Development of a convex polyhedral discrete element simulation framework for NVIDIA Kepler based GPUs.

Nicolin Govender*[1,2], Daniel N Wilke[2], Schalk Kok[2], Rosanne Els[3]

[1] *Advanced Mathematical Modelling CSIR, Pretoria 0001, South Africa*

[2] *University of Pretoria, Department of Mechanical and Aeronautical Engineering, Pretoria 0001, South Africa*

[3] *University of Kwazulu-Natal, School of Mathematics, Statistics and Computer Science, Pietermaritzburg 3201, South Africa*

Abstract

Understanding the dynamical behavior of Granular Media (GM) is extremely important to many industrial processes. Thus simulating the dynamics of GM is critical in the design and optimization of such processes. However, the dynamics of GM is complex in nature and cannot be described by a closed form solution for more than a few particles. A popular and successful approach in simulating the underlying dynamics of GM is by using the Discrete Element Method (DEM). Computational viable simulations are typically restricted to a few particles with realistic complex interactions or a larger number of particles with simplified interactions. This paper introduces a novel DEM based particle simulation code (BLAZE-DEM) that is capable of simulating millions of particles on a desktop computer utilizing a NVIDIA Kepler Graphical Processor Unit (GPU) via the CUDA programming model. The GPU framework of BLAZE-DEM is limited to applications that require large numbers of particles with simplified interactions such as hopper flow which exhibits task level parallelism that can be exploited on the GPU. BLAZE-DEM also performs real-time visualization with interactive capabilities. In this paper we discuss our GPU framework and validate our code by comparison between experimental and numerical hopper flow.

Keywords: GPU, DEM, Polyhedra, Large-scale DEM, Granular Media, Nvidia-Kepler.

1

1. Introduction

1.1. Background and Motivation

Transport processes involving Granular Media (GM) occur in many areas of science and engineering over a variety of length scales. Thus understanding the dynamical behavior of GM is central to a large number of engineering disciplines with applications in mining, agriculture and various other fields [1, 2, 3, 4]. Methods belonging to the Discrete Element Method (DEM) family which treats granular material as a system of individual particles, as opposed to a continuum description which averages particle properties, has shown the most promise [5]. The DEM approach which uses a local constitutive law to determine the forces between two contacting particles and consequently the resultant motion of all particles in the system, was first described by Cundall and Strack [6]. The

[1] *Corresponding Author.
Email address: govender.nicolin@gmail.com

DEM is however computationally expensive as all particles in the system have to be checked for contact at each time step. This involves a considerable number of calculations depending on particle geometry and number [7]. To reduce computational cost, particle shape is often approximated using spheres, for which contact detection is trivial. This approximation however results in the system exhibiting different mechanical behavior to reality as discussed by Latham and Munjiza [8, 9]. The clumped-sphere approach [10] provides a better description of shape by using a number of spheres to represent a particle. However, this approach is limited in the number of particles and introduces non-physical artifacts into the simulation, as discussed by Horner [11].

In modeling GM correctly there are two general aspects that must be taken into consideration:

1. Particle shape.
2. Detailed physics interaction between particles.

The Graphics Processor Unit (GPU) offers cluster type performance on a desktop computer at a fraction of the cost, and is well suited to computations that can be executed in parallel resulting in a performance benefit over the traditional CPU [12]. Radeke and Glasser [7] utilize the GPU to simulate powder mixing taking into account detailed particle interactions between spherical particles. They report that a one minute simulation of one million spherical particles requires 96 hours computing time using a single GPU. Longmore et.al [13] take into account particle shape by using multiple spheres to represent a sand grain with simple particle particle interactions and are able to simulate 256 thousand sand grains at 120 FPS on the GPU. This significant improvement in performance suggests that detailed particle interactions which requires particle contact history is costly on the memory constrained GPU. Thus our DEM framework focuses on the accurate representation of particle shape while using simplified interaction models that are suited to parallel implementation. Such a model finds application in particle flow problems where a simplified physics model can capture the dynamical bulk behavior of the system [13, 14].

Polyhedral shaped particles represent most GM accurately and hence exhibit similar mechanical behavior to that of the actual system [15, 16]. However, the number of polyhedral particles that can be simulated on current CPUs is limited [17, 18], with the largest simulations containing at most a few hundred thousand convex polyhedra [19]. In Nassauer and Liedke's work, 800 polyhedra are simulated with detailed particle interactions (1 FPS) using a parallel CPU implementation. To the best of the authors knowledge there has been no GPU implementations for polyhedral shaped particles. This paper is intended to be a feasibility study to illustrate a new performance level of DEM by utilizing a physics model that is suited to the GPU while taking into account detailed particle shape. BLAZE-DEM requires 3 minutes computational time for a one minute simulation of one million spheres (55 FPS), and 150 minutes for one million convex polyhedra on a single GPU (0.9 seconds per time-step) using the simple physics model described by Longmore et.al [13] and Bell et.al [14]. In this paper we develop a GPU orientated DEM environment and determine if it is useful in simulating hopper flow problems.

1.2. GPU

Driven by the demand for real-time 3D graphics with prices kept low due to high selling volumes from the consumer gaming market, the programmable Graphic Processor Unit (GPU) has evolved into a multi-core, multi-threaded processor which offers cluster type performance at a fraction of the cost [20]. Figure 1 shows the hardware design of the CPU and GPU processor chips. We see a major difference in the number of cores and threads present on each chip. CPU cores are designed to be general purpose, hence they are able to do complex logical operations such as running an operating system while being able to perform arithmetical operations. The closest equivalent of a CPU core on a GPU is a Streaming Multi-Processor (SM) which has most of its transistors as dedicated Arithmetic Logic Units (ALU) rather than control and cache in the case of the CPU [21]. GPUs are designed for graphics rendering which involves the manipulation of millions of pixels simultaneously, requiring many parallel algebraic operations hence the large amount of ALUs.

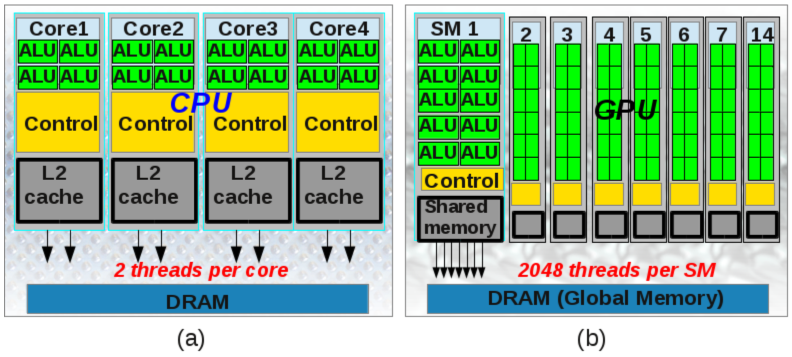
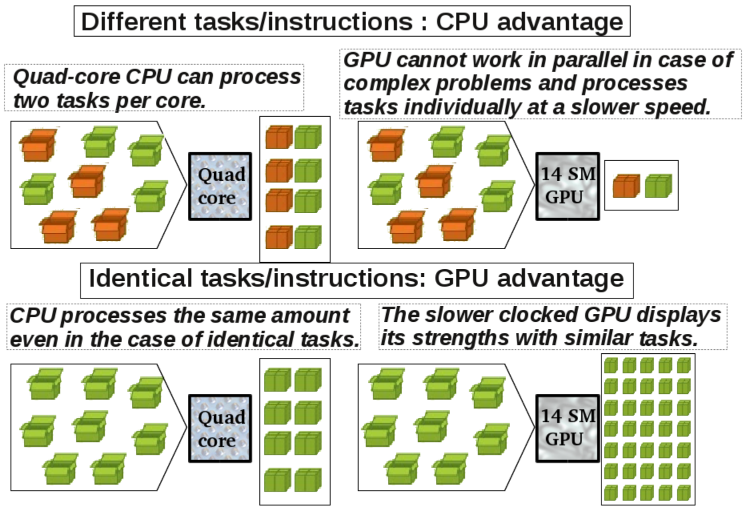


Figure 1: a) Quad core Intel CPU and b) NVIDIA Kepler GPU Chip layouts.

