Multi-Body Dynamics Simulation on the GPU

Dan Negrut  
Simulation-Based Engineering Lab  
University of Wisconsin, Madison

Alessandro Tasora  
Department of Mechanical Engineering  
University of Parma, Italy

JPL  
September 3, 2009
Acknowledgement

- **People:**
  - Mihai Anitescu – Argonne National Lab, Chicago, IL
  - Toby Heyn, Hammad Mazhar, Justin Madsen – University of Wisconsin

- **Partial financial support**
  - NSF, CAREER award
  - Argonne National Laboratory
  - NVIDIA
  - BAE Systems

- **Collaborators**
  - TARDEC (Tank Automotive Research & Development Engineering Center)
Talk Overview

- Quick intro of the dynamics problem of interest
- The methods used and their GPU implementation
  - Parallel collision detection
  - Parallel CCP
- Numerical Experiments
- Validation efforts
- Conclusions
What’s pushing the need for high performance computing in multi-body dynamics simulation?
Rover on Granular Terrain

- We’d like to use sand/gravel type terrain
- Of interest: wheeled/tracked vehicle mobility on granular terrain
Frictional Contact Simulation: ADAMS

ADAMS Specific Setting:
- Spheres: 60 mm diameter and mass 0.882 kg
- Forces: smoothing with stiffness of 1E5, force exponent of 2.2, damping coefficient of 10.0, and a penetration depth of 0.1
- Simulation length: 3 seconds

\[
y = 0.8385x^2 - 7.2607x + 16.154 \\
R^2 = 0.9985
\]
The Issue and How It’s Addressed

- Long simulation times in ADAMS traced back to the underlying formulation of the frictional contact problem
  - Draws on a “smoothing” (penalty) approach
    - Sophisticated but slow
    - General purpose tool

- Solution embraced draws on DVI (Differential Variational Inequalities)
  - A set of differential equations combined with inequality constraints
  - Less general than ADAMS’ penalty approach
The DVI Approach

Kinematic differential equations (links generalized positions to generalized velocities)

\[ \dot{\mathbf{q}} = \mathbf{L}(\mathbf{q}) \mathbf{v} \]

Force balance equation (Newton’s Second Law)

\[ \mathbf{M} \dot{\mathbf{v}} = \mathbf{f}(t, \mathbf{q}, \mathbf{v}) + \sum_{i=1}^{c} (\tilde{\gamma}_{i,n} \mathbf{D}_{i,n} + \tilde{\gamma}_{i,u} \mathbf{D}_{i,u} + \tilde{\gamma}_{i,w} \mathbf{D}_{i,w}) \]

For \( 1 \leq i \leq c \) : \( \tilde{\gamma}_{i,n} \geq 0 \perp \Phi_i(\mathbf{q}) \geq 0 \)

Contact complementarity conditions

For \( 1 \leq i \leq c \) : \( (\tilde{\gamma}_{i,u}, \tilde{\gamma}_{i,w}) = \operatorname{argmin}_{\mu_i \tilde{\gamma}_{i,n} \geq \sqrt{(\tilde{\gamma}_{i,u})^2 + (\tilde{\gamma}_{i,w})^2}} \mathbf{v}^T (\tilde{\gamma}_{i,u} \mathbf{D}_{i,u} + \tilde{\gamma}_{i,w} \mathbf{D}_{i,w}) \).

Optimality condition (regards energy dissipated in unit time by friction): captures Coulomb model
Traditional Discretization Scheme

\[ \mathbf{q}^{(l+1)} = \mathbf{q}^{(l)} + h\mathbf{L}(\mathbf{q}^{(l)})\mathbf{v}^{(l+1)} \]

\[ \mathbf{M}(\mathbf{v}^{(l+1)} - \mathbf{v}^{l}) = h\mathbf{f}(t^{(l)}, \mathbf{q}^{(l)}, \mathbf{v}^{(l)}) + \sum_{i \in \mathcal{A}(\mathbf{q}^{(l)}, \delta)} (\gamma_{i,n} \mathbf{D}_{i,n} + \gamma_{i,u} \mathbf{D}_{i,u} + \gamma_{i,w} \mathbf{D}_{i,w}) \]

\[ i \in \mathcal{A}(\mathbf{q}^{(l)}, \delta) : \quad 0 \leq \frac{1}{h} \Phi_i(\mathbf{q}^{(l)}) + \mathbf{D}_{i,n}^T \mathbf{v}^{(l+1)} \perp \gamma_i \geq 0, \]

\[ (\gamma_{i,u}, \gamma_{i,w}) = \arg\min_{\mu_i \gamma_{i,n} \geq \sqrt{\gamma_{i,u}^2 + \gamma_{i,w}^2}} \mathbf{v}^T (\gamma_{i,u} \mathbf{D}_{i,u} + \gamma_{i,w} \mathbf{D}_{i,w}). \]

