernel, here each thread is assigned a node from the top downwards. If that node is not a leaf, then room is made in sort_dev for all the leafs in that non-leaf node and if that node is a leaf, then we write the leaf id to sort_dev. See occlude/src/cl_tree.cl on the Github repository or see the Appendix.*

After this kernel is completed, we now have an array of leaf ids — **sort_dev[]**, that holds the ids of leafs sorted by their spatial location. So particles that are spatially close will have ids that are close in **sort_dev[]**.

3.7 Gravity Kernel

Once the tree has been created and the centers of mass and total masses of each tree node have been stored, we can then use the tree to speed up the

calculation of the acceleration due to (self-)gravity. In the last kernel, we computed an array of leaf ids sorted by spatial location and this sort will now prove to be quite useful.

We will assign each thread to compute the gravity force on a body, or equivalently the acceleration of that body due to each of the other bodies in the simulation. As we have discussed in Section (2.4), each one of these threads is part of a small execution group called a wavefront. To reiterate, a wavefront is a group of threads that execute in lockstep on the device. That means, that they must all execute the same instructions on different (or perhaps the same) data items at the same time. If there is a data dependent branch in the instruction path that perhaps only one thread in the wavefront passes through, then all of the other threads in the wavefront must stall while that single thread passes through this instruction path. Thus, most of the threads in the wavefront, and correspondingly, the cores in a multiprocessor on the GPU, are idle. This problem, which leads to slower performance due to less active cores, is so common in GPU codes that it has a name — thread divergence.

If every thread traverses the tree for a given body, then this traversal will be dependent upon the spatial location of the body the thread is handling and thus there will be different instruction paths for each of the threads in a wavefront. Furthermore, even if two threads in the same wavefront happen to be calculating the tree condition (Equation (12)) on the same tree cell, they might disagree on whether that cell is too far, which would lead to divergent instruction paths (one thread would descend down the tree further, while the other would move back up the tree). Thus, we have two potential areas of divergence. We can eliminate both areas almost completely by doing the following. Each wave front will use the sorted leaf ids to access the particle positions and other particle data. This means that each wavefront will be working with spatially close particles. Secondly, the thread with the lowest id in the wavefront will do the tree traversal for the entire wavefront, but the entire wavefront will vote on the tree condition (Equation (12)) to determine whether or not the entire wavefront descends the tree, that is, where the first thread goes next. Thus, if only one thread in the wavefront thinks the tree cell is close enough to descend, then the entire wavefront will descend that tree node. Since each wavefront works on spatially close bodies, there is a high probability that the votes will be either unanimously negative or unanimously positive in favour of descending that particle tree node.

```
__kernel void cl_gravity_calculate_acceleration_for_particle(
                  __global float* x_dev,
                  __global float* y_dev,
                  __global float* z_dev,
                  __global float* ax_dev,
                  __global float* ay_dev,
                  __global float* az_dev,
                  __global float* mass_dev,
                  __global int* sort_dev,
                  __global int* children_dev,
                  __global int* maxdepth_dev,
                  __global float* t_dev,
                  __constant float* OMEGA_dev,
                  __constant float* boxsize_dev,
                  __constant int* num_nodes_dev,
                  __constant int* num_bodies_dev,
                  __constant float* inv_opening_angle2_dev,
                  __constant float* softening2_dev,
                  __constant float* G_dev,
                  __local int* children_local,
                  __local int* pos_local,
                  __local int* node_local,
                  __local float* dr2_cutoff_local,
                  __local float* nodex_local,
                  __local float* nodey_local,
                  __local float* nodez_local,
                  __local float* nodem_local,
                  __local int* wavefront_vote_local
                  ){


  int i, j, l, node, depth, base, sbase, diff, local_id;
  float body_x, body_y, body_z, body_ax, body_ay, body_az, dx, dy, dz, ↩
      temp_register;
  float shiftx, shifty, shiftz;
  __local int maxdepth_local;
  __local float t_local;

#ifdef KAHAN_SUMMATION
  // accumulators for Kahan summation of phase
  float kahanC_ax = 0.f;
  float kahanC_ay = 0.f;
  float kahanC_az = 0.f;
  float kahanT = 0.f;
  float kahanY = 0.f;
#endif

   //... see code in figure below
```

Figure 17: *Part 1 of the gravity tree kernel, this is the initial preramble where the variables are declared and initialized. See occlude/src/cl_gravity_tree.cl on the Github repository or see the Appendix.*

The kernel starts off (See Figure (17)) by declaring the needed thread variables for the computation. There are a few new variables introduced here. First, the kernel arguments with the __local prefix are stored in local memory and are used to store the tree node information for the wavefront. For example, if there are 10 wavefronts stored on the a multi-processor for execution, then the first wavefront will have node information stored in **nodex_local[0]**, **nodey_local[0]** , ... , etc. For the second wavefront, the node information will be stored in **nodex_local[1]**, **nodey_local[1]** , ... , etc. The node_local array hold's the id for the node the wavefront is currently examining, pos_local holds the position (0 to 7) of the current child cell of node_local the wavefront is examining, and children_local holds the node id of that child the wavefront is currently examining. The wavefront_vote_local array holds the binary votes for each thread in the wavefront.

Second, since current GPU architectures run much faster with single precision than double precision, but single precision values are more susceptible to round-off error accumulation when summing, we use the Kahan summation algorithm to sum the acceleration contributions from each of the bodies and eliminate accumulation of roundoff error. We have a cumulator variable kahanC_ax, kahanC_ay and kahanC_az, for each dimension of the acceleration we are summing.

Next we use the first thread in each workgroup to load some important variables into local memory and block the workgroup from moving forward until this is done through a **barrier()** call. For each wavefront, we a variable used for the tree condition (See Equation (12)) in **dr2_cutoff_local[]** for each depth of the tree. The simulation time **t_dev** and the maximum depth of the tree **maxdepth_dev**, are also loaded into local memory.

```
  //... see code in figure above

local_id = get_local_id(0);

if (local_id == 0){
  maxdepth_local = *maxdepth_dev;
  temp_register = *boxsize_dev;
  t_local = *t_dev;
  dr2_cutoff_local[0] = temp_register * temp_register * (*inv_opening_angle2_dev↩
      );
  for (i  = 1; i < maxdepth_local; i++)
    dr2_cutoff_local[i] = dr2_cutoff_local[i-1] * .25f;
}

barrier(CLK_LOCAL_MEM_FENCE);
float t = t_local;

  //... see code in figure below
```

Figure 18: *Part 2 of the gravity tree kernel, we use the first thread in each workgroup to load some important variables into local memory. See occlude/src/cl_gravity_tree.cl on the Github repository or see the Appendix.*

We then assign each thread a body with leaf id "i" from the sorted array **sort_dev** and store the positions and accelerations of this body in thread registers,

```
   //... see code in figure above

if (maxdepth_local <= MAX_DEPTH){

  //wave front id;
   base = local_id / WAVEFRONT_SIZE;
  //thread id of first thread in wavefront
   sbase = base * WAVEFRONT_SIZE;
   //useful id for accessing dr_cutoff
   j = base * MAX_DEPTH;

   diff = local_id - sbase;
   if (diff < MAX_DEPTH){
     dr2_cutoff_local[diff + j] = dr2_cutoff_local[diff];
   }
   barrier(CLK_LOCAL_MEM_FENCE);

   for (int k = local_id + get_group_id(0)*get_local_size(0); k < *↩
       num_bodies_dev; k += get_global_size(0)){

     i = sort_dev[k];
     body_x = x_dev[i];
     body_y = y_dev[i];
     body_z = z_dev[i];

     body_ax = 0.f;
     body_ay = 0.f;
     body_az = 0.f;

   //... see code in figure below
```

Figure 19: *Part 3 of the gravity tree kernel, we assign each thread a body with leaf id "i" from the sorted array. See occlude/src/cl_gravity_tree.cl on the Github repository or see the Appendix.*

For this body, and the wavefront this thread is in, we iterate through each ghostbox and traverse the tree using two while loops, the outer loop traverses each parent node and the inner loop traverses the 8 child nodes. The first thread in a wavefront (the one that satisfies **sbase** == **local_id**) sets the tree node and stores it's information in local memory.

```
  //... see code in figure above

for (int gbx = -1; gbx <= 1; gbx++){
      for (int gby = -1; gby <= 1; gby++){
        for (int gbz = -1; gbz <= 1; gbz++){
          cl_boundaries_get_ghostbox(gbx,gby,gbz,&shiftx,&shifty,&shiftz,&↩
              temp_register,&temp_register,&temp_register, *OMEGA_dev, *↩
              boxsize_dev, *boxsize_dev, *boxsize_dev, t);

    depth = j;
    //first thread in wavefront leads the pack by picking the node
    if (sbase == local_id){
      node_local[j] = *num_nodes_dev;
      pos_local[j] = 0;
    }
    mem_fence(CLK_LOCAL_MEM_FENCE);

    while (depth >= j){
      while(pos_local[depth] < 8){
  //first thread in wavefront leads
  if(sbase == local_id){
    node = children_dev[node_local[depth]*8 + pos_local[depth]];
    pos_local[depth]++;
    children_local[base] = node;
    if (node >= 0){
      nodex_local[base] = x_dev[node] + shiftx;
      nodey_local[base] = y_dev[node] + shifty;
      nodez_local[base] = z_dev[node] + shiftz;
      nodem_local[base] = mass_dev[node];
    }
  }
  //the wavefronts do not move forward until they see the new local memory (↩
      mem_fence)
  mem_fence(CLK_LOCAL_MEM_FENCE);
  node = children_local[base];

  //... see code in figure below
```