Figure 2 illustrates the type of tasks that each unit excels at performing. The limited GPU outperforms the versatile CPU in spite of the considerably lower clock rate when processing similar data packets. Each CPU core is capable of launching two threads which can work independently and perform complex logical operations. Each SM on a Kepler GK110 GPU can launch 2048 threads which are only capable of performing the same task (Single Instruction Multiple Data (SIMD))[22]. The NVIDIA Kepler architecture has a lower clock speed than the previous generation Fermi architecture but has more SMs, which benefits applications that have thread level parallelism [21]. In-order to realize a speed up on the GPU we need to ensure that our DEM algorithm is completely decoupled and expressed as a SIMD problem, in that we carry out the same instructions on different data elements which are particles in our case.



Different tasks/instructions : CPU advantage
Quad-core CPU can process two tasks per core.
GPU cannot work in parallel in case of complex problems and processes tasks individually at a slower speed.
Quad core
14 SM GPU
Identical tasks/instructions: GPU advantage
CPU processes the same amount even in the case of identical tasks.
The slower clocked GPU displays its strengths with similar tasks.
Quad core
14 SM GPU

Figure 2: Comparison between an i7 Quad-core CPU and NVIDIA GK110 GPU task processing.

In this paper we exploit the computational power of the GPU via the NVIDIA developed CUDA programming model [22], which allows us to issue commands to the GPU from C++ code as opposed to a graphics language like OpenGL. The CUDA programming model batches threads into blocks (max 1024 threads) for execution on a SM. Threads within blocks can access fast shared memory with each thread in turn having access to its own 32 bit registers (fastest memory available). CUDA allows us to create thousands of thread blocks containing millions of threads which get scheduled for execution on the hardware as SMs become available (we don't have control of the execution order of blocks). The execution of a block will only complete once all threads within the block have reached an end point. This is very important and requires us to design algorithms that require similar times to complete for all threads to best utilize the parallelism on the GPU. The GPU has two memory spaces: on-chip memory (shared memory and registers) which are very fast but limited in size and scope; and off-chip memory (global-memory) which is much more plentiful and can be accessed by all SMs as well as the CPU. Global-memory is however about one hundred times slower than on-chip memory and can cause major performance degradation if not used efficiently and correctly.

1.3. Discrete Element Method

The flow-diagram in Figure 3 describes the DEM process that we model. We only consider simple interactions between particles that are embarrassingly parallel to ensure an efficient GPU implementation.

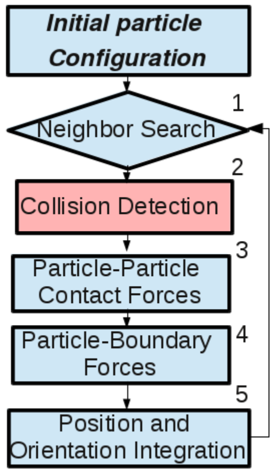

Initial particle Configuration
1
Neighbor Search
2
Collision Detection
3
Particle-Particle Contact Forces
4
Particle-Boundary Forces
5
Position and Orientation Integration

Figure 3: Flow chart of DEM simulation procedure.

This simplistic modeling of particle interactions is sufficient for simulating the bulk behavior of GM flow as discussed by Bell [14] and Longmore[13]. By simulating just GM flow we are also able to make certain assumptions that drastically decreases the computational cost while still capturing the macroscopic details of the physics, which is of interest. An assumption in many DEM simulations is that particles are considered to be perfectly rigid for the duration of a simulation. In reality perfectly rigid particles do not exist, as all bodies will experience (to some extent) local deformations during contact. These deformations however occur on a time scale which is much smaller than what is required for capturing the macroscopic behavior of a system. Thus it is often sufficient to use a constitutive law, such as a linear spring to model contact forces. Computing the time evolution of the system requires us to solve simultaneously Newton's equations of motion for all contacting particles, which on current hardware (2013) is only possible for a few thousand rigid bodies. Thus we assume that there are only binary contacts between particles at any given time. The total force acting on a particle is obtained by summing the individual contributions of all the binary contacts of a particle per time-step, as illustrated in Figure 4. This is a good approximation of reality provided the particles are of a similar size and move very little during a time step.

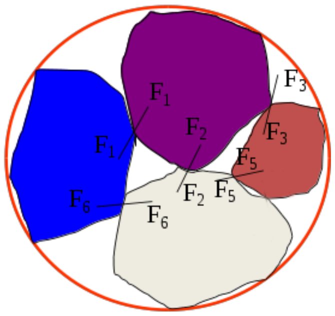


Figure 4: DEM force assumption.

In summary assumptions that we make are :

1. Rigid bodies.
2. Binary contact.
3. Local short-range interactions.
4. Single point contact and Columbic friction model.
5. Similar particle size.
6. Convex polyhedral particles (Section 2).

These assumptions result in a system that is completely decoupled and can be expressed as a Lagrangian type process in which we are able simulate the motion of individual particles independently of each other [6]. Further details about the processes depicted in Figure 3 are given in Sections 2-4.

2. Collision Detection

2.1. Particle representation and storage

By restricting our analysis to only convex particles we can represent a polyhedra as a collection of half-spaces $f_i(\vec{\mathbf{n}}, \vec{\mathbf{c}})$ as illustrated in Figure 5, in which we use the faces as half-spaces [23]. We summarize this result as :

The half-space subtended by a face of a convex polyhedra completely partitions space into two distinct regions:

The region $f_i(\vec{\mathbf{n}}, \vec{\mathbf{c}}) \leq 0$ *as indicated in Figure 5, containing the entire polygon.*

The region $f_i(\vec{\mathbf{n}}, \vec{\mathbf{c}}) > 0$ *an infinite half-space in the direction indicated by the normal to the plane.*

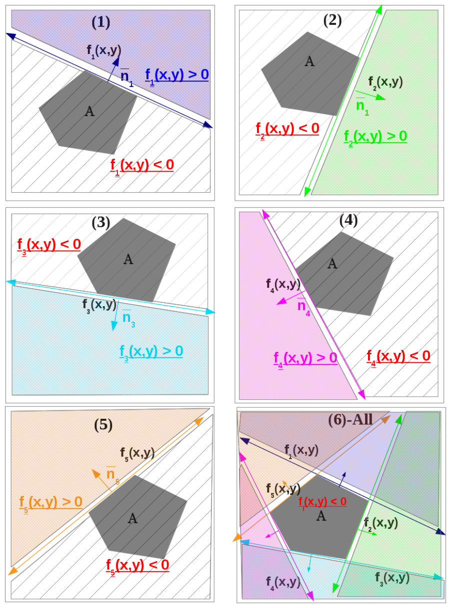
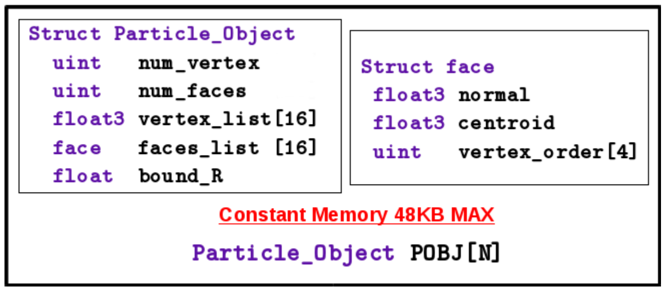


Figure 5: Planar polygon representation.

The choice of faces as half-spaces results in a minimalistic representation that allows us to store data in the very fast but limited constant memory on the GPU. Each particle type is stored as a **Particle_Object** as illustrated in Figure 6. Vertex information is stored as an array of vectors (**vertex_list**). We store each face plane as **face** struct which contains the normal and centroid of the face. We also store the indcies of the vertexes (**vertex_order**) that make up the face as references to **vertex_list**, which is required for narrow phase collision detection. The radius of a sphere that bounds the polyhedra is stored in the variable **bound_R** which we will use for culling in the broad phase.



```
Struct Particle_Object
  uint    num_vertex
  uint    num_faces
  float3  vertex_list[16]
  face    faces_list [16]
  float   bound_R
```

```
Struct face
 float3 normal
 float3 centroid
 uint   vertex_order[4]
```

Constant Memory 48KB MAX

```
Particle_Object POBJ[N]
```

Figure 6: Particle and World object representation.

