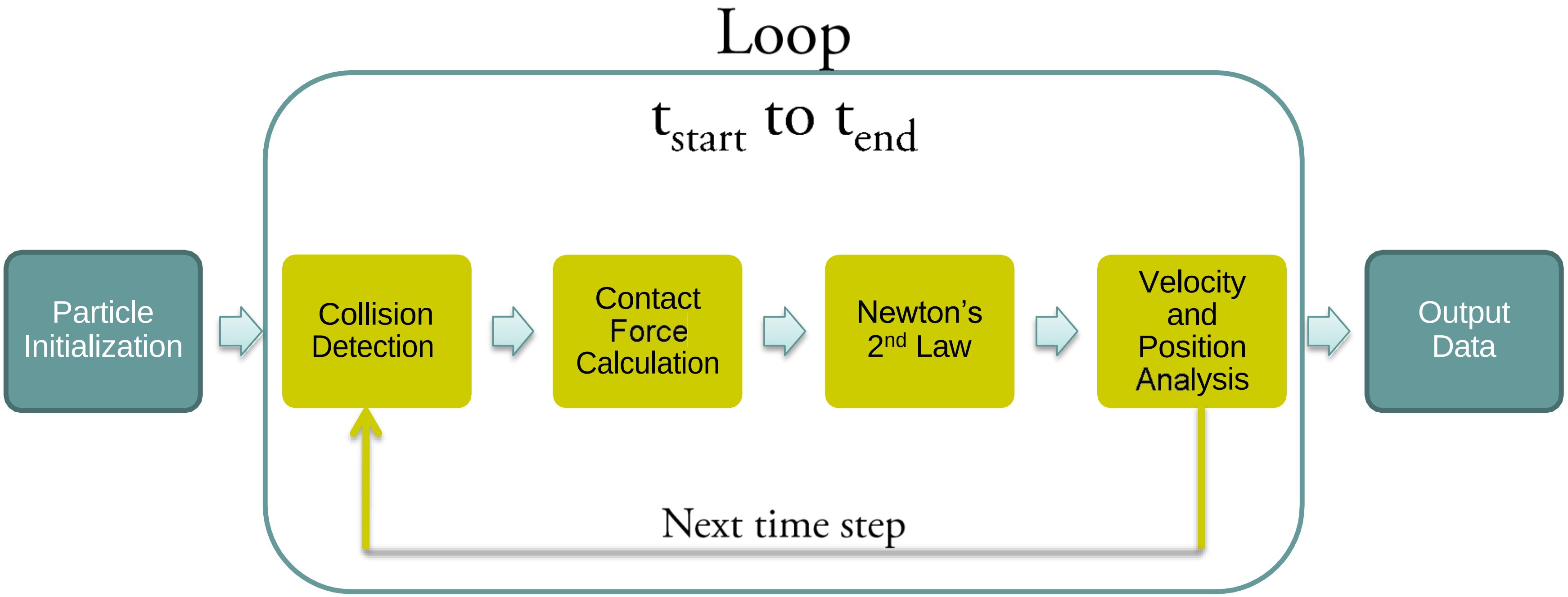
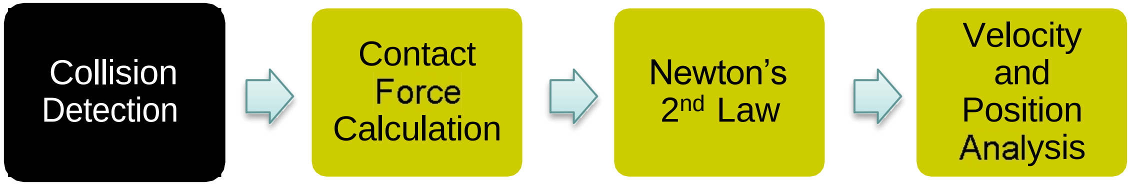
Discrete Element Method

- Collision detection determines pairs of colliding bodies
- Contact forces computed based on constitutive relation (spring-damper model)
- Requires small time-steps
- Newton’s Second Law used to compute accelerations
- Numerical integration (e.g., Velocity Verlet) used to compute velocity, position of all bodies