Figure 20: *Part 4 of the gravity tree kernel. See occlude/src/cl_gravity_tree.cl on the Github repository or see the Appendix.*

The wavefront then votes on whether the cell is far enough away to justify using the multiple expansion. They vote on the condition given in Equation (12).

```
  //... see code in figure above

  //if the node is not null
  if (node >= 0){
    dx = nodex_local[base] - body_x;
    dy = nodey_local[base] - body_y;
    dz = nodez_local[base] - body_z;
    temp_register = dx*dx + dy*dy + dz*dz;

    //the vote
    wavefront_vote_local[local_id] = (temp_register >= dr2_cutoff_local[depth↩
        ]) ? 1 : 0;

    //the first thread in the wavefront adds up the votes
    if (local_id == sbase)
      for(l = 1; l < WAVEFRONT_SIZE; l++)
        wavefront_vote_local[sbase] += wavefront_vote_local[sbase + l];
    mem_fence(CLK_LOCAL_MEM_FENCE);
  //... see code in figure below
```

Figure 21: *Part 5 of the gravity tree kernel. See occlude/src/cl_gravity_tree.cl on the Github repository or see the Appendix.*

If the wavefront votes unanimously that the cell is too far away, or the node points to a leaf, we then calculate the cell/leaf's contribution to the acceleration of the body.

```
    //... see code in figure above

      if ((node < *num_bodies_dev) || wavefront_vote_local[sbase] >= ↩
          WAVEFRONT_SIZE){

        //if the node isn't the body we are computing the acc for
        if (node != i){
          temp_register = rsqrt(temp_register + *softening2_dev);
          temp_register = *G_dev * nodem_local[base] * temp_register * ↩
              temp_register * temp_register;
#ifdef KAHAN_SUMMATION
          kahanY = dx * temp_register - kahanC_ax;
          kahanT = body_ax + kahanY;
          kahanC_ax = (kahanT - body_ax) - kahanY;
          body_ax = kahanT;

          kahanY = dy * temp_register - kahanC_ay;
          kahanT = body_ay + kahanY;
          kahanC_ay = (kahanT - body_ay) - kahanY;
          body_ay = kahanT;

          kahanY = dz * temp_register - kahanC_az;
          kahanT = body_az + kahanY;
          kahanC_az = (kahanT - body_az) - kahanY;
          body_az = kahanT;
#else
          body_ax += dx * temp_register;
          body_ay += dy * temp_register;
          body_az += dz * temp_register;
#endif
        }
      }

    //... see code in figure below
```

Figure 22: *Part 5 of the gravity tree kernel. See occlude/src/cl_gravity_tree.cl on the Github repository or see the Appendix.*

If one of the threads in the wavefront votes in the negative, then we descend (See Figure (23)).

```
    //... see code in figure above
                  else{
                    depth++;
                    if(sbase == local_id){
                      node_local[depth] = node;
                      pos_local[depth] = 0;
                    }
                    mem_fence(CLK_LOCAL_MEM_FENCE↩
                        );
                  }
                }

    //... see code in figure below
```

Figure 23: *Part 6 of the gravity tree kernel. See occlude/src/cl_gravity_tree.cl on the Github repository or see the Appendix.*

At the end of the kernel, we add the calculate acceleration contribution to the arrays stored in global memory.

```
    //... see code in figure above

              // if child is null then remaining children of this node is ↩
                  null
              // due to the mini-stream compaction in the ↩
                  tree_gravity_update
              // kernel so move back up the tree
              else{
                depth = max(j, depth - 1);
              }
            }
            depth--;
          }
        }
      }
    }
    ax_dev[i] = body_ax;
    ay_dev[i] = body_ay;
    az_dev[i] = body_az;
  }
}
```

Figure 24: *Part 7 of the gravity tree kernel. See occlude/src/cl_gravity_tree.cl on the Github repository or see the Appendix.*

After the kernel has executed, we have three updated acceleration arrays **ax_dev[]**, **ay_dev[]** and **az_dev[]**.

3.8 Collisions Search Kernel

We also use the tree to speed up the search for possible collisions between the spherical particles with radii r_1 and r_2. A collision occurs if two criteria are met. Let $\Delta\boldsymbol{x}$ be the relative distance between two possible colliders and Δv be the relative velocity. A collision is added to the stack if both the following conditions hold simultaneously:

$$\begin{aligned} &|\Delta\boldsymbol{x}| \leq r_1 + r_2, \\ &\Delta\boldsymbol{x} \cdot \Delta\boldsymbol{v} < 0. \end{aligned} \tag{18}$$

Each thread in a wavefront is given a particle to query collisions for. The first thread in the wavefront traverses the tree for the group. If the node is a leaf, then each thread in the wavefront computes the above two conditions and appends either the leaf id if there was a collision, or a value of -1 if there was no collision to the collision array called **collisions_dev**. Just as was done for the "gravity tree kernel" (see Section (3.7)), the entire wavefront votes on whether or not to descend a tree node. If a single thread finds that a particular cell of the tree could have a potential collision, then the first thread in the wavefront shifts a local variable, node_local, to point to that node and the collision querying is repeated for this cell. The condition the wavefront votes on is the same that REBOUND uses: If $\Delta\boldsymbol{x}$ represents the distance between a particle's position and the center of the cell with width w, r is the radius of the particle, r_{max} is the maximum radius of a particle, then a collision could occur if

$$|\Delta\boldsymbol{x}| \leq r + r_{max} + \frac{\sqrt{3}}{2}w \tag{19}$$

The beginning code of this kernel is roughly the same as in the gravity tree kernel, so we will emit commenting on it. The main difference comes in the wavefront voting section, shown in Figure (25).

```
    //if the node is not null
    if (node >= 0){
      dx = (body_x + shiftx) - nodex_local[base];
      dy = (body_y + shifty) - nodey_local[base];
      dz = (body_z + shiftz) - nodez_local[base];
      temp_register = dx*dx + dy*dy + dz*dz;
      // if it's a leaf cell
      if (node < *num_bodies_dev)
      {
        //if the node isn't the same body, check for a collision
        if (node != i){
          rp = body_rad + noderad_local[base];
          dvx = (body_vx + shiftvx) - nodevx_local[base];
          dvy = (body_vy + shiftvy) - nodevy_local[base];
          dvz = (body_vz + shiftvz) - nodevz_local[base];
          if ( temp_register <=  rp*rp && dvx*dx + dvy*dy + dvz*dz < 0){
            collisions_dev[i + gbx_offset*(gbx+1) + gby_offset*(gby+1) + ↩
                gbz_offset*(gbz+1)] = node;
          }
        }
      }
      else{
        rp = body_rad + dr_cutoff_local[depth];
        wavefront_vote_local[local_id] = (temp_register >= rp*rp) ? 1 : 0;
        if (local_id == sbase)
        for(l = 1; l < WAVEFRONT_SIZE; l++)
        wavefront_vote_local[sbase] += wavefront_vote_local[sbase + l];
        //this blocks the warp from moving forward since they must wait for ↩
            their leader (local_id = sbase)
        mem_fence(CLK_LOCAL_MEM_FENCE);
        //the warp votes whether or not to descend. If one member of the warp ↩
            votes descend, the warp descends
        if (wavefront_vote_local[sbase] < WAVEFRONT_SIZE){
          depth++;
          if(sbase == local_id){
            node_local[depth] = node;
            pos_local[depth] = 0;
          }
          mem_fence(CLK_LOCAL_MEM_FENCE);
        }
      }
    }
```

Figure 25: *The wavefront voting part of the collisions search kernel. See occlude/src/cl_collisions_tree.cl on the Github repository or see the Appendix for the full kernel.*

In Figure (25), we see an if-else branch set. The if branch relates to the case where the node is a leaf, and each thread in the wavefront evaluates the collision conditions given by (18). In the else branch, we handle the case where the node is neither a leaf nor null, and each thread in the wavefront

evaluates condition (19) and the final tally determines whether or not the wavefront descends into this node or discards it completely in the collision search.

Once this tree search has been completed for all bodies, we then have a large array **collisions_dev** which holds for each body, either a "-1" for no collision or a leaf id for the second body in the collision pair.