The major bottle-neck on the GPU is memory utilization. We thus need to ensure that we keep memory transactions to a minimum and utilize the different memory spaces available on the GPU to achieve the best possible performance. For each particle we need to store 4 kinematic parameters (position **P**, velocity **v**, orientation **Q**, angular velocity **w**).

The only option we have for storing this information on the GPU is global memory, which is very slow. However we can minimize the impact on performance by minimizing the number of transactions we have to make by ensuring memory transactions are coalesced. Consider Figure 7, which shows two types of commonly used data structures:

1. Array Of Structures (AOS): all kinematic parameters for each particle is stored in adjacent memory locations.
2. Structure Of Arrays (SOA): each kinematic array for all particles is stored in adjacent memory locations

To gauge the effective performance of the two representations on the GPU, we ran a simulation of 2 million particles and found that **AOS is three times slower** than SOA. We see better performance with SOA as it allows for coalesced thread access, in that neighboring threads access adjacent memory locations resulting in better utilization of cache, requiring fewer memory transactions.

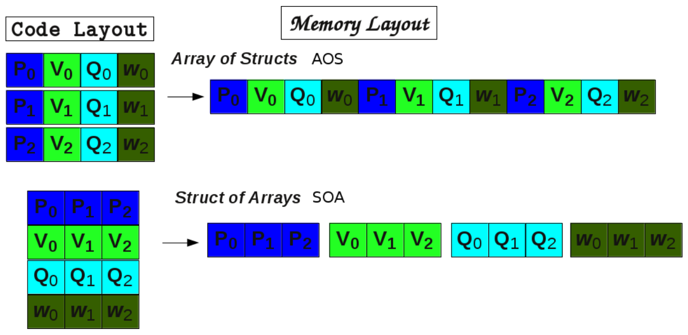
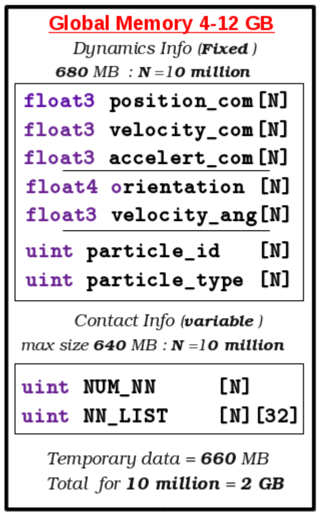


Figure 7: AOS vs SOA data access patterns.

In addition to the particle information we also store the force (acceleration) and information about nearest neighbors in global memory, as depicted in Figure 8. Information that stays fixed for the simulation is stored in high speed constant memory as described in Figure 6. We calculate the evolution of the inertia tensor as required instead of storing it, which is a far more expensive. We also do not store geometric information for each particle as this will be very costly. Rather each particle has a reference to a particle object (particle_type) which contains the geometric information of that particle type as illustrated in Figure 6.



```
float3  position_com [N]
float3  velocity_com [N]
float3  accelert_com [N]
float4  orientation  [N]
float3  velocity_ang [N]
uint  particle_id    [N]
uint  particle_type  [N]
```

```
uint  NUM_NN      [N]
uint  NN_LIST     [N][32]
```

Figure 8: Data representation on the GPU.

2.2. Broad-phase Collision Detection

In DEM simulations particles only interact via mechanical forces. Hence, we only need to consider particles that can physically be in contact with each other. We are thus able to use spatial subdivision to limit the number of particle pairs that need to be checked for collision at each step. In choosing an algorithm to perform spatial subdivision, we have to take into consideration the following requirements:

1. Particles that cannot be in contact must be excluded with minimal computational cost.
2. The algorithm must be suited to the SIMD nature of the GPU.

We thus use a collision grid approach [12] which discritizes geometry into a 3D grid as illustrated by Figure 9 in which a rectangular hopper is shown. Each particle is assigned a discrete grid position given by $\mathrm{GP}^{\mathrm{i}}_{\mathrm{j}} = \mathrm{floor}((\mathrm{P^i}_\mathrm{j} - \mathrm{W_j})/\mathrm{Ncell_j})$, $\mathrm{j} = 1, 3$, where $P^{\mathrm{i}}(\mathrm{x}, \mathrm{y}, \mathrm{z})$ is the center of mass (COM) position of the i^{th}particle in the global coordinate system, $\mathbf{W}$ is the starting point of the grid and $(\mathrm{Ncell_x}, \mathrm{Ncell_y}, \mathrm{Ncell_z})$ number of of cells in each dimension of the $\mathbf{W}$ system as illustrated in Figure 9. The minimum size of a cell is that of the largest particle bound radius in the simulation (particle size can vary by a factor of at most 2 without impacting performance significantly). Based on the number of threads available on the GPU, the number of cells is optimized.

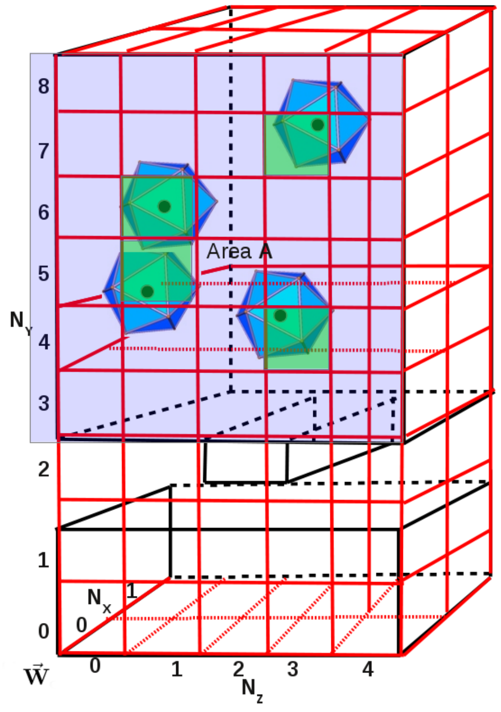


Figure 9: Broad phase collision detection grid.

To minimize memory costs we store each particle's grid position GP as a single integer value given by the mapping function "hashing":

$$\mathrm{P_{Hash}} = \mathrm{GP}_{\mathbf{x}} + \mathrm{GP}_{\boldsymbol{z}} \times \mathrm{Ncell_x} + (\mathrm{GP}_{\mathbf{y}} \times \mathrm{Ncell}_z \times \mathrm{Ncell_y}) \tag{1}$$

The mapping function we have chosen has the following properties:

1. Maps particles which are in the same grid cell to the same hash.
2. Maps particles that are close to each other, to hashes which are close.
3. The particle closest to the origin has the smallest hash and one furthest away the largest.

These properties allow for quick particle look-ups and improved caching. To determine potential contact pairs we only need to consider particles that are in the same cell or in the nearest neighboring cells relative to the particle of interest. Consider two adjacent cells **a** and **b**, since we assign each particle to a thread and execute this in parallel, we cannot use the fact that cell **a** and **b** being neighbors are the same as cell **b** and **a,** which for serial calculations requires only 14 cells to be checked. On some parallel architectures it is possible to use atomic operations which allow different threads to write to the same location in the safe manner and thus exploit the symmetry as in serial calculations. However we have found that this overhead breaks the parallelism and it is cheaper to check all 27 cells in parallel than using atomic operations. We also sort the dynamics info arrays for each particle described in Figure 8 according to the hash to improve memory coherence. We found a **thirty two times speed** up in run-time **when sorting** which includes sorting time.

2.3. Narrow phase

Unlike the trivial contact check for spherical particles, determining contact between two polyhedra is not a trivial matter and can account for as much as 70% of simulation time [16, 17, 18]. We use a novel multiphase heuristic approach that detects the two primary modes of contact between a pair of contacting convex polyhedra, as illustrated in Figure 10. Our approach is based on the idea of a separating plane first described by Cundall [24] to determine if there is contact between convex polyhedra. This approach maps very well to the GPU using Dynamic parallelism [21] on the Kepler generation of GPUs.