Discrete Element Method



DEM



Collision Detection → Contact Force Calculation → Newton's 2^{nd} Law → Velocity and Position Analysis

- Spatial Subdivision
- 2 particles: $\mathbf{r_i}, \mathbf{r_j}$

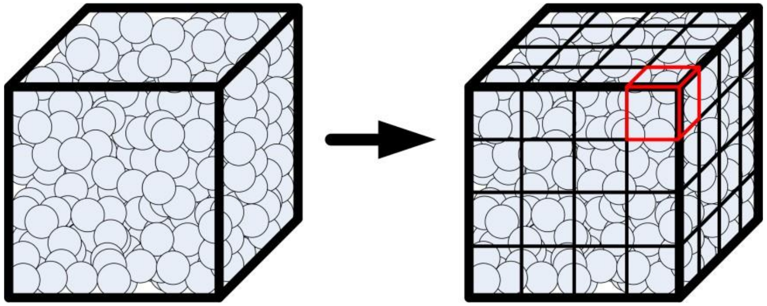
$$\mathbf{r_{ij}} = \mathbf{r_i} - \mathbf{r_j}$$

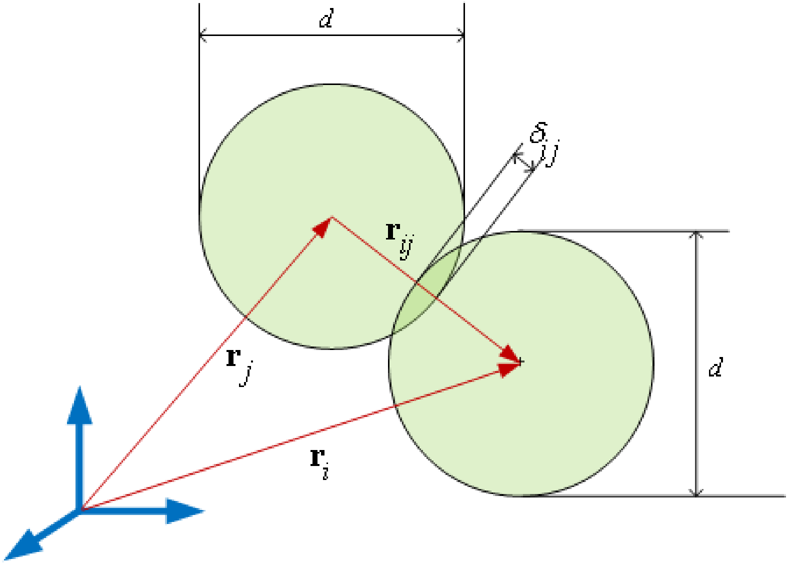
- If $r_{ij} \leq d$

$$\delta_{ij} = d - r_{ij}$$

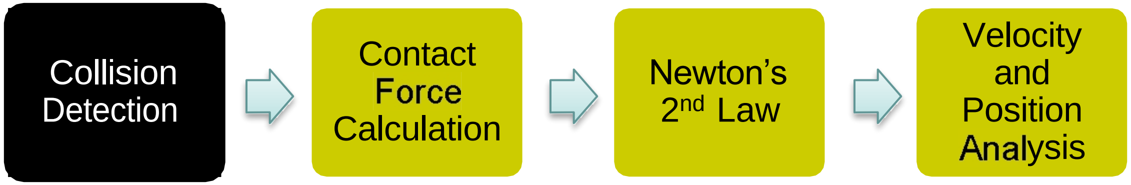
$$\mathbf{n_{ij}} = \frac{\mathbf{r_{ij}}}{r_{ij}}$$

- Otherwise no collision





DEM



Collision Detection

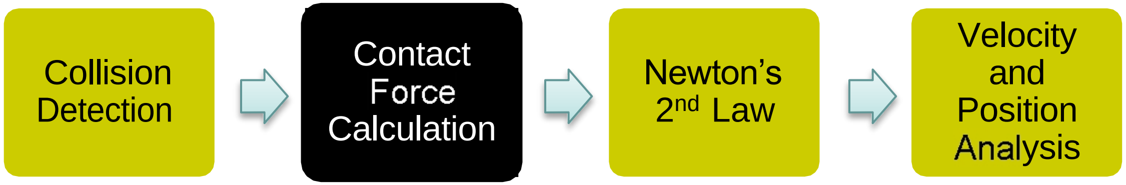
Contact Force Calculation

Newton's 2^{nd} Law

Velocity and Position Analysis

- Collision detection code will be provided to you
- Input: Arrays of sphere positions and radii
- Output: Array of collision data