3.9 Collision Resolution Kernel

In the last section, the collision search kernel used the tree to create an array of collisions to be processed – **collisions_dev[]**, with a value of -1 for no collision and a value of the leaf id of the second particle in the collision if there was a collision.

The collision resolution uses a hard-sphere model along with Equation Set (13) to resolve each collision stored in **collisions_dev[]**. Each thread is assigned one entry in **collisions_dev[]** corresponding to the sorted leaf id in **sort_dev[]**. This helps ensure that each wavefront is handling particles that are spatially close and therefore more likely to have collided or not have collided with other groups of particles. Since for each collision between two particles stored in **collisions_dev[]**, there is the corresponding collision.

Since this kernel is a straightforward calculation, which ends in updates of the **vx_dev[]**, **vy_dev[]** and **vz_dev[]** arrays, we will not delve into the actual code, which is listed in the Appendix.

3.10 Tests

3.10.1 Tests for Accuracy

Each kernel was independently checked for accuracy, with the gravity calculations checked against a direct summation algorithm computed on the CPU and the collision search kernel checked against a direct nearest-neighbour search computed on the CPU. To test the output of the entire code, I ran

simulations with radius distributions and other important parameters tuned to ones consistent with data for Saturn's A and B rings.

Specifically, the radius distribution is based on current estimates of the sizes of the icy satellites surrounding Saturn, which are determined based off of probes backscattering light of certain wavelengths. The current understanding is that the size of ring particles could be anywhere from fine dust to embedded moonlets, which are kilometers across. The most natural fit for such a distribution is a power law. This means that the number of particles per unit volume in a small size interval dr, centered on radius r:

$$N(r)dr = C_0 r^{-p} dr;\ r_{min} < r < r_{max} \tag{20}$$

I used standard values of p = 3 with $r_{min} = 1m$ and $r_{max} = 4m$. The masses are calculated by setting the particle density parameter and assuming the ring particles are spheres with radii given by Equation (20).

Other independent parameters are summarized in Table (1).

σ_0 (surface density)	2000 kg m^{-2}
ρ (particle density)	200 kg m^{-3}
Ω_0	$1.3143527 \times 10^{-4} s^{-1}$
Δt	$10^{-3}\Omega_0^{-1} 2\pi$
G	$6.67428 \times 10^{-11} m^3$ kg^{-1} s^{-2}
w (box size)	100 m
θ (opening angle)	.5
ϵ (softening)	.1

Table 1: *Parameters used in the OCCLUDE shearing-sheet simulations.*

The coefficient of restitution used for the collision kernels is based on laboratory experiments with ice particles at temperatures and pressures generally associated with the planetary ring environment. These laboratory measurements made by Bridges et al. [11] showed that the normal restitution coefficient ϵ_n decreases monotonically with the normal component of the impact speed v_n:

$$\epsilon_n(v_n) = \left(\frac{v_n}{v_c}\right)^{-.234}, \tag{21}$$

where v_c = .0077 cm/s. Equation (21) is used widely in simulation codes such as REBOUND and it's used in OCCLUDE too.

The centers of the particles in the rotation plane are initially uniformly spread around the box and the position in the longitudinal direction are normally distributed about the rotation plane. Figure (26) shows the initial centers of the particles and Figure (27) shows the centers of the particles after 1 orbit. We can see the self-gravity wakes forming, which confirms that the simulation code is working correctly.

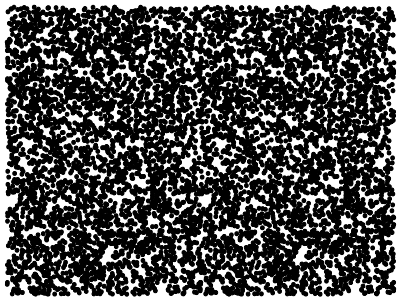

Figure 26: *The initial locations of the centers of the spherical ring particles.*

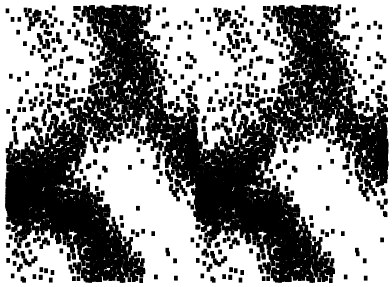

Figure 27: *The locations of the centers of the spherical ring particles after two orbits. We can see structure such as self-gravity wakes forming.*

3.10.2 Tests for Speed

Initial tests for a speedup with OCCLUDE on my Tesla generation NVIDIA Geforce 9800 GT card show that OCCLUDE is almost 2.5 times faster than REBOUND at higher particle numbers and the speedup monotonically increases with the number of particles (see Figure (28)). The parameters used in the simulation are given in the last section and the particle numbers listed in Figure (28) include the ghost particles, which are included in all calculations.

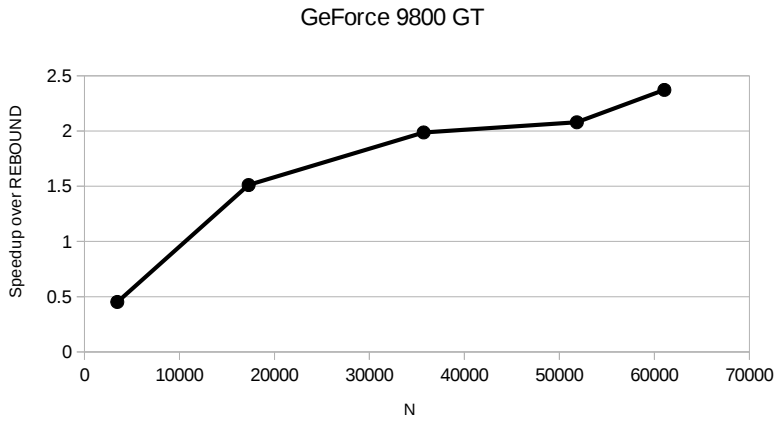

Figure 28: *The results of a speed test of OCCLUDE against REBOUND on a single CPU. THE OCCLUDE code was executed on a NVIDIA GeForce 9800 GT card.*

A much higher speedup would be expected on higher-end cards. The Geforce 9800 GT is a couple of years old and NVIDIA has since been through at least three generations of GPU architectures. However, further profiling and tweaking of the algorithm (especially the collision detection) should ensure at least a 10 times speedup. We don't expect the 74 times speedup that Burtscher et al. [7] received since he was comparing against a suboptimal CPU algorithm (the CPU version has the tree recreated every iteration instead of updated) and he ignored collisions. Also, Burtscher et al. used proprietary CUDA functions, which are not available in the OpenCL model (in particular, the __all() and __any() warp-voting functions). Furthermore, we don't expect the 20 times speedup that Bedorf et al. received since he didn't consider collisions and was comparing against a suboptimal CPU algorithm in the same manner.

4 Conclusion

In it's current state, OCCLUDE appears to accurately evolve the shearing-sheet N-body model with a speed up of over 2.5 times that of REBOUND on a single CPU. With more tweaking and profiling, we should be able to see at least a ten times speedup of OCCLUDE over REBOUND on high-end graphics cards. With such a speedup comes the ability to do ring simulations with higher densities, which is critical for investigating the dense A, B and C rings of Saturn. As far as the author is aware, this is the first publicly available OpenCL collisional N-body code.

5 References

[1] J. Barnes and P. Hut, "A hierarchical o (n log n) force-calculation algorithm," 1986.

[2] L. Esposito, Planetary Rings: A Post-Equinox View. Cambridge Planetary Science, Cambridge University Press, 2014.

[3] J. Schmidt, H. Salo, F. Spahn, and O. Petzschmann, "Viscous overstability in saturn's b-ring: Ii. hydrodynamic theory and comparison to simulations," Icarus, vol. 153, no. 2, pp. 316–331, 2001.

[4] J. Binney and S. Tremaine, Galactic Dynamics. Princeton series in astrophysics, Princeton University Press, 1987.

[5] H. Rein and S.-F. Liu, "Rebound: an open-source multi-purpose n-body code for collisional dynamics," arXiv preprint arXiv:1110.4876, 2011.

[6] L. Nyland, M. Harris, and J. Prins, "Fast n-body simulation with cuda,"

[7] M. Burtscher and K. Pingali, "An efficient cu da implementation of the tree-based barnes hut n-body algorithm," GPU computing Gems Emerald edition, p. 75, 2011.

[8] J. Bédorf, E. Gaburov, and S. Portegies Zwart, "A sparse octree gravitational< i> n</i>-body code that runs entirely on the gpu processor," Journal of Computational Physics, vol. 231, no. 7, pp. 2825–2839, 2012.

[9] M. Scarpino, OpenCL in Action: How to Accelerate Graphics and Computation. Manning Publications Company, 2011.

[10] H. Rein and S. Tremaine, "Symplectic integrators in the shearing sheet," Monthly Notices of the Royal Astronomical Society, vol. 415, no. 4, pp. 3168–3176, 2011.