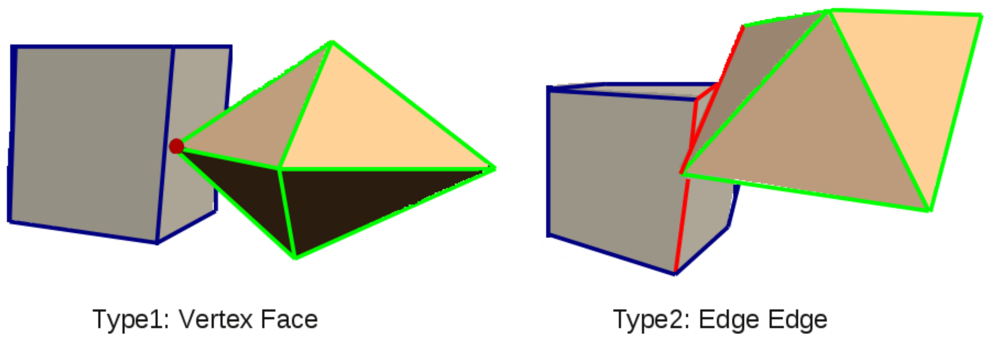


Figure 10: Polyhedra contact types.

If there is contact between two polyhedra we obtain the point of contact $\mathbf{p}_c(x, y, z)$, penetration distance δ and a normal direction $\bar{\mathbf{n}}_s$. For Type 1 contact where there is a single contact point $\bar{\mathbf{n}}_s$ is just that of the contacting face. For Type 2 contact if an edge is contacting a face as depicted in Figure 10 $\bar{\mathbf{n}}_s$ is also just that of the contacting face. However for the extreme case of two contacting edges there is no obvious choice for a normal [24]. We construct a vector from the COM of each polyhedra to $\mathbf{p}_c(x, y, z)$ and use the average of the two vectors to obtain normal that is consistent with Newtons 3^{rd} law.

3. Contact Resolution

3.1. Force Calculations

The most common contact resolution model is the soft-sphere [14] approach of using the amount of inter-penetration between two contacting particles to determine a point force $\mathbf{F} = \mathbf{F}_\mathrm{N} + \mathbf{F}_\mathrm{T}$, as depicted in Figure 11.

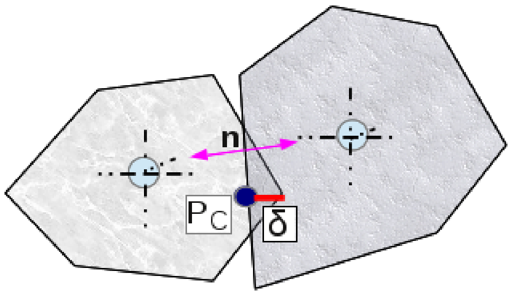


Figure 11: Force interaction model.

We use a linear spring to model the normal force as given by :

$$\mathbf{F}_{\mathrm{N}} = (\mathrm{K}_{\mathrm{N}}\delta)\bar{\mathbf{n}}_{\mathrm{s}} - \eta\mathbf{v}_{21} \tag{2}$$

where δ is the penetration depth, $\mathbf{v}_{21} = \mathbf{v}_1 - \mathbf{v}_2$ is the relative translational velocity, K_{N} the spring stiffness, η the viscous damping coefficient and $\bar{\mathbf{n}}_{\mathrm{s}}$ the normal at the contacting face. There are two approaches to determine the tangential shear force, namely

(i) history independent models which require only knowledge of the current kinematic state and

(ii) history dependent models which require information about previous contacts.

Approach (i) is attractive as it is computationally cheap, but is limited in application. We limit ourselves to the history independent models as they are easily implemented on the GPU and is sufficient for the problems we wish to solve [13, 14].

The tangential friction force is given by :

$$\mathbf{F}_{\mathrm{T}} = -\min\left[\mu(\mathrm{K}_{\mathrm{N}}\delta), \gamma\left\|\mathbf{v}_{\mathrm{T}}\right\|\right]\left(\frac{\mathbf{v}_{\mathrm{T}}}{\|\mathbf{v}_{\mathrm{T}}\|}\right) \tag{3}$$

where $\mathbf{v}_{\mathrm{T}} = (\mathbf{v}_{21} - (\mathbf{v}_{21}.\bar{\mathbf{n}}_{\mathrm{s}}))\bar{\mathbf{n}}_{\mathrm{s}}$ current relative tangential velocity and μ the coefficient of dynamic friction.

In addition to translation forces a particle also experiences a torque as a result of contact given by :

$$\mathbf{\Gamma} = (\mathbf{r} \times \mathbf{F}) \tag{4}$$

where $\mathbf{r}$ is the vector from the COM to the contact point $\mathbf{p}_c(x, y, z)$.

3.2. Numerical Integration

We use the explicit velocity Verlet algorithm, which is second order accurate, to obtain the position $\mathbf{x}$ and velocity $\mathbf{v}$ of a particle i at time k:

$$\mathbf{x}_{\mathrm{k}} = \left[\mathbf{x}_{\mathrm{k}-1} + \mathbf{v}_{\mathrm{k}-1}\triangle\mathrm{t} + \frac{1}{2}\mathbf{a}_{\mathrm{k}-1}\triangle\mathrm{t}^2\right] \tag{5}$$

$$\mathbf{v}_{\mathrm{k}} = \left[\mathbf{v}_{\mathrm{k}-1} + \frac{1}{2}(\mathbf{a}_{\mathrm{k}-1} + \mathbf{a}_{\mathrm{k}})\triangle\mathrm{t}\right] \tag{6}$$

The acceleration $\mathbf{a}$ at time k is given by $\mathbf{a}_\mathrm{k} = \frac{\mathbf{F}^\mathbf{net}_\mathrm{k}}{\mathrm{m}}$ where $\mathbf{F}^\mathbf{net}_\mathrm{k} = \sum \mathbf{F}^\mathrm{ij}_\mathrm{k}$ is the net sum of all binary contact forces experienced by the particle as described in Figure 4. This explicit time integration is expected to be stable when $\triangle\mathrm{t} < \frac{2}{\frac{\sqrt{k_N}}{m}}$.

The angular velocity ω at time k is obtained using the forward Euler integration scheme.

$$\omega_\mathrm{k} = \omega_\mathrm{k-1} + \mathbf{a}^\mathrm{ang}_\mathrm{k}\triangle\mathrm{t}. \tag{7}$$

The angular acceleration $\mathbf{a}^\mathrm{ang}$ at time k is given by $\mathbf{a}^\mathrm{ang}_\mathrm{k} = \mathbf{I}^{-1}_\mathrm{k}\mathbf{\Gamma}^\mathbf{net}_\mathrm{k}$ where $\mathbf{\Gamma}^\mathbf{net}_\mathrm{k} = \sum \mathbf{\Gamma}^\mathrm{ij}$ is the net sum of all the body contact torques experienced by particle i and $\mathbf{I}_\mathrm{k}$ the inertia tensor at time k . The orientation of a particle is represented by a quaternion $\mathbf{q}(w, x, y, z)$ [25]. Quaternions have minimal storage requirements and are more robust than other representations such as Euler angles. The orientation of a particle at time k is given by:

$$\mathbf{q}_\mathrm{k} = \mathbf{q}_\mathrm{k-1} \times \triangle\mathbf{q} \tag{8}$$

where $\triangle\mathbf{q} = (\cos(\|\omega_\mathrm{k}\|, \sin(\|\omega_\mathrm{k}\|)\frac{\omega_\mathrm{k}}{\|\omega_\mathrm{k}\|}))$ [26].

4. Computational Implementation

We used two configurations of hardware for our simulations, which are listed in Table 1. The consumer grade workstation is a typical PC used by a scientist/engineer. The computing grade workstation has a TESLA computing graphics card which is dedicated for numerical computations.

Table 1: Hardware Specifications.