DEM



Collision Detection

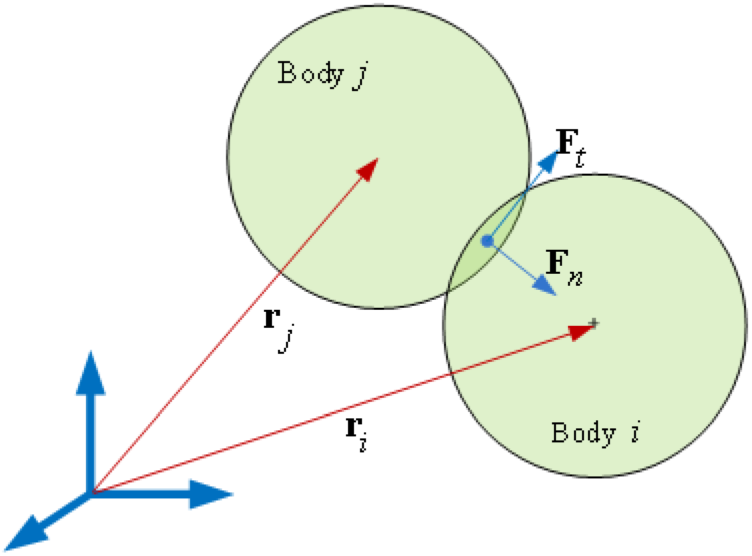
Contact Force Calculation

Newton's 2^{nd} Law

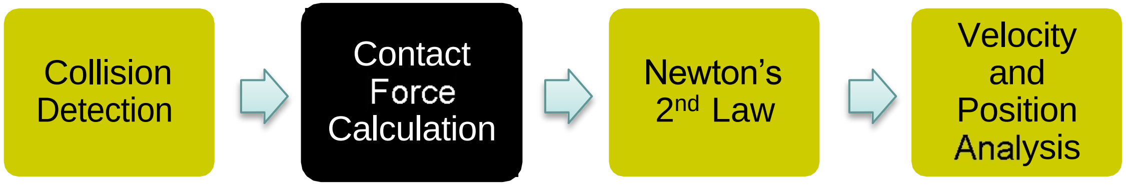
Velocity and Position Analysis

- Contact force components
  - **normal**
  - tangential

- Four different categories:
  - Continuous potential models
  - **Linear viscoelastic models**
  - Non-linear viscoelastic models
  - Hysteretic models



DEM



Collision Detection → Contact Force Calculation → Newton's 2nd Law → Velocity and Position Analysis

$$\mathbf{v_{ij}} = \mathbf{v_i} - \mathbf{v_j}$$

$$m_{eff} = \frac{m_i m_j}{(m_i + m_j)}$$

$$\mathbf{v_{n_{ij}}} = (\mathbf{v_{ij}} \cdot \mathbf{n_{ij}})\mathbf{n_{ij}}$$

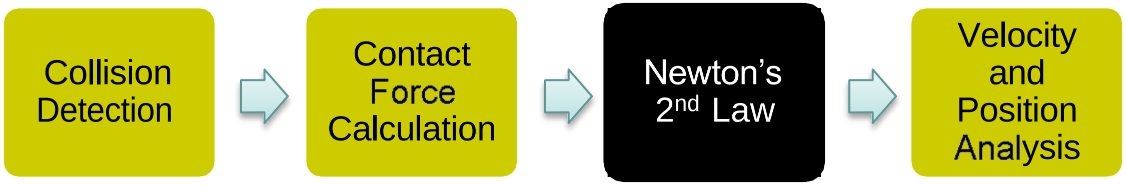
- Normal Force $\mathbf{F_{n_{ij}}}$ computed as:

$$\mathbf{F_{n_{ij}}} = f\left(\frac{\delta_{ij}}{d}\right)(k_n \delta_{ij} \mathbf{n_{ij}} - \gamma_n m_{eff} \mathbf{v_{n_{ij}}})$$

k_n – *spring stiffness*

γ_n – *damping coefficient*

DEM



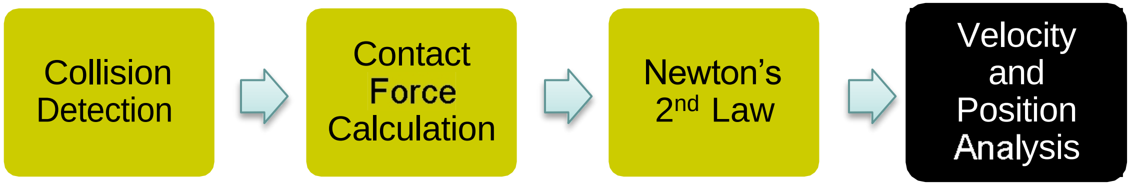
- Force on one particle is the sum of its contact forces and gravity:

$$\mathbf{F_i^{tot}} = m_i\mathbf{g} + \sum_j \mathbf{F_{n_{ij}}}$$

- Calculation acceleration:

$$\mathbf{F_i^{tot}} = m_i\mathbf{a_i} \rightarrow \mathbf{a_i} = \frac{\mathbf{F_i^{tot}}}{m_i}$$

DEM



Collision Detection

Contact Force Calculation

Newton's 2nd Law

Velocity and Position Analysis

- Use explicit numerical integration methods like Explicit Euler or Velocity Verlet Integration

- Explicit Euler:

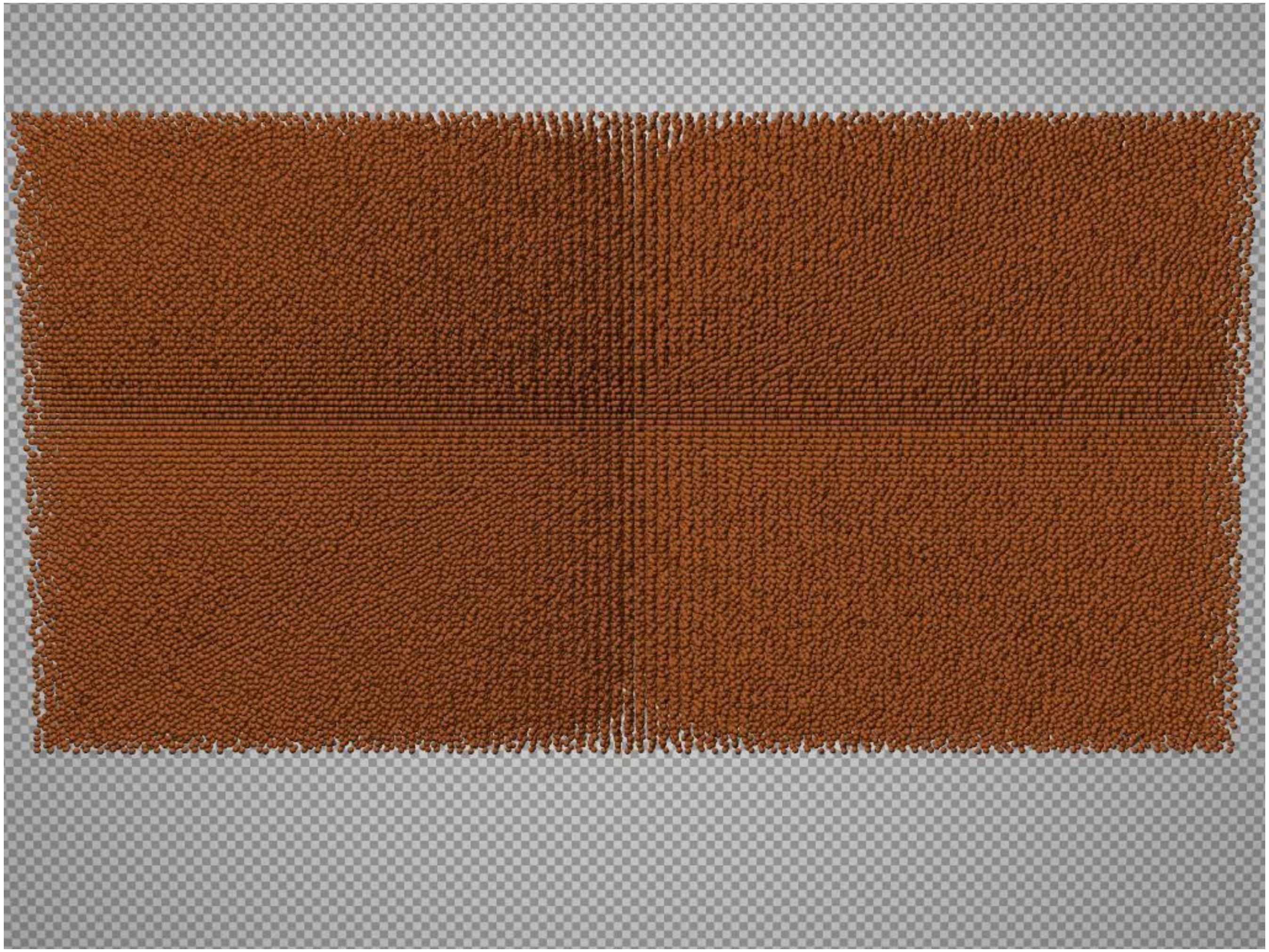
$$\mathbf{r}_\mathbf{i}(t+\Delta t) = \mathbf{r}_\mathbf{i}(t) + \mathbf{v}_\mathbf{i}(t)\Delta t$$