[11] F. G. Bridges, A. Hatzes, and D. Lin, "Structure, stability and evolution of saturn's rings," 1984.

[12] W. Dehnen, "A very fast and momentum-conserving tree code," The Astrophysical Journal Letters, vol. 536, no. 1, p. L39, 2000.

[13] J. Wisdom and S. Tremaine, "Local simulations of planetary rings," The Astronomical Journal, vol. 95, pp. 925–940, 1988.

[14] L. Esposito and S. Krimigis, Saturn from Cassini-Huygens. Springer-Link: Springer e-Books, Springer, 2009.

Appendices

A OpenCL Kernels

A.1 Integrator Part 1 Kernel

See Appendix A.7 for the definition of the HO12 function.

```
__kernel void cl_integrator_part1(
            __global float* x_dev,
            __global float* y_dev,
            __global float* z_dev,
            __global float* vx_dev,
            __global float* vy_dev,
            __global float* vz_dev,
            __global float* ax_dev,
            __global float* ay_dev,
            __global float* az_dev,
            __global float* t_dev,
            __constant int* num_bodies_dev,
            __constant float* dt_dev,
            __constant float* OMEGA_dev,
            __constant float* OMEGAZ_dev,
            __constant float* sindt_dev,
            __constant float* tandt_dev,
            __constant float* sindtz_dev,
            __constant float* tandtz_dev
          )
{
  __local float t_local;
  int id,k, inc;

  id = get_local_id(0);
  if (id == 0){
    t_local = *t_dev;
  }
  barrier(CLK_LOCAL_MEM_FENCE);
  float t = t_local;

  id += get_group_id(0)*get_local_size(0);
  inc = get_global_size(0);

  for (k = id; k < *num_bodies_dev; k += inc){
    operator_H012(
      k,
      *dt_dev,
      *OMEGA_dev,
      *OMEGAZ_dev,
      *sindt_dev,
```

```
      *tandt_dev,
      *sindtz_dev,
      *tandtz_dev,
      x_dev,
      y_dev,
      z_dev,
      vx_dev,
      vy_dev,
      vz_dev,
      ax_dev,
      ay_dev,
      az_dev
      );
  }

  t += *dt_dev/2.f;
  if (id == 0)
    *t_dev = t;
}
```

A.2 Boundaries Check Kernel

```
__kernel void cl_boundaries_check(
             __global float* x_dev,
             __global float* y_dev,
             __global float* z_dev,
             __global float* vx_dev,
             __global float* vy_dev,
             __global float* vz_dev,
             __global float* t_dev,
             __constant float* boxsize_dev,
             __constant float* OMEGA,
             __constant int* num_bodies_dev
             ){

  int id,k, inc;
  __local float t_local;

  id = get_local_id(0);
  if (id == 0){
    t_local = *t_dev;
  }
  barrier(CLK_LOCAL_MEM_FENCE);
  float t = t_local;
  id += get_group_id(0)*get_local_size(0);
  inc = get_global_size(0);
  float boxsize_x = *boxsize_dev;
  float boxsize_y = *boxsize_dev;
  float boxsize_z = *boxsize_dev;
  //offset of origin to touching six blocks to the right of main six blocks
  float offsetp1 = -fmod(-1.5f*(*OMEGA)*boxsize_x*t+boxsize_y/2.f,boxsize_y)-↩
      boxsize_y/2.f;
```

```
  //offset of origin to touching six blocks to the left of main six blocks
  float offsetm1 = -fmod( 1.5f*(*OMEGA)*boxsize_x*t-boxsize_y/2.f,boxsize_y)+↩
      boxsize_y/2.f;
  for (k = id; k < *num_bodies_dev; k += inc){
    // Radial
    while(x_dev[k] > boxsize_x/2.f){
      x_dev[k] -= boxsize_x;
      y_dev[k] += offsetp1;
      vy_dev[k] += 3.f/2.f*(*OMEGA)*boxsize_x;
    }
    while(x_dev[k] < -boxsize_x/2.f){
      x_dev[k] += boxsize_x;
      y_dev[k] += offsetm1;
      vy_dev[k] -= 3.f/2.f*(*OMEGA)*boxsize_x;
    }
    // Azimuthal
    while(y_dev[k] > boxsize_y/2.f){
      y_dev[k] -= boxsize_y;
    }
    while(y_dev[k] < -boxsize_y/2.f){
      y_dev[k] += boxsize_y;
    }
    // Vertical
    while(z_dev[k] > boxsize_z/2.f){
      z_dev[k] -= boxsize_z;
    }
    while(z_dev[k] < -boxsize_z/2.f){
      z_dev[k] += boxsize_z;
    }
  }
}
```

A.3 Tree Kernel

```
__kernel void cl_tree_add_particles_to_tree(
              __global float* x_dev,
              __global float* y_dev,
              __global float* z_dev,
              __global float* mass_dev,
              __global int* start_dev,
              __global int* children_dev,
              __global int* maxdepth_dev,
              __global int* bottom_node_dev,
              __constant float* boxsize_dev,
              __constant float* rootx_dev,
              __constant float* rooty_dev,
              __constant float* rootz_dev,
              __constant int* num_nodes_dev,
              __constant int* num_bodies_dev
              )
{
```

```
int i, j, k, parent_is_null, inc, child, node, cell, locked, patch, ↩
    maxdepth_thread, depth;
float body_x,body_y,body_z,r, cell_x, cell_y, cell_z;
__local float root_cell_radius, root_cell_x, root_cell_y, root_cell_z;

i = get_local_id(0);

//the first thread in the work group initializes the data
if (i == 0){
  root_cell_radius = *boxsize_dev/2.0f;
  root_cell_x = x_dev[*num_nodes_dev] = *rootx_dev;
  root_cell_y = y_dev[*num_nodes_dev] = *rooty_dev;
  root_cell_z = z_dev[*num_nodes_dev] = *rootz_dev;
  mass_dev[*num_nodes_dev] = -1.0f;
  start_dev[*num_nodes_dev] = 0;
  *bottom_node_dev = *num_nodes_dev;
  *maxdepth_dev = 1;
  //set root children to NULL
  for (k = 0; k < 8; k++)
    children_dev[*num_nodes_dev*8 + k] = -1;
}
barrier(CLK_LOCAL_MEM_FENCE);

maxdepth_thread = 1;
parent_is_null = 1;
inc = get_global_size(0);
i += get_group_id(0)*get_local_size(0);

while (i < *num_bodies_dev){

  //insert new body at root
  if(parent_is_null == 1){
    parent_is_null = 0;
    body_x = x_dev[i];
    body_y = y_dev[i];
    body_z = z_dev[i];

    //root node
    node = *num_nodes_dev;
    depth = 1;
    r = root_cell_radius;
    j = 0;
    if (root_cell_x < body_x) j = 1;
    if (root_cell_y < body_y) j += 2;
    if (root_cell_z < body_z) j += 4;
  }

  child = children_dev[node*8 + j];

  //if child is not a leaf or is not null, we
  //descend the tree untill we get
  //to a NULL or leaf
  while (child >= *num_bodies_dev){
    node = child;
    depth++;
    r *= 0.5f;
    j = 0;
```

```
      if (x_dev[node] < body_x) j = 1;
      if (y_dev[node] < body_y) j += 2;
      if (z_dev[node] < body_z) j += 4;
      child = children_dev[node*8 + j];
    }

    //if child is not locked
    if (child != -2){
      locked = node*8 + j;
      if (child == atom_cmpxchg(&children_dev[locked], child, -2)){ //try to ↩
          lock child

  //if NULL, insert body
  if(child == -1){
    children_dev[locked] = i;
  }

  //create new subtree by moving *bottom_node_dev down one node
  else {
    patch = -1;
    do {
      depth++;
      cell = atomic_sub(bottom_node_dev,1) - 1;

      patch = max(patch,cell) ;

      cell_x = (j & 1) * r;
      cell_y = ((j >> 1) & 1) * r;
      cell_z = ((j >> 2) & 1) * r;
      r *= 0.5f;

      mass_dev[cell] = -1.0f;
      start_dev[cell] = -1;
      cell_x = x_dev[cell] = x_dev[node] -r + cell_x;
      cell_y = y_dev[cell] = y_dev[node] -r + cell_y;
      cell_z = z_dev[cell] = z_dev[node] -r + cell_z;

      for (k = 0; k < 8; k++)
        children_dev[cell*8 + k] = -1;

      if (patch != cell)
        children_dev[node*8 + j] = cell;

      j = 0;
      if (cell_x < x_dev[child]) j = 1;
      if (cell_y < y_dev[child]) j += 2;
      if (cell_z < z_dev[child]) j += 4;
      children_dev[cell*8+j] = child;

      node = cell;
      j = 0;
      if (cell_x < body_x) j = 1;
      if (cell_y < body_y) j += 2;
      if (cell_z < body_z) j += 4;
      child = children_dev[node*8 + j];

    } while ( child >= 0 );
```

```
    children_dev[node*8 + j] = i;

    // after mem_fence, all work items now see the added sub-tree
    mem_fence(CLK_GLOBAL_MEM_FENCE);
    children_dev[locked] = patch;
  }
  maxdepth_thread = max(depth, maxdepth_thread);
  i += inc;
  parent_is_null = 1;
      }
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  atomic_max(maxdepth_dev, maxdepth_thread);
}
```