	Consumer Grade		Computing Grade
CPU	Intel i7 - 2.40 GHz (8 cores)	**CPU**	Intel i7 - 3.50 GHz (12 cores)
RAM	16 GB DDR3 - 1600 Mhz	**RAM**	32 GB DDR3 - 1600 MhZ
GPU	GTX 780M - 0.80GhZ (8 SM's)	**GPU**	TESLA K20 - 0.71GhZ (13 SM's)
VRAM	4GB GDDR5 - 2500 MhZ	**VRAM**	5GB GDDR5 - 2500 MhZ
HDD	120 GB SSD	**HDD**	120 GB SSD

4.1. BLAZE-DEM framework.

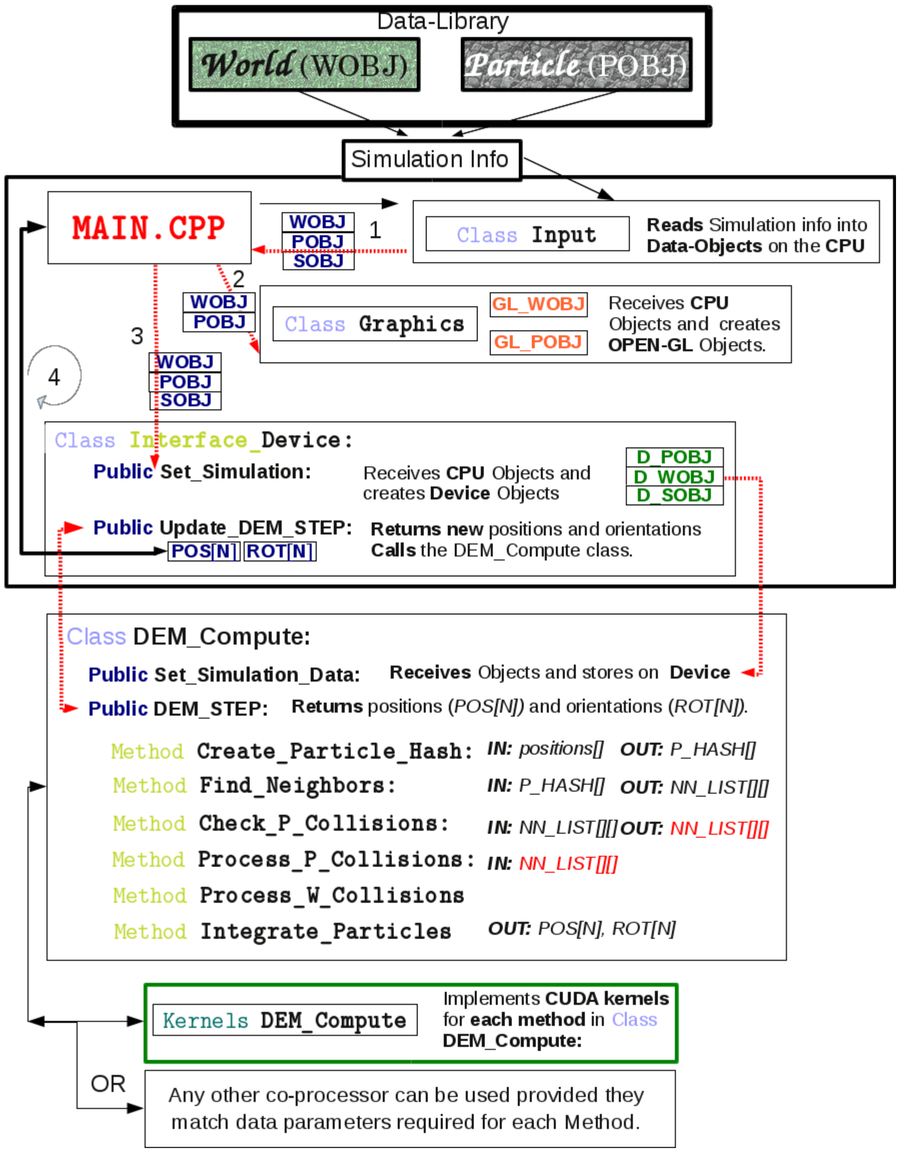
In designing the framework for BLAZE-DEM we took the following features into consideration:

1. A modular environment that can be easily extended to simulate additional physics, such as fluid interactions.
2. A light-weight transparent class design than can be integrated easily into gaming and simulation environments.
3. A 3D graphics environment that is interactive and can display millions of polyhedra.
4. An interface between the numerical task computing and the DEM algorithm.
5. Portability to new architectures.

Figure 12 describes the BLAZE-DEM framework. The Data-Library is analogous to what is found in a typical game and allows complex simulation environments to be created using a combination of world and particles objects. The simulation information is stored in a text file:

1. Names of the world and particle objects.
2. Total number of particles for each particle object type.
3. Spatial location and orientation of particles.

4. Initial conditions and the values of the physical parameters.
5. Specification of the force models to be used for particle interaction.
6. Required statistics e.g. system energy and number of collisions.



Data-Library
World (WOBJ)
Particle (POBJ)
Simulation Info
MAIN.CPP
WOBJ
POBJ
SOBJ
1
2
3
4
Class Input
Reads Simulation info into Data-Objects on the CPU
Class Graphics
GL_WOBJ
GL_POBJ
Receives CPU Objects and creates OPEN-GL Objects.
Class Interface_Device:
Public Set_Simulation:
Receives CPU Objects and creates Device Objects
D_POBJ
D_WOBJ
D_SOBJ
Public Update_DEM_STEP:
POS[N]
ROT[N]
Returns new positions and orientations
Calls the DEM_Compute class.
Class DEM_Compute:
Public Set_Simulation_Data:
Receives Objects and stores on Device
Public DEM_STEP:
Returns positions (POS[N]) and orientations (ROT[N]).
Method Create_Particle_Hash: IN: positions[] OUT: P_HASH[]
Method Find_Neighbors: IN: P_HASH[] OUT: NN_LIST[][]
Method Check_P_Collisions: IN: NN_LIST[][] OUT: NN_LIST[][]
Method Process_P_Collisions: IN: NN_LIST[][]
Method Process_W_Collisions
Method Integrate_Particles OUT: POS[N], ROT[N]
Kernels DEM_Compute
Implements CUDA kernels for each method in Class DEM_Compute:
OR
Any other co-processor can be used provided they match data parameters required for each Method.

Figure 12: BLAZE-DEM Framework.

The code is object-orientated C++ using the OpenGL graphics library for visualization and QT for the graphical user interface. The *Input* class reads the simulation input and creates world, particle and simulation objects which are passed to the function *Main*. *Main* passes the world and particle objects to the graphics class which creates the required OpenGL graphics objects. After creation of the graphics objects, *Main* then passes all the objects to the *Set_ Simulation* method in the *Interface* Class, which creates objects for the computing device which is currently only the GPU. A Kernel on the GPU stores the objects from the interface into GPU memory. Once the initial data objects are passed to the required classes, *Main* then makes a request to the *Interface* to advance the simulation and return new positions and orientations. The *Interface* then calls the *DEM_ Compute* class, which calls each method that in-turn invokes a kernel which performs the required operations. The GPU implementation is separate from the CPU implementation with only communication between the interface. This allows us to easily do computations on different devices and implement new physics models, without having to change the entire code.

4.2. Data Representation

Algorithm 1 describes our GPU implementation. The argument in angle brackets launches **N** parallel threads on the GPU (no need for a loop as in serial calculations). We use thread level parallelism, mapping a single particle to a single thread. At the start of the simulation we copy the initial data described in Figure 11 into GPU memory. No further memory transactions between the CPU and GPU, as we create a handle between OpenGL, and the particle position and orientation GPU memory spaces. Each line (1-6) of Algorithm 1 is a CUDA kernel optimized for the Kepler architecture (see line comments for description).