(Stewart, 1998)
Relaxed Discretization Scheme Used

\[ q^{(l+1)} = q^{(l)} + hL(q^{(l)})v^{(l+1)} \]

\[
M(v^{(l+1)} - v^l) = hf(t^{(l)}, q^{(l)}, v^{(l)}) + \sum_{i \in A(q^{(l)}, \delta)} \left( \gamma_{i,n} D_{i,n} + \gamma_{i,u} D_{i,u} + \gamma_{i,w} D_{i,w} \right)
\]

\[ i \in A(q^{(l)}, \delta) : \quad 0 \leq \frac{1}{h} \Phi_i(q^{(l)}) + D_{i,n}^T v^{(l+1)} - \mu_i \sqrt{(v^T D_{i,u})^2 + v^T D_{i,w})^2} \quad \gamma^i_{n} \geq 0, \]

\[ (\gamma_{i,u}, \gamma_{i,w}) = \operatorname{argmin}_{\mu_i \gamma_{i,n} \geq \sqrt{\gamma_{i,u}^2 + \gamma_{i,w}^2}} v^T (\gamma_{i,u} D_{i,u} + \gamma_{i,w} D_{i,w}) . \]

(Anitescu & Tasora, 2008)
Cone Complementarity Problem (CCP)

- Introduce the convex hypercone...

\[ \gamma = \bigoplus_{i \in A(q^l, \epsilon)} FC^i \]

- And its polar hypercone:

\[ \gamma^\circ = \bigoplus_{i \in A(q^l, \epsilon)} FC^{i\circ} \]

\( FC^i \in \mathbb{R}^3 \) represents the \( i^{th} \) friction cone

CCP assumes following form: Find \( \gamma \) such that

\[-(N\gamma + d) \in \gamma^\circ \quad \perp \quad \gamma \in \gamma\]
Problem Solved At Each Time Step
(going from $t_l$ to $t_{l+1}$)

- It boils down to solving this:

\[
\begin{align*}
\gamma \in \mathcal{Y} & \perp -(N\gamma + d) \in \mathcal{Y}^o \\
v^{(l+1)} &= M^{-1} \left( \bar{k} + D\gamma \right) \\
q^{(l+1)} &= q^{(l)} + hL(q^{(l)})v^{(l+1)} \\
-(N\gamma + d) \in \mathcal{Y}^o
\end{align*}
\]
Implementation of Method

- The method outlined implemented using two loops
  - Outer loop – runs the time stepping
  - Inner loop – CCP Algorithm (solves CCP problem at each time step)
Outer Loop (Time-Stepping)

1. Set $t = 0$, step counter $l = 0$, provide initial values for $q^{(l)}$ and $v^{(l)}$.

2. Perform collision detection between bodies. For each contact $i$, compute $D_{i,n}$, $D_{i,u}$, $D_{i,w}$.

3. For each body, compute forces $f(t^{(l)}, q^{(l)}, v^{(l)})$.

4. Use CCP Algorithm to solve the cone complementarity problem and obtain unknown impulse $\gamma$ and velocity $v^{(l+1)}$.

5. Update positions using $q^{(l+1)} = q^{(l)} + h L(q^{(l)}) v^{(l+1)}$.

6. Increment $t := t + h$, $l := l + 1$, and repeat from step 2 until $t > t_{\text{end}}$.
Inner Loop (CCP Algorithm)

1. For each contact $i$, evaluate $\eta_i = 3/\text{Trace}(D_i^T M^{-1} D_i)$.

2. If some initial guess $\gamma^*$ is available for multipliers, then set $\gamma^0 = \gamma^*$, otherwise $\gamma^0 = 0$.

3. Initialize velocities: $v^0 = \sum_i M^{-1} D_i \gamma^0_i + M^{-1} \tilde{k}$.

4. For each contact $i$, compute changes in multipliers for contact constraints:

   $\gamma_i^{r+1} = \lambda \Pi_{T_i} (\gamma_i^r - \omega \eta_i (D_i^T v^r + b_i)) + (1 - \lambda) \gamma_i^r$;

   $\Delta \gamma_i^{r+1} = \gamma_i^{r+1} - \gamma_i^r$;

   $\Delta v_i = M^{-1} D_i \Delta \gamma_i^{r+1}$.

5. Apply updates to the velocity vector:

   $v^{r+1} = v^r + \sum_i \Delta v_i$

6. $r := r + 1$. Repeat from 4 until convergence or $r > r_{\text{max}}$.
Parallelism, Opportunities

1. Perform parallel collision detection
2. Copy contact, body, and constraint data structures to GPU
3. (Body parallel) Force kernel
4. (Contact parallel) Contact preprocessing kernel
5. (Contact parallel) CCP contact kernel
6. (Constraint parallel) CCP constraint kernel
7. (Reduction-slot parallel) Reduction kernel
8. (Body parallel) Body velocity update kernel
9. (Body parallel) Time integration kernel
Parallel Collision Detection
Scalable Collision Detection (CD)

- 30,000 feet perspective:
  - Carry out spatial partitioning of the volume occupied by the bodies
    - Place bodies in bins (cubes, for instance)
  - Follow up by brute force search for all bodies touching each bin
    - Embarrassingly parallel
Key Components, CD Method