A.4 Tree Update Gravity Data Kernel

```
__kernel void cl_tree_update_tree_gravity_data(
                      __global float* x_dev,
                __global float* y_dev,
                __global float* z_dev,
                __global float* mass_dev,
                __global int* children_dev,
                __global int* count_dev,
                __global int* bottom_node_dev,
                __constant int* num_nodes_dev,
                __constant int* num_bodies_dev,
                __local int* children_local
                    )
{

  //POTENTIAL SIMPLE OPTIMIZATION: repl ace instances of get_local_id(0) with a↩
       register variable thread_id

  int i, num_children_processed, k, child, inc, num_children_notcalculated, ↩
      count, num_group_threads, local_id;
  float child_mass, cell_mass, cell_x, cell_y, cell_z;
  __local int bottom_node;

  local_id = get_local_id(0);

  if (local_id == 0){
    bottom_node = *bottom_node_dev;
  }
  barrier(CLK_LOCAL_MEM_FENCE);

  inc = get_global_size(0);
  num_group_threads = get_local_size(0);
  num_children_notcalculated = 0;
```

```
  //start processing at the bottom
  k = (bottom_node & (-WAVEFRONT_SIZE)) + local_id + get_group_id(0)*↩
      num_group_threads;
  if (k < bottom_node)
    k += inc;

  while (k <= *num_nodes_dev){

    if (num_children_notcalculated == 0){
      cell_mass = 0.f;
      cell_x = 0.f;
      cell_y = 0.f;
      cell_z = 0.f;
      count = 0;
      num_children_processed = 0;

      for (i = 0; i < 8; i++){
    child = children_dev[k*8 + i];
    if (child >= 0){
      if (i != num_children_processed){
        //move child to the front
        children_dev[k*8+i] = -1;
        children_dev[k*8+num_children_processed] = child;
      }
      children_local[num_children_notcalculated*num_group_threads + local_id] =↩
           child;
      child_mass = mass_dev[child];
      num_children_notcalculated++;

      if(child_mass >= 0.f){
        //cell is ready to be calculated
        num_children_notcalculated--;
        if (child >= *num_bodies_dev){
          count += count_dev[child] - 1; //subtract one
        }
        cell_mass += child_mass;
        cell_x += x_dev[child]*child_mass;
        cell_y += y_dev[child]*child_mass;
        cell_z += z_dev[child]*child_mass;
      }
      num_children_processed++;
    }
      }
      count += num_children_processed;
    }

    //if there are some children not ready
    if (num_children_notcalculated != 0){
      do {
    child = children_local[(num_children_notcalculated - 1)*num_group_threads +↩
         get_local_id(0)];
    child_mass = mass_dev[child];
  //child is ready
  if (child_mass >= 0.f){
      num_children_notcalculated--;
      if (child >= *num_bodies_dev){
```

```
        count += count_dev[child] - 1; //we subtract one b/c ↩
            num_children_processed counts the cell
      }
      cell_mass += child_mass;
      cell_x += x_dev[child] * child_mass;
      cell_y += y_dev[child] * child_mass;
      cell_z += z_dev[child] * child_mass;
    }
      } while ( (child_mass >= 0.f) && (num_children_notcalculated != 0) );
    }

    //all children and subchildren of this node have been accounted for
    //it's time to add everything up for node k
    if (num_children_notcalculated == 0){
      count_dev[k] = count;
      //child_mass is used as a temporary storage device here
      //for the inverse mass
      child_mass = 1.0f/cell_mass;
      x_dev[k]=cell_x*child_mass;
      y_dev[k]=cell_y*child_mass;
      z_dev[k]=cell_z*child_mass;
      //before freeing up this cell by setting a nonzero mass_dev, we must ↩
          first make sure
      //the cell information is loaded into memory first, by using a MEM_FENCE
      mem_fence(CLK_GLOBAL_MEM_FENCE);
      mass_dev[k] = cell_mass;
      k += inc;
    }
  }
}
```

A.5 Tree Sort Kernel

```
__kernel void cl_tree_sort_particles(
          __global int* children_dev,
          __global int* count_dev,
          __global int* start_dev,
          __global int* sort_dev,
          __global int* bottom_node_dev,
          __constant int* num_nodes_dev,
          __constant int* num_bodies_dev
             ){

  int i,k, child, dec, start, bottom;
  __local int bottoms;

  if (get_local_id(0) == 0){
    bottoms = *bottom_node_dev;
  }
  barrier(CLK_LOCAL_MEM_FENCE);
  bottom = bottoms;
```

```
  dec = get_global_size(0);
  k = *num_nodes_dev + 1 - dec + get_local_id(0) + get_group_id(0)*↩
      get_local_size(0);

  while (k >= bottom){
    start = start_dev[k];
    //when the node is ready
    if (start >= 0){
      for (i = 0; i < 8; i++){
  child = children_dev[k*8 + i];
  //non-leaf - set starting place to write in sorted array
  if (child >= *num_bodies_dev){
    start_dev[child] = start;
    start += count_dev[child];
  }
  //leaf - write to sorted array
  else if (child >= 0) {
    sort_dev[start] = child;
    start++;
  }
      }
      k -= dec;
    }
  }
}
```