Algorithm 1 Discrete Element GPU Implementation.

```
COPY (HOST-DEVICE): dynamics data arrays →global memory and particle data arrays →constant memory on the GPU.
1. CalculateParticleHash (position_com) <<N>> /* Calculate hash given by Equation (1) */
2. SortParticleDynamics (p_hash[]) <<N>> /* Sort the dynamics data arrays based on hash to improve memory access as discussed in section 2.1 */
3. Find_NN_Phase1 (position_com) <<N>> /* Create NN_LIST[N][x] containing NumNN[N] potentially contacting particles */
4. Contact_Detection (position_com) <<N>> /* Detailed contact detection as described in Section 3 */
       for i=0 to NumNN [thread.id] do
         if particle i and thread.id are in contact.
              calculate force $\bar{\mathbf{F}}_{ij}$.
         end if
         $\bar{\mathbf{F}}_i += \mathbf{F}_{ij}$
       end for
       update $\vec{\mathbf{v}}_{\mathrm{i}}$,$\vec{\mathbf{a}}_{\mathrm{i}}$
5. Integrate_Position (position_com, velocity_com, accelrat_com) <<N>> /* Numerical integration described in Section 4.2 */
6.Check_WorldCollision (position_com, velocity_com) <<N>> /* Ray-Trace between particle vertexes and world surfaces and resolve collisions */
```

5. Simulation examples with BLAZE-DEM

5.1. Numerical Verification of code

The numerical verification of a DEM code that simulates non-spherical 3D particle behavior is a complex matter and debugging is a non-trivial task as there are no standard tests that can be used to verify a model other than comparison with experimental or other data [5]. Furthermore the debugging of GPU code is difficult and caution must be taken as C++ Object Orientation is not fully implemented on the GPU. To verify that the numerical integration on the GPU yields the expected results, we simulated the motion of a single particle falling under the effects of gravity and air-resistance ($\mathrm{F} = \mathrm{mg} - \alpha v^2$), which is a 2nd order non-linear system and should be matched well by our numerical simulation, as we are using a 2nd order integration scheme. The analytical expressions for the velocity and position of the particle are $v(t) = \sqrt{\frac{mg}{\alpha}} \tanh\left(\frac{t}{\sqrt{\frac{m}{g\alpha}}}\right)$ and $x(t) = \frac{m}{\alpha} \ln\left[\cosh\left(\frac{t}{\sqrt{\frac{m}{g\alpha}}}\right)\right]$. Figure 13 shows the results with $\alpha = 0.5$. We see good agreement for both time-steps with the numerical error bound at 0.10% for a step size of 10^{-3} and 0.010% for a step size of 10^{-4}.

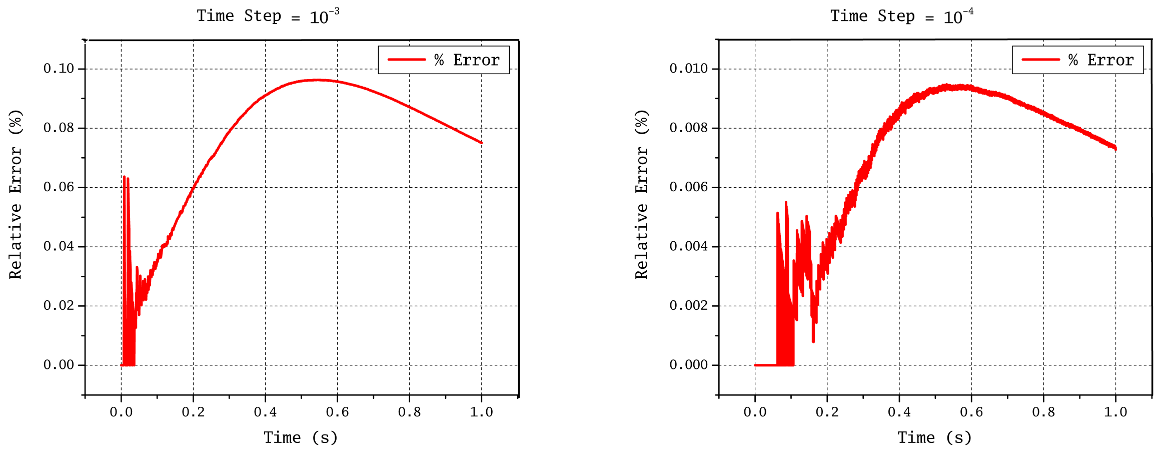


Figure 13: Verification of numerical integration scheme for different time steps.

5.2. Gravity Packing

To verify that the simulated bulk behavior of the system is satisfactory, we generated a grid of (128×128×64) polyhedra (0.001cm spacing) with the grid starting at a height of 1cm above the bottom of a (200×200×128) cm container. The particles have no initial velocity and fall under the influence of gravity into the container. Figure 14 shows the system at the start and after 1 second. This simulation tests the robustness of our contact detection algorithm and numerical stability as particles collide with each other and the world until they reach a final configuration at rest (quasi-static conditions). Simulating the gravity packing of particles have been used by numerous researchers [27] to test the robustness of algorithms in terms of numerical stability. Figure 15 shows a plot of the kinetic energy of the system over the duration of the simulation. We notice that the simulation does converge numerically as the energy decreases due to friction and damping in the system, which results from collisions between particles and particles and the container (world).

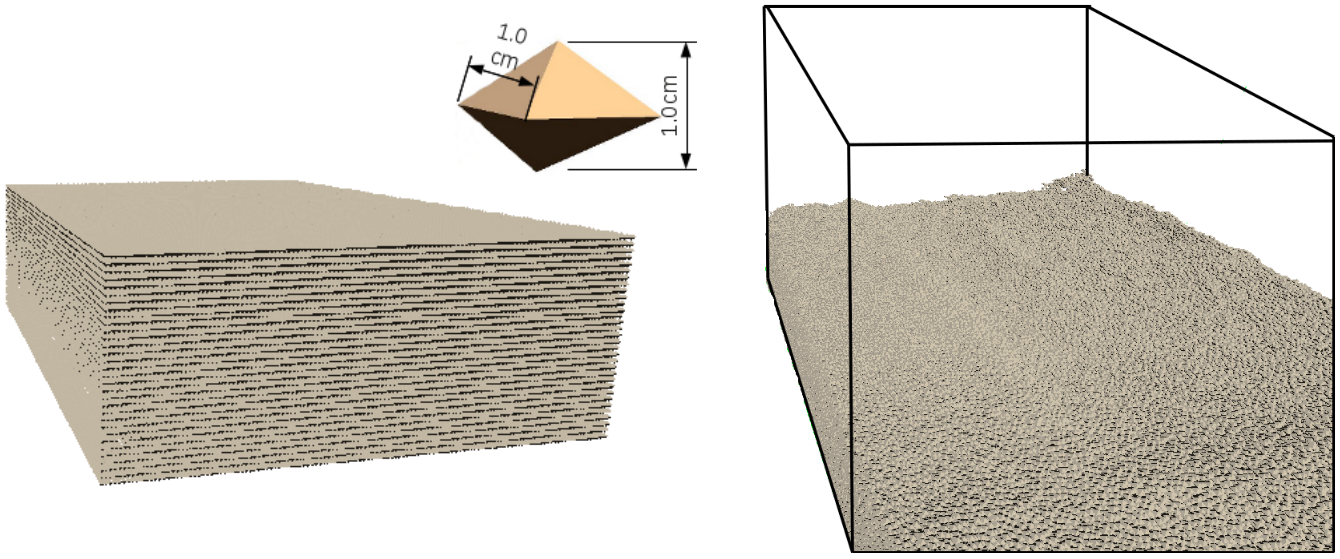


Figure 14: Gravity packing for 1 million polyhedra.

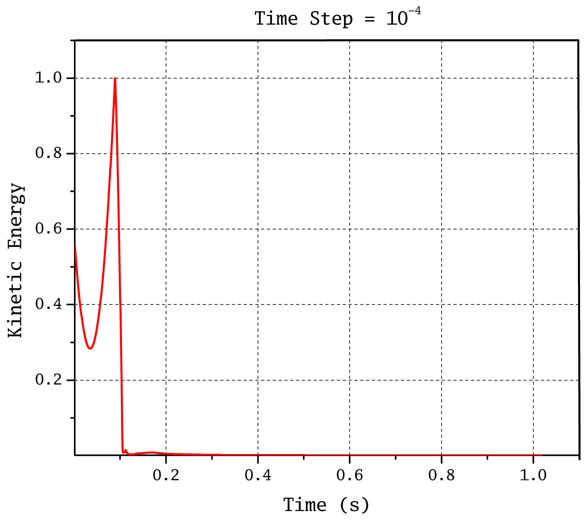


Figure 15: Numerical stability of system for gravity packing problem.

Figure 16 shows the scaling performance of the code as we increase the number of particles (8 face polyhedra) for the gravity packing simulation shown in Figure 15. We see that the computational time scales by a factor N up to a million particles. The TESLA card with more SMs than the GTX 780 is only 30% faster and costs 4 times as much.