- The method proposed draws on
  - Sorting (Radix Sort)
    - $O(N)$ parallel implementation
  - Exclusive Prefix Scan
    - $O(N)$ parallel implementation
  - Fast binning operation for the simple convex geometries
    - On a rectangular grid it is very easy to figure out where the center of a sphere lands
Collision Detection (CD): Binning

- Example: 2D collision detection, bins are squares

- Body 4 touches bins A4, A5, B4, B5
- Body 7 touches bins A3, A4, A5, B3, B4, B5, C3, C4, C5
- In proposed algorithm, bodies 4 and 7 will be checked for collision by three threads (associated with bin A4, A5, B4)
Parallel Binning: Summary of Stages

- Stage 1: Find number of bins touched by each body, populate $T$ (body parallel)
- Stage 2: Parallel exclusive prefix scan of $T$ (length of $T$: $N$)
- Stage 3: Determine body-to-bin association, populate $B$ (body parallel)
- Stage 4: Parallel sort of $B$ (length of $B$: $M$)
- Stage 5: Find bin starting index, populate $C$ (bin parallel)
- Stage 6: Parallel sort of $C$ for pruning (length of $C$: $N_b$)
- Stage 7: Determine # of collisions in each bin, store in $D$ (bin parallel)
- Stage 8: Parallel prefix scan of $D$ (length of $D$: $N_b$)
- Stage 9: Run collision detection and populate $E$ with required collision info (bin parallel)

<table>
<thead>
<tr>
<th>N – number of bodies</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N_b$ – number of bins</td>
</tr>
<tr>
<td>M – total number of bins touched by the bodies present in the problem</td>
</tr>
</tbody>
</table>
Stage 1 (Body Parallel)

- Purpose: find the number of bins touched by each body
- Store results in the “T”, array of N integers
- Key observation: it’s easy to bin bodies
Stage 2: Parallel Inclusive Scan

- Run a parallel inclusive sum on the array $T$
  - The last element is the total number of bin touches, including the last body

- Complexity of Stage: $O(N)$, based on parallel scan algorithm of Harris (GPU Gem 3 and CUDA SDK)

- Purpose: determine the number of entries $M$ needed to store the indices of all the bins touched by each body in the problem
Stage 3: Determine body-&-bin association

- Stage executed in parallel on a per-body basis

- Allocate an array $B$ of $M$ pairs of integers.
  - The key (first entry of the pair), is the bin index
  - The value (second entry of pair) is the body that touches that bin
Stage 4: Radix Sort

- In parallel, run a radix sort to order the B array according to the keys

- Work load: O(N)
  - Algorithm of Garland and Harris
Stage 5: Bin Starting Index

- Host allocates on device an array of length $N_b$ of pairs of unsigned integers, call it C.

- Run in parallel, on a per bin basis:
  - Load in parallel in shared memory chunks of the B array and find the location where each bin starts.
  - Store it in entry $k$ of C, as the key associated with this pair.
  - Empty bins set to maximum unsigned int value of 0xffffffff.
Stage 6: Sort C for Pruning

- Do a parallel radix sort on the array C based on the key
- Purpose: move unused bins to the end of array
- Effort: $O(N_b)$
Stage 7: Investigate Collisions in each Bin

- Carried out in parallel, one thread per bin

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>7</td>
<td>4</td>
<td>7</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

A1 A2 A2 A3 A3 A4 A4 A5 A5 B1 B1 ...

B-array

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>3</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
</tr>
</tbody>
</table>

C-array

| 1 | 2 | 2 | 2 | 2 | 3 | ... |

- To store information generated during this stage, host needs to allocate an unsigned integer array $D$ of length $N_b$
  - Array $D$ stores the number of actual contacts occurring in each bin
  - $D$ is in sync with (linked to) $C$, which in turn is in sync with (linked to) $B$

- Parallelism: one thread per bin
  - Thread $k$ reads the pair key-value in entry $k$ of array $C$
  - Thread $k$ reads does rehearsal for brute force collision detection
  - Outcome: the number $s$ of active collisions taking place in a bin
    - Value $s$ stored in $k^{th}$ entry of the $D$ array
Stage 7, details…

- In order to carry out this stage you need to keep in mind how C is organized, which is a reflection of how B is organized.

The drill: thread 0 relies on info at C[0], thread 1 relies on info at C[1], etc.

Let’s see what thread 2 (goes with C[2]) does:
- Read the first 2 bodies that start at offset 6 in B.
  - These bodies are 4 and 7, and as B indicates, they touch bin A4.
  - Bodies 4 and 7 turn out to have 1 contact in A4, which means that entry 2 of D needs to reflect this.
Stage 7, details...

- In order to carry out this stage you need to keep in mind how C is organized, which is a reflection of how B is organized.

The drill: thread 0 relies on info at C[0], thread 1 relies on info at C[1], etc.

Let’s see what thread 2 (goes with C[2]) does:
- Read the first 2 bodies that start at offset 6 in B.
  - These bodies are 4 and 7, and as B indicates, they touch bin A4.
  - Bodies 4 and 7 turn out to have 1 contact in A4, which means that entry 2 of D needs to reflect this.
Stage 7, details

- Brute Force CD rehearsal
  - Carried