A.6 Gravity Tree Kernel

```
__kernel void cl_gravity_calculate_acceleration_for_particle(
                    __global float* x_dev,
                    __global float* y_dev,
                    __global float* z_dev,
                    __global float* ax_dev,
                    __global float* ay_dev,
                    __global float* az_dev,
                    __global float* mass_dev,
                    __global int* sort_dev,
                    __global int* children_dev,
                    __global int* maxdepth_dev,
                    __global float* t_dev,
                    __constant float* OMEGA_dev,
                    __constant float* boxsize_dev,
                    __constant int* num_nodes_dev,
                    __constant int* num_bodies_dev,
                    __constant float* inv_opening_angle2_dev,
                    __constant float* softening2_dev,
                    __constant float* G_dev,
                    __local int* children_local,
                    __local int* pos_local,
                    __local int* node_local,
                    __local float* dr2_cutoff_local,
                    __local float* nodex_local,
```

```
                    __local float* nodey_local,
                    __local float* nodez_local,
                    __local float* nodem_local,
                    __local int* wavefront_vote_local
                    ){


  int i, j, l, node, depth, base, sbase, diff, local_id;
  float body_x, body_y, body_z, body_ax, body_ay, body_az, dx, dy, dz, ↩
      temp_register;
  float shiftx, shifty, shiftz;
  __local int maxdepth_local;
  __local float t_local;

#ifdef KAHAN_SUMMATION
  // accumulators for Kahan summation of phase
  float kahanC_ax = 0.f;
  float kahanC_ay = 0.f;
  float kahanC_az = 0.f;
  float kahanT = 0.f;
  float kahanY = 0.f;
#endif

  local_id = get_local_id(0);

  if (local_id == 0){
    maxdepth_local = *maxdepth_dev;
    temp_register = *boxsize_dev;
    t_local = *t_dev;
    dr2_cutoff_local[0] = temp_register * temp_register * (*↩
        inv_opening_angle2_dev);
    for (i  = 1; i < maxdepth_local; i++)
      dr2_cutoff_local[i] = dr2_cutoff_local[i-1] * .25f;
  }

  barrier(CLK_LOCAL_MEM_FENCE);
  float t = t_local;

  if (maxdepth_local <= MAX_DEPTH){
    base = local_id / WAVEFRONT_SIZE;
    sbase = base * WAVEFRONT_SIZE;
    j = base * MAX_DEPTH;

    diff = local_id - sbase;
    if (diff < MAX_DEPTH){
      dr2_cutoff_local[diff + j] = dr2_cutoff_local[diff];
    }
    barrier(CLK_LOCAL_MEM_FENCE);

    for (int k = local_id + get_group_id(0)*get_local_size(0); k < *↩
        num_bodies_dev; k += get_global_size(0)){

      i = sort_dev[k];
      body_x = x_dev[i];
      body_y = y_dev[i];
      body_z = z_dev[i];
```

```
    body_ax = 0.f;
    body_ay = 0.f;
    body_az = 0.f;

    for (int gbx = -1; gbx <= 1; gbx++){
      for (int gby = -1; gby <= 1; gby++){
        for (int gbz = -GBZ_COL; gbz <= GBZ_COL; gbz++){

          cl_boundaries_get_ghostbox(gbx,gby,gbz,&shiftx,&shifty,&shiftz,&↩
              temp_register,&temp_register,&temp_register, *OMEGA_dev, *↩
              boxsize_dev, *boxsize_dev, *boxsize_dev, t);

    depth = j;
    //first thread in wavefront leads the pack by picking the node
    if (sbase == local_id){
      node_local[j] = *num_nodes_dev;
      pos_local[j] = 0;
    }
    mem_fence(CLK_LOCAL_MEM_FENCE);

    while (depth >= j){
      while(pos_local[depth] < 8){
  //first thread in wavefront leads
  if(sbase == local_id){
    node = children_dev[node_local[depth]*8 + pos_local[depth]];
    pos_local[depth]++;
    children_local[base] = node;
    if (node >= 0){
      nodex_local[base] = x_dev[node] + shiftx;
      nodey_local[base] = y_dev[node] + shifty;
      nodez_local[base] = z_dev[node] + shiftz;
      nodem_local[base] = mass_dev[node];
    }
  }
  //the wavefronts do not move forward until they see the new local memory (↩
      mem_fence)
  mem_fence(CLK_LOCAL_MEM_FENCE);
  node = children_local[base];

  //if the node is not null
  if (node >= 0){
    dx = nodex_local[base] - body_x;
    dy = nodey_local[base] - body_y;
    dz = nodez_local[base] - body_z;
    temp_register = dx*dx + dy*dy + dz*dz;

    //the vote
    wavefront_vote_local[local_id] = (temp_register >= dr2_cutoff_local[depth↩
        ]) ? 1 : 0;

    //the first thread in the wavefront adds up the votes
    if (local_id == sbase)
      for(l = 1; l < WAVEFRONT_SIZE; l++)
        wavefront_vote_local[sbase] += wavefront_vote_local[sbase + l];
    mem_fence(CLK_LOCAL_MEM_FENCE);
```

```
      //the node is either a body or the wavefront voted unanimously that the ↩
          node cell is too far away
      if ((node < *num_bodies_dev) || wavefront_vote_local[sbase] >= ↩
          WAVEFRONT_SIZE){

        //if the node isn't the body we are computing the acc for
        if (node != i){
          temp_register = rsqrt(temp_register + *softening2_dev);
          temp_register = *G_dev * nodem_local[base] * temp_register * ↩
              temp_register * temp_register;
#ifdef KAHAN_SUMMATION
          kahanY = dx * temp_register - kahanC_ax;
          kahanT = body_ax + kahanY;
          kahanC_ax = (kahanT - body_ax) - kahanY;
          body_ax = kahanT;

          kahanY = dy * temp_register - kahanC_ay;
          kahanT = body_ay + kahanY;
          kahanC_ay = (kahanT - body_ay) - kahanY;
          body_ay = kahanT;

          kahanY = dz * temp_register - kahanC_az;
          kahanT = body_az + kahanY;
          kahanC_az = (kahanT - body_az) - kahanY;
          body_az = kahanT;
#else
          body_ax += dx * temp_register;
          body_ay += dy * temp_register;
          body_az += dz * temp_register;
#endif
        }
      }

      //one thread voted negative - descend into child cell
      else{
        depth++;
        if(sbase == local_id){
          node_local[depth] = node;
          pos_local[depth] = 0;
        }
        mem_fence(CLK_LOCAL_MEM_FENCE);
      }
    }

    // if child is null then remaining children of this node is null
    // due to the mini-stream compaction in the tree_gravity_update
    // kernel so move back up the tree
    else{
      depth = max(j, depth - 1);
    }
        }
        depth--;
      }
    }
  }
      }
      ax_dev[i] = body_ax;
```

```
      ay_dev[i] = body_ay;
      az_dev[i] = body_az;
    }
  }
}
```

A.7 Integrator Part 2 Kernel

```
  void operator_H012(
       int i,
       float dt,
       float OMEGAZ,
       float OMEGA,
       float sindt,
       float tandt,
       float sindtz,
       float tandtz,
       __global float* x_dev,
       __global float* y_dev,
       __global float* z_dev,
       __global float* vx_dev,
       __global float* vy_dev,
       __global float* vz_dev,
       __global float* ax_dev,
       __global float* ay_dev,
       __global float* az_dev
       )
{
  /* Integrate vertical  motion */
  float zx = z_dev[i] * OMEGAZ;
  float zy = vz_dev[i];

  /* Rotation implemented as 3 shear operators to avoid roundoff errors */
  float zt1 = zx - tandtz*zy;
  float zyt = sindtz*zt1 + zy;
  float zxt = zt1 - tandtz*zyt;
  z_dev[i] = zxt/OMEGAZ;
  vz_dev[i] = zyt;

  float a0 = 2.f*vy_dev[i] + 4.f*x_dev[i]*OMEGA;
  float b0 = y_dev[i]*OMEGA -2.f*vx_dev[i];

  float ys = (y_dev[i]*OMEGA-b0)/2.f;
  float xs = (x_dev[i]*OMEGA-a0);

  float xst1 = xs - tandt*ys;
  float yst = sindt*xst1 + ys;
  float xst = xst1 - tandt*yst;

  x_dev[i] = (xst + a0) / OMEGA;
  y_dev[i] = (yst*2.f + b0) / OMEGA - 3.f/4.f*a0*dt;
  vx_dev[i] = yst;
```

```
  vy_dev[i] = -xst*2.f - 3.f/2.f*a0;
}

void operator_phi1(
       int i,
       float dt,
       __global float* vx_dev,
       __global float* vy_dev,
       __global float* vz_dev,
       __global float* ax_dev,
       __global float* ay_dev,
       __global float* az_dev
       )
{
  //kick step
  vx_dev[i] += ax_dev[i] * dt;
  vy_dev[i] += ay_dev[i] * dt;
  vz_dev[i] += az_dev[i] * dt;
}

__kernel void cl_integrator_part2(
            __global float* x_dev,
            __global float* y_dev,
            __global float* z_dev,
            __global float* vx_dev,
            __global float* vy_dev,
            __global float* vz_dev,
            __global float* ax_dev,
            __global float* ay_dev,
            __global float* az_dev,
            __global float* t_dev,
            __constant int* num_bodies_dev,
            __constant float* dt_dev,
            __constant float* OMEGA_dev,
            __constant float* OMEGAZ_dev,
            __constant float* sindt_dev,
            __constant float* tandt_dev,
            __constant float* sindtz_dev,
            __constant float* tandtz_dev
          )
{
  __local float t_local;
  int id,k, inc;

  id = get_local_id(0);
  if (id == 0){
    t_local = *t_dev;
  }
  barrier(CLK_LOCAL_MEM_FENCE);
  float t = t_local;

  id += get_group_id(0)*get_local_size(0);
  inc = get_global_size(0);

  for (k = id; k < *num_bodies_dev; k += inc){
    operator_phi1(k, *dt_dev, vx_dev, vy_dev, vz_dev, ax_dev, ay_dev, az_dev);
    operator_H012(k,
```

```
      *dt_dev,
      *OMEGA_dev,
      *OMEGAZ_dev,
      *sindt_dev,
      *tandt_dev,
      *sindtz_dev,
      *tandtz_dev,
      x_dev,
      y_dev,
      z_dev,
      vx_dev,
      vy_dev,
      vz_dev,
      ax_dev,
      ay_dev,
      az_dev
      );
  }

  t += *dt_dev/2.f;
  if (id == 0)
    *t_dev = t;
```