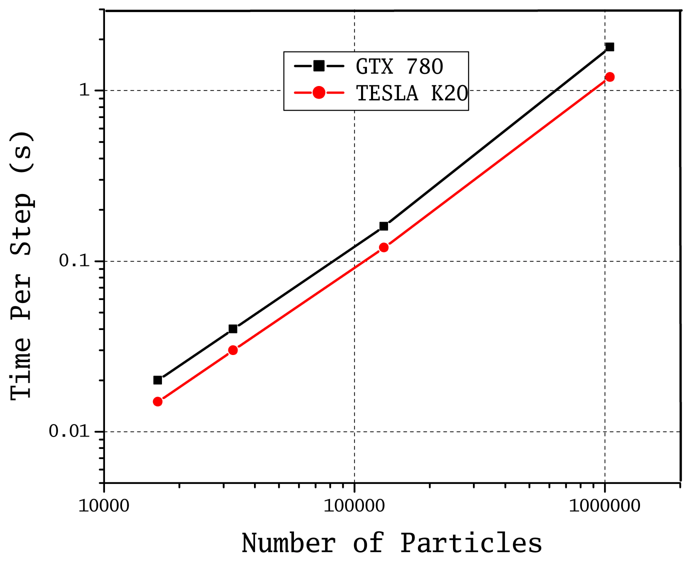


Figure 16: Performance scaling with number of polyhedral particles.

5.3. Hopper flow

To validate our code we devised an experiment that captures the motion of glass marbles in a square hopper constructed from Plexiglas, as depicted in Figure 17 (a). We packed a total of 836 marbles into 6 layers consisting of 148 marbles in a ($11\times4\times3$) arrangement with the top layer containing 176 marbles, as depicted in Figure 17 (b). Alternating layers have different colors so that we can do a visual comparison between experiment and simulation at various instants in time to verify the correctness of our code in modeling GM dynamics.

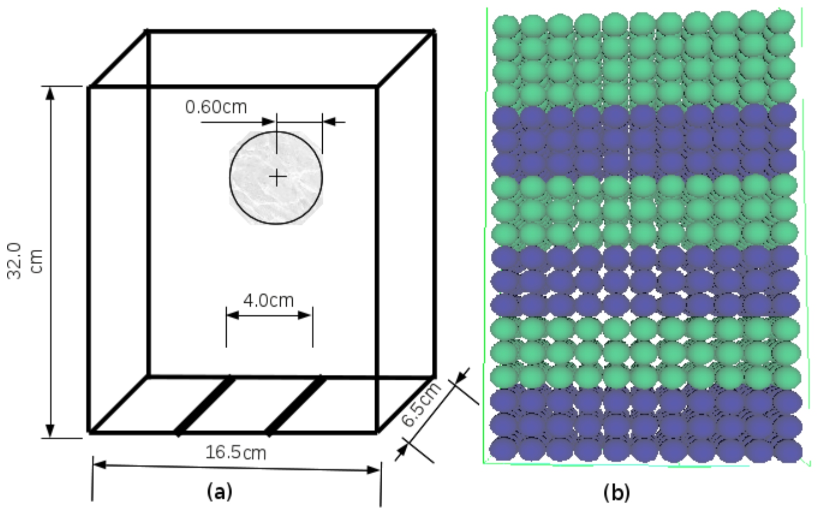


Figure 17: Simulation details.

The initial packing of the marbles in the experiment is stochastic in nature and thus impossible to exactly reproduce in simulation. The initial packing between experiment and simulation (t=0.0) is representative and allows us to make qualitative comparisons between flow patterns. The flow patterns between experiment and simulation for spherical particles are shown in Figure 18. We see a very good agreement between experiment and simulation for the bulk flow behavior.

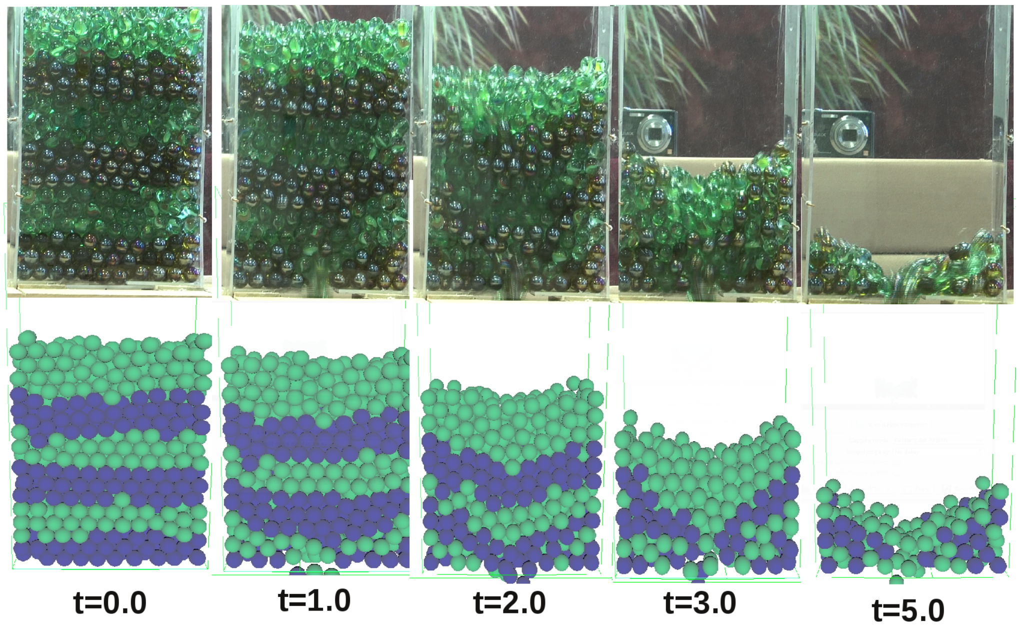


Figure 18: Comparison of experimental and DEM results for hopper flow of 836 spheres.

We were not to able to perform experimental validation for polyhedral shaped particles at the time of writing this paper. However we can still do a visual check to see if we are correctly simulating the bulk behavior of a system as the flow pattern will be similar to that of spheres. Figure 19 shows corn shaped polyhedral particles flowing in a hopper (same as Figure 17(a), with the bottom inclined by 5 degrees to enable smooth flow as polyhedra tend to arch as illustrated in Figure 20). They are initially packed in a $(32 \times 27 \times 16)$ grid. We see that the polyhedra form a denser packing in the corners due to interlocking, as a result of their shape and also flow much slower than spheres.