$$\mathbf{v}_\mathbf{i}(t+\Delta t) = \mathbf{v}_\mathbf{i}(t) + \mathbf{a}_\mathbf{i}(t)\Delta t$$

Parallelism

- Parallel collision detection (provided)
- (Per-contact): Compute collision forces
- (Per-body): Reduction to resultant force per body
- (Per-body): Solution of Newton’s Second Law, time integration

Example



- 1 million spheres
- 0.5 sec long simulation
- ~12,000 sec computational time
- GPU

Suggested Code Structure

- Class ParticleSystem
  - void initializeSim()
  - void performCD()
  - void computeForces()
  - void integrate()
  - void getGPUdata()
  - void outputState()

void initializeSim()

- Set initial conditions of all bodies
- Copy state data from host to device

void performCD()

- Call GPU CD function (provided) to determine pairs of colliding spheres
- Returns array of contact_data structs
  - data members: objectIdA, objectIdB

void computeForces()

- Compute contact force for each contact
- Compute resultant force acting on each body
- Compute and add reaction force for contact with boundary planes

void integrate()

- Compute acceleration of each body
- Update velocity and position of each body

void getGPUdata()

- Copy state data back to host

void outputState()

- Output sphere positions and radii to a text file

main function

```
int main(int argc, char* argv[])
{
    float  t_curr=0.0f;
    float  t_end=1.0f;
    float h=0.00005f;
    ParticleSystem *psystem = new ParticleSystem(…);
    psystem->initializeSim();
    while(t_curr<=t_end)
    {
        psystem->performCD();
        psystem->computeForces();
        psystem->integrate();
        t_curr+=h;
    }
    delete psystem;
    return 0;
}
```

Other Tips (Force computation)

1. Compute force for each contact with one thread per contact
   - Store key-value array with body ID as key, force as value
   - Note each contact should create a force on two bodies
2. Sort by key (body ID)
   - thrust::sort_by_key(…)

Other Tips (Force computation)

3. Sum all forces acting on a single body
   - thrust::reduce_by_key(…)
   - One thread per entry in output, copy to appropriate place in net force list
4. Add gravity force to each body’s net force
   - One thread per body

Other Tips (Force computation)

5. Contact with planes
   - Assume infinite planes
   - A plane is defined by a point (**p**) and normal direction (**N**)
   - One thread per sphere (at position **r**)
     - Compute $d = \mathbf{N} \cdot (\mathbf{r} - \mathbf{p})$
     - Contact if *d*<*radius*
     - Compute force as before, add to net force