A.8 Tree Kernel (without mass data)

```
__kernel void cl_tree_add_particles_to_tree_no_mass(
          __global float* x_dev,
          __global float* y_dev,
          __global float* z_dev,
          __global int* count_dev,
          __global int* start_dev,
          __global int* children_dev,
          __global int* maxdepth_dev,
          __global int* bottom_node_dev,
          __constant float* boxsize_dev,
          __constant float* rootx_dev,
          __constant float* rooty_dev,
          __constant float* rootz_dev,
          __constant int* num_nodes_dev,
          __constant int* num_bodies_dev
          )
{
  int i, j, k, parent_is_null, inc, child, node, cell, locked, patch, ↩
      maxdepth_thread, depth;
  float body_x,body_y,body_z,r, cell_x, cell_y, cell_z;
  __local float root_cell_radius, root_cell_x, root_cell_y, root_cell_z;

  i = get_local_id(0);
  if (i == 0){
    root_cell_radius = *boxsize_dev/2.0f;
    root_cell_x = x_dev[*num_nodes_dev] = *rootx_dev;
    root_cell_y = y_dev[*num_nodes_dev] = *rooty_dev;
```

```
    root_cell_z = z_dev[*num_nodes_dev] = *rootz_dev;
    count_dev[*num_nodes_dev] = -1;
    start_dev[*num_nodes_dev] = 0;
    *bottom_node_dev = *num_nodes_dev;
    *maxdepth_dev = 1;
    //set root children to NULL
    for (k = 0; k < 8; k++)
      children_dev[*num_nodes_dev*8 + k] = -1;
  }
  barrier(CLK_LOCAL_MEM_FENCE);

  maxdepth_thread = 1;
  parent_is_null = 1;
  inc = get_global_size(0);
  i += get_group_id(0)*get_local_size(0);

  while (i < *num_bodies_dev){
    //insert new body at root
    if(parent_is_null == 1){
      parent_is_null = 0;
      body_x = x_dev[i];
      body_y = y_dev[i];
      body_z = z_dev[i];

      //root node
      node = *num_nodes_dev;
      depth = 1;
      r = root_cell_radius;
      j = 0;
      if (root_cell_x < body_x) j = 1;
      if (root_cell_y < body_y) j += 2;
      if (root_cell_z < body_z) j += 4;
    }

    child = children_dev[node*8 + j];

    //if child is not a leaf we
    //descend the tree untill we get
    //to a NULL or filled leaf
    while (child >= *num_bodies_dev){
      node = child;
      depth++;
      r *= 0.5f;
      j = 0;
      if (x_dev[node] < body_x) j = 1;
      if (y_dev[node] < body_y) j += 2;
      if (z_dev[node] < body_z) j += 4;
      child = children_dev[node*8 + j];
    }

    //if child is not locked
    if (child != -2){
      locked = node*8 + j;
      if (child == atom_cmpxchg(&children_dev[locked], child, -2)){ //try to ↩
          lock child

      //if NULL, insert body
```

```
      if(child == -1){
        children_dev[locked] = i;
      }

      //create new subtree by moving *bottom_node_dev down one node
      else {
        patch = -1;
        do {
        depth++;
          cell = atomic_sub(bottom_node_dev,1) - 1;

          patch = max(patch,cell) ;

          cell_x = (j & 1) * r;
          cell_y = ((j >> 1) & 1) * r;
          cell_z = ((j >> 2) & 1) * r;
          r *= 0.5f;

          count_dev[cell] = -1;
        start_dev[cell] = -1;
          cell_x = x_dev[cell] = x_dev[node] -r + cell_x;
          cell_y = y_dev[cell] = y_dev[node] -r + cell_y;
          cell_z = z_dev[cell] = z_dev[node] -r + cell_z;

          for (k = 0; k < 8; k++)
      children_dev[cell*8 + k] = -1;

          if (patch != cell)
      children_dev[node*8 + j] = cell;

          j = 0;
          if (cell_x < x_dev[child]) j = 1;
          if (cell_y < y_dev[child]) j += 2;
          if (cell_z < z_dev[child]) j += 4;
          children_dev[cell*8+j] = child;

          node = cell;
          j = 0;
          if (cell_x < body_x) j = 1;
          if (cell_y < body_y) j += 2;
          if (cell_z < body_z) j += 4;
          child = children_dev[node*8 + j];

        } while ( child >= 0 );

        children_dev[node*8 + j] = i;

        // after mem_fence, all work items now see the added sub-tree
        mem_fence(CLK_GLOBAL_MEM_FENCE);
      children_dev[locked] = patch;
      }
    maxdepth_thread = max(depth, maxdepth_thread);
      i += inc;
      parent_is_null = 1;
    }
    }
      barrier(CLK_LOCAL_MEM_FENCE);
```

```
    }
  atomic_max(maxdepth_dev, maxdepth_thread);
}
```

A.9 Tree Collision Data Update Kernel

```
__kernel void cl_tree_update_tree_collisions_data(
               __global int* children_dev,
               __global int* count_dev,
               __global int* bottom_node_dev,
               __constant int* num_nodes_dev,
               __constant int* num_bodies_dev,
               __local int* children_local
                   )
{

  int i, num_processed, k, child, inc, num_notcalculated, count, child_count, ↩
      num_group_threads, local_id;
  __local int bottom_node;

  local_id = get_local_id(0);

  if (local_id == 0){
    bottom_node = *bottom_node_dev;
  }
  barrier(CLK_LOCAL_MEM_FENCE);

  inc = get_global_size(0);
  num_group_threads = get_local_size(0);
  num_notcalculated = 0;

  //start processing at the bottom
  k = (bottom_node & (-WAVEFRONT_SIZE)) + local_id + get_group_id(0)*↩
      num_group_threads;
  while (k < bottom_node)
    k += inc;

  while (k <= *num_nodes_dev){

    //when there is no children on the stack that aren't ready
    if (num_notcalculated == 0){
      count = 0;
      num_processed = 0;

      for (i = 0; i < 8; i++){
    child = children_dev[k*8 + i];
    if (child >= 0){

      if (i != num_processed){
        //move child to the front
        children_dev[k*8+i] = -1;
        children_dev[k*8+num_processed] = child;
```

```
      }
      children_local[num_notcalculated*num_group_threads + local_id] = child;
      child_count = count_dev[child];
      num_notcalculated++;

      if(child_count != -1){
        //cell is ready to be calculated
        num_notcalculated--;
        if (child >= *num_bodies_dev){
        //subtract one because we will be adding num_processed at the end
          count += child_count - 1;
        }
      }
      num_processed++;
    }
      }
      count += num_processed;
    }

   //if there are some children not ready
    if (num_notcalculated != 0){
      do {
    child = children_local[(num_notcalculated - 1)*num_group_threads + ↩
        get_local_id(0)];
    child_count = count_dev[child];
  //child is ready
  if (child_count != -1){
      num_notcalculated--;
      if (child >= *num_bodies_dev){
        count += child_count - 1; //we subtract one b/c num_processed counts ↩
            the cell
      }
    }
      } while ( (child_count != -1) && (num_notcalculated != 0) );
    }

    //all children and subchildren of this node have been accounted for
    //it's time to add everthing up for node k
    if (num_notcalculated == 0){
      count_dev[k] = count;
      k += inc;
    }
  }
}
```