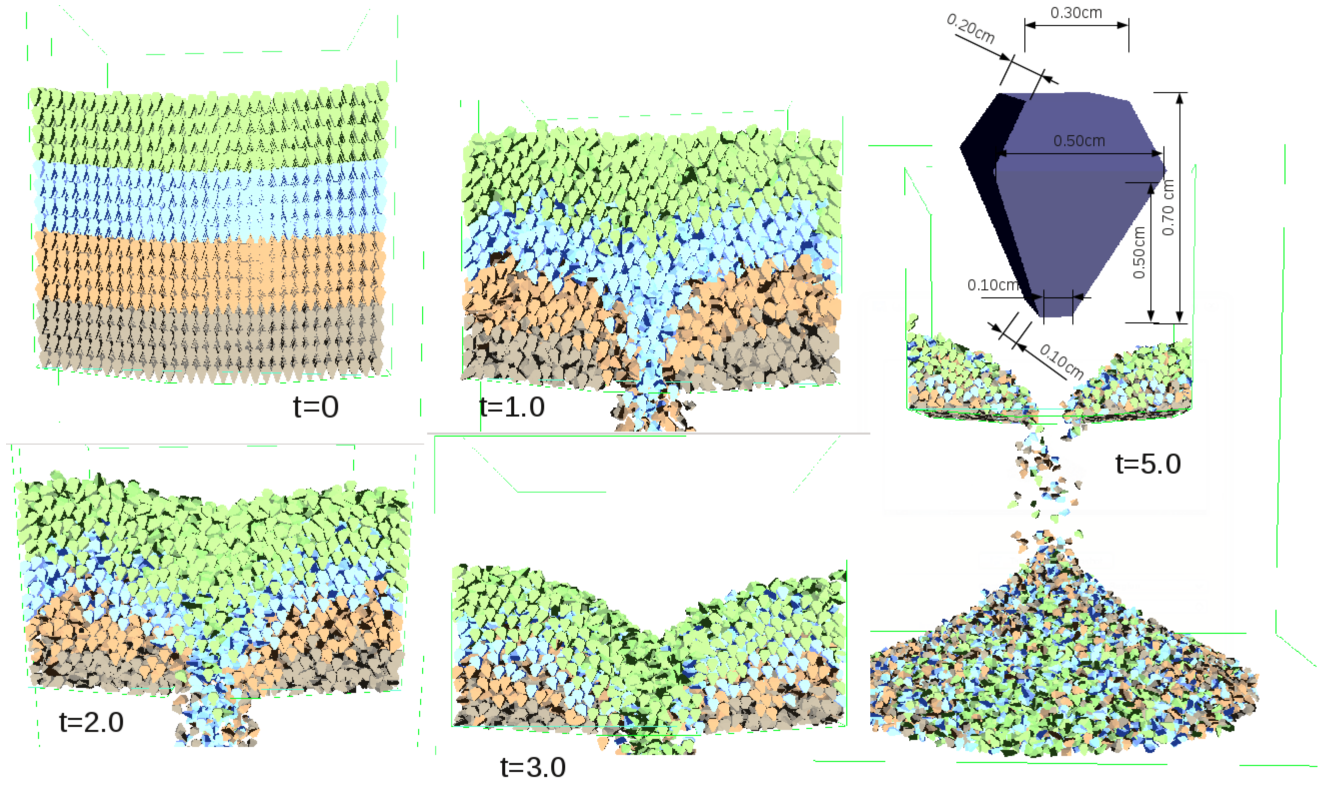


Figure 19: Hopper flow with 13824 corn shaped polyhedral particles.

The effect of particle shape indeed has an effect on particle dynamics as polyhedra have the ability to restrict flow significantly compared to a spherical representation of particles as illustrated in Figure 20. This finding is consistent with other authors [9, 8, 16]. We achieved a frame rate of 167 FPS (0.006 s per step) with a step size of 10^{-4} the simulation of 5 seconds required 5 minutes of computational time.

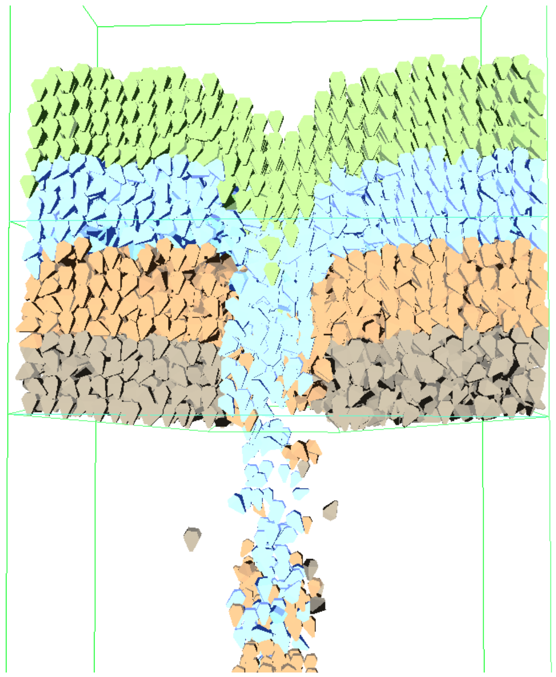


Figure 20: Polyhedra arching to restrict flow.

6. Conclusions and Future Work

This paper develops a computational framework and algorithms for discrete element modeling of convex polyhedral and spherical particles using a simple physics model on the GPU. These developments are proposed to increase computational efficiency for large-scale granular material simulation where a simplified physics model is sufficient. Several simulations are presented to demonstrate the numerical stability and performance of BLAZE-DEM. The comparison between experiment and simulation for the marble problem is promising and future plans include the exploitation of temporal coherence and further optimization of GPU code to get better performance, as well as an experimental comparison for polyhedral particles. We conclude that the GPU is well suited to DEM simulations where a simple physics model can be used to expose parallelism in DEM, albeit to limited applications

References

[1] P. Cleary, The filling of dragline buckets, Math. Eng. Ind. 29 (1998) 1–24.

[2] B. Mishra, R. Rajamani, Simulation of charge motion in ball mills. part 1: experimental verifications, Int. J. Mineral Process 40 (1994) 171–186.

[3] W. Ketterhagen, J. Curtis, C. Wassgren, Predicting the flow mode from hoppers using the discrete element method, Powder Technology 195 (2009) 1–10.

[4] M. Moakher, T. Shinbrot, F. Muzzio, Experimentally validated computations of flow, mixing and segregation of non-cohesive grains in 3d tumbling blenders, Powder Technology 109 (2000) 58–71.

[5] P. Langston, Distinct element modelling of non-spherical frictionless particle flow, Chemical Engineering Science 59 (2004) 425–435.

[6] P. Cundall, Strack, A discrete numerical model for granular assemblies, Geotechnique 29 (1979) 47–65.

[7] C. Radeke, B. Glasser, J. Khinast, Large-scale powder mixer simulations using massively parallel gpu architectures, Chemical Engineering Science 65 (2010) 6435–6442.

[8] J. Latham, A. Munjiza, The modelling of particle systems with real shapes, Philosophical Transactions of The Royal Society of London Series A: Mathematical Physical and Engineering Sciences 362 (2004) 1953–1972.

[9] P. Cleary, M. Sawley, Dem modelling of industrial granular flows: 3d case studies and the effect of particle shape on hopper discharge, Applied Mathematical Modelling 26 (2002) 89–111.

[10] H. Abou-Chakra, J. Baxter, U. Tuzun, Three-dimensional particle shape descriptors for computer simulation of non-spherical particulate assemblies, Advanced Powder Technology 15 (2004) 63–77.

[11] D. Hohner, S. Wirtz, V. Emden, H.K. Scherer, Comparison of the multi-sphere and polyhedral approach to simulate non-spherical particles within the discrete element method: Influence on temporal force evolution for multiple contacts, Powder Technology 208 (2011) 643–656.

[12] A. Anderson, et.al, General purpose molecular dynamics simulations fully implemented on graphics processing units, Journal of Computational Physics 47 (2008) 1–17.

[13] J. Longmore, et.al, Towards realistic and interactive sand simulation: A gpu-based framework, Powder Technology 235 (2013) 983–1000.

[14] N. Bell, Y. Yu, Particle-based simulation of granular materials, Eurographics/ACM SIGGRAPH Symposium on Computer Animation 25 (2005) 29–31.

[15] D. Markauska, Investigation of adequacy of multi-sphere approximation of elliptical particles for dem simulations, Granular Matter 12 (2010) 107–123.

[16] S. Mack, P. Langston, C. Webb, York.T., Experimental validation of polyhedral discrete element model, Powder Technology 214 (2011) 431–442.

[17] B. Nassauer, T. Liedke, Polyhedral particles for the discrete element method, Granular Matter 15 (2013) 85–93.

[18] C. Boon, G. Houlsby, S. Utili, A new algorithm for contact detection between convex polygonal and polyhedral particles in the discrete element method, Computers and Geotechnics 44 (2012) 73–82.

[19] D. Zhao, E. Nezami, Y. Hashash, J. Ghaboussi.J., Three-dimensional discrete element simulation for granular materials, Computer-Aided Engineering Computations: International Journal for Engineering and Software 23 (2006) 749–770.

[20] NVIDIA, Cuda 6 (May 2014).
URL `http://www.nvidia.com/cuda`

[21] NVIDIA, Nvida kepler gk110 architecture whitepaper (2012).
URL `http://www.nvidia.com/NVIDA KEPLER GK110 Architecture Whitepaper`

[22] J. Sanders, E. Kandrot, CUDA by example, Vol. 12, 2010.

[23] B. Grunbaum, Convex Polytopes, 2nd edition, Volker Kaibel, ISBN 978-0-387-40409-7, 2003.

[24] P. Cundall, Formulation of a three-dimensional distinct element model - part i: a scheme to detect and represent contacts in a system composed of many polyhedral blocks, Int. J. of Rock Mech 25 (1988) 107–116.

[25] E. Battey-Pratt, T. Racey, Geometric model for fundamental particles, International Journal of Theoretical Physics 19 (1980) 6.

[26] T. Harada, GPU Gems 3: Real-time rigid body simulation on GPUs, Vol. 3, 2008.

[27] Sanni.I, A reliable algorithm to solve 3d frictional multi-contact problems: Application to granular media, Journal of Computational and Applied Mathematics 234 (2010) 1161–1171.