Parallel Collision Detection

Overview

- Method 1: Brute Force
  - Easier implementation
  - $O(N^2)$ Complexity
- Method 2: Parallel Binning
  - More involved
  - O(N) Complexity

Brute Force Approach

- Three Steps:
  - Run preliminary pass to understand the memory requirements by figuring out the number of contacts present
  - Allocate on the device the required amount of memory to store the desired collision information
  - Run actual collision detection and populate the data structure with the information desired

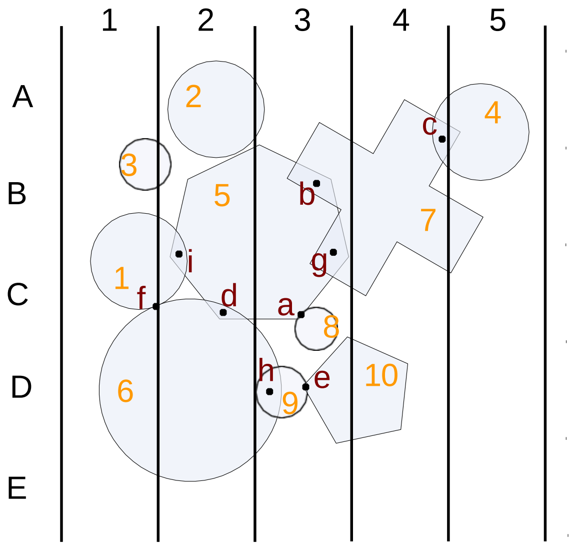
Step 1: Search for contacts

- Create on the device an array of unsigned integers, equal in size to the number *N* of bodies in the system
  - Call this array *dB*, initialize all its entries to zero
  - Array *dB* to store in entry *j* the number of contacts that body *j* will have with bodies of higher index
    - If body 5 collides with body 9, no need to say that body 9 collides with body 5 as well

```
Do in parallel, one thread per body basis
       for body j, loop from k=j+1 to N
              if bodies j and k collide, dB[j] += 1
       endloop
endDo
```

Step 1, cont.

Body Index	1	2	3	4	5	6	...
dB	2	0	0	1	4	1	...

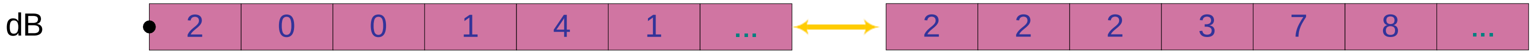


Step 2: Parallel Scan Operation

- Allocate memory space for the collision information
  - Step 2.1: Define first a structure that might help (this is not the most efficient approach, but we'll go along with it…)

```
struct collisionInfo
{ float3 rA;
float3 rB;
float3 normal;
unsigned int indxA;
unsigned int indxB;
}
```

  - Step 2.2: Run a parallel inclusive prefix scan on *dB*, which gets overwritten during the process



dB

2	0	0	1	4	1	…

2	2	2	3	7	8	…

  - Step 2.3: Based on the last entry in the *dB* array, which holds the total number of contacts, allocate from the host on the device the amount of memory required to store the desired collision information. To this end you'll have to use the size of the "struct" *collisionInfo*. Call this array *dCollisionInfo*.

Step 3

- Parallel pass on a per body basis (one thread per body – similar to step 1)
  - Thread *j* (associated with body *j*), computes its number of contacts as *dB[j]-dB[j-1]*, and sets the variable *contactsProcessed=0*
  - Thread *j* runs a loop for *k=j+1* to *N*
  - If body *j* and *k* are in contact, populate entry *dCollisionInfo[dB[j-1]+contactsProcessed]* with this contact's info and increment *contactsProcesed*++
    - Note: you can break out of the look after k as soon as *contactsProcesed== dB[j]-dB[j-1]*

Concluding Remarks, Brute Force

- Level of effort for discussed approach
  - Step 1, O(N^2) (checking body against the rest of the bodies)
  - Step 2: prefix scan is O(N)
  - Step 3, O(N^2) (checking body against the rest of the bodies, basically a repetition of Step 1)
- No use of the atomicAdd, which is a big performance bottleneck
- Numerous versions of this can be contrived to improve the overall performance
  - Not discussed here for this brute force idea, rather moving on to a different approach altogether, called “binning”

Parallel Binning