A.10 Collisions Search Kernel

```
__kernel void cl_collisions_search(
            __global float* x_dev,
            __global float* y_dev,
            __global float* z_dev,
            __global float* vx_dev,
```

```
            __global float* vy_dev,
            __global float* vz_dev,
            __global float* rad_dev,
            __global int* sort_dev,
            __global int* collisions_dev,
            __global int* children_dev,
            __global int* maxdepth_dev,
            __constant float* boxsize_dev,
            __constant int* num_nodes_dev,
            __constant int* num_bodies_dev,
            __local int* children_local,
            __local int* pos_local,
            __local int* node_local,
            __local float* dr_cutoff_local,
            __local float* nodex_local,
            __local float* nodey_local,
            __local float* nodez_local,
            __local float* nodevx_local,
            __local float* nodevy_local,
            __local float* nodevz_local,
            __local float* noderad_local,
            __local int* wavefront_vote_local,
            __constant float* collisions_max2_r_dev,
            __constant float* OMEGA_dev,
            __global float* t_dev
            // __global float* error_dev
            ){

  float body_x, body_y, body_z, /* body_vx, body_vy, body_vz, */ body_rad, dx, ↩
      dy, dz, temp_register, shiftx, shifty, shiftz, shiftvx, shiftvy, shiftvz;
  int i, j, l, node, depth, base, sbase, diff, local_id, gbx, gby, gbz;
  __local int maxdepth_local;
  __local float t_local;

  /* int gbx_offset = *num_bodies_dev; */
  /* int gby_offset = gbx_offset*3; */
  /* int gbz_offset = gby_offset*3; */

  local_id = get_local_id(0);

  if (local_id == 0){
    maxdepth_local = *maxdepth_dev;
    t_local = *t_dev;
    dr_cutoff_local[0] = *boxsize_dev*0.86602540378443;
    for (i = 1; i < maxdepth_local; i++){
      dr_cutoff_local[i] = dr_cutoff_local[i-1] * .5f;
      dr_cutoff_local[i-1] += *collisions_max2_r_dev;
    }
    dr_cutoff_local[maxdepth_local - 1] += *collisions_max2_r_dev;
#ifdef ERROR_CHECK
    if (maxdepth_local > MAXDEPTH){
    *error_dev = -2;
  }
#endif
  }
  barrier(CLK_LOCAL_MEM_FENCE);
```

```
  float t = t_local;

  if (maxdepth_local <= MAX_DEPTH){
    base = local_id / WAVEFRONT_SIZE;
    sbase = base * WAVEFRONT_SIZE;
    j = base * MAX_DEPTH;

    diff = local_id - sbase;
    if (diff < MAX_DEPTH){
      dr_cutoff_local[diff + j] = dr_cutoff_local[diff];
    }
    barrier(CLK_LOCAL_MEM_FENCE);

    for (int k = local_id + get_group_id(0)*get_local_size(0); k < *↩
        num_bodies_dev; k += get_global_size(0)){

      i = sort_dev[k];
      body_x = x_dev[i];
      body_y = y_dev[i];
      body_z = z_dev[i];
      /* body_vx = vx_dev[i]; */
      /* body_vy = vy_dev[i]; */
      /* body_vz = vz_dev[i]; */
      body_rad = rad_dev[i];

      for (gbx = -1; gbx <= 1; gbx++){
  for (gby = -1; gby <= 1; gby++){
    for (gbz = -GBZ_COL; gbz <= GBZ_COL; gbz++){

      //send shifts to function as pointers, get rid of struct
      cl_boundaries_get_ghostbox(gbx,gby,gbz,&shiftx,&shifty,&shiftz,&shiftvx,&↩
          shiftvy,&shiftvz, *OMEGA_dev,*boxsize_dev, *boxsize_dev, *boxsize_dev↩
          , t);

      depth = j;
      //first thread in wavefront leads
      if (sbase == local_id){
        node_local[j] = *num_nodes_dev;
        pos_local[j] = 0;
      }
      mem_fence(CLK_LOCAL_MEM_FENCE);

      //initialize collisions array
      collisions_dev[i + (*num_bodies_dev)*(gbx+1) + (*num_bodies_dev*3)*(gby↩
          +1) + (*num_bodies_dev*9)*GBZ_COL*(gbz+1)] = -1;

      while (depth >= j){
        while(pos_local[depth] < 8){
    //first thread in wavefront leads
    if(sbase == local_id){
      node = children_dev[node_local[depth]*8 + pos_local[depth]];
      pos_local[depth]++;
      children_local[base] = node;
      if (node >= 0){
        nodex_local[base] = x_dev[node];
        nodey_local[base] = y_dev[node];
        nodez_local[base] = z_dev[node];
```

```
        if (node < *num_bodies_dev){
          nodevx_local[base] = vx_dev[node];
          nodevy_local[base] = vy_dev[node];
          nodevz_local[base] = vz_dev[node];
          noderad_local[base] = rad_dev[node];
        }
      }
    }
    mem_fence(CLK_LOCAL_MEM_FENCE);

    //each wavefront member grabs the node the wavefront leader put in local ↩
        memory
    node = children_local[base];

    //if the node is not null
    if (node >= 0){
      dx = (body_x + shiftx) - nodex_local[base];
      dy = (body_y + shifty) - nodey_local[base];
      dz = (body_z + shiftz) - nodez_local[base];
      temp_register = dx*dx + dy*dy + dz*dz;
      // if it's a leaf cell
      if (node < *num_bodies_dev)
        {
          //if the node isn't the same body, check for a collision
          if (node != i){
      if ( temp_register <=  (body_rad + noderad_local[base])*(body_rad + ↩
          noderad_local[base]) && ((vx_dev[i] + shiftvx) - nodevx_local[base])*↩
          dx + ((vy_dev[i] + shiftvy) - nodevy_local[base])*dy + ( (vz_dev[i] +↩
           shiftvz) - nodevz_local[base])*dz < 0){
        collisions_dev[i + (*num_bodies_dev)*(gbx+1) + (*num_bodies_dev*3)*(gby↩
            +1) + (*num_bodies_dev*9)*GBZ_COL*(gbz+1)] = node;
      }
          }
        }
      else{
        wavefront_vote_local[local_id] = (temp_register >= (body_rad + ↩
            dr_cutoff_local[depth])*(body_rad + dr_cutoff_local[depth])) ? 1 : ↩
            0;
        if (local_id == sbase)
          for(l = 1; l < WAVEFRONT_SIZE; l++)
      wavefront_vote_local[sbase] += wavefront_vote_local[sbase + l];
        //this blocks the warp from moving forward since they must wait for ↩
            their leader (local_id = sbase)
        mem_fence(CLK_LOCAL_MEM_FENCE);
              //the warp votes whether or not to descend. If one member of the ↩
                  warp votes descend, the
        //warp descends
        if (wavefront_vote_local[sbase] < WAVEFRONT_SIZE){
          depth++;
          if(sbase == local_id){
      node_local[depth] = node;
      pos_local[depth] = 0;
          }
          mem_fence(CLK_LOCAL_MEM_FENCE);
        }
      }
    }
```

```
    //if child is null then remaining children of this node is null so move ↩
        back up the tree
    else{
      depth = max(j, depth - 1);
    }
        }
        depth--;
      }
    }
  }
      }
    }
  }
}
```

A.11 Collisions Resolve Kernel

```
__kernel void cl_collisions_resolve(
            __global float* x_dev,
            __global float* y_dev,
            __global float* z_dev,
            __global float* vx_dev,
            __global float* vy_dev,
            __global float* vz_dev,
            __global float* mass_dev,
            __global float* rad_dev,
            __global int* sort_dev,
            __global int* collisions_dev,
            __constant int* num_bodies_dev,
            __constant float* minimum_collision_velocity_dev,
            __constant float* OMEGA_dev,
            __constant float* boxsize_dev,
            __global float* t_dev
             )

{

  __local float t_local;
  float shiftx, shifty, shiftz, shiftvx, shiftvy, shiftvz;
  int i,body1,body2;
  int gbx_offset = *num_bodies_dev;
  int gby_offset = gbx_offset*3;
  int gbz_offset = gby_offset*3*GBZ_COL;
  int id = get_local_id(0) + get_group_id(0)*get_local_size(0);
  int inc = get_global_size(0);

  if (get_local_id(0) == 0){
    t_local = *t_dev;
  }
  barrier(CLK_LOCAL_MEM_FENCE);
  float t = t_local;
```

```
for (i = id; i < *num_bodies_dev; i += inc){
  for (int gbx = -1; gbx <= 1; gbx++){
    for (int gby = -1; gby <= 1; gby++){
for (int gbz = -GBZ_COL; gbz <= GBZ_COL; gbz++){
  body1 = sort_dev[i];
  body2 = collisions_dev[body1 + gbx_offset*(gbx+1) + gby_offset*(gby+1) + ↩
      gbz_offset*(gbz+1)];
        //check to make sure no duplicates
  if ( body2 > body1){
    cl_boundaries_get_ghostbox(gbx,gby,gbz, &shiftx, &shifty, &shiftz, &↩
        shiftvx, &shiftvy, &shiftvz, *OMEGA_dev, *boxsize_dev, *boxsize_dev, ↩
        *boxsize_dev, t);

    float x21 = x_dev[body1] + shiftx - x_dev[body2];
    float y21 = y_dev[body1] + shifty - y_dev[body2];
    float z21 = z_dev[body1] + shiftz - z_dev[body2];

    float vx21 = vx_dev[body1] + shiftvx - vx_dev[body2];
    float vy21 = vy_dev[body1] + shiftvy - vy_dev[body2];
    float vz21 = vz_dev[body1] + shiftvz - vz_dev[body2];

          //get y component of normal (vy21n)
    float angle1 = atan2(z21, y21);
    float sin_angle1 = sin(angle1);
    float cos_angle1 = cos(angle1);
    float vy21n = cos_angle1 * vy21 + sin_angle1 * vz21;
    float y21n = cos_angle1 * y21 + sin_angle1 * z21;

    //the normal unit vector = (cos_angle2, sin_angle2)
    float angle2 = atan2(y21n, x21);
    float cos_angle2 = cos(angle2);
    float sin_angle2 = sin(angle2);
    float vx21nn = cos_angle2 * vx21 + sin_angle2 * vy21n;

    //calculate coefficient of restitution contribution to the normal ↩
        component
    float dvx2 = -(1.f + coefficient_of_restitution_bridges(vx21nn))*vx21nn;
    float body1_r = rad_dev[body1];
    float body2_r = rad_dev[body2];
    float minr = (body1_r > body2_r)? body2_r:body1_r;
    float maxr = (body1_r > body2_r)? body2_r:body1_r;
    float mindv = minr*(*minimum_collision_velocity_dev);
    float r = sqrt(x21*x21 + y21*y21 + z21*z21);

    mindv *= 1.f-(r-maxr)/minr;
    if (mindv > maxr*(*minimum_collision_velocity_dev))
      mindv = maxr*(*minimum_collision_velocity_dev);
    if (dvx2 < mindv)
      dvx2 = mindv;

    float dvx2n = cos_angle2 * dvx2;
    float dvy2n = sin_angle2 * dvx2;
    float dvy2nn = cos_angle1 * dvy2n;
    float dvz2nn = sin_angle1 * dvy2n;

    float body1_mass = mass_dev[body1];
```

```
      float body2_mass = mass_dev[body2];

      float prefactor = body1_mass/(body1_mass + body2_mass);
      vx_dev[body2] -= prefactor*dvx2n;
      vy_dev[body2] -= prefactor*dvy2nn;
      vz_dev[body2] -= prefactor*dvz2nn;

      prefactor = body2_mass/(body1_mass + body2_mass);
      vx_dev[body1] += prefactor*dvx2n;
      vy_dev[body1] += prefactor*dvy2nn;
      vz_dev[body1] += prefactor*dvz2nn;
    }
  }
      }
    }
  }
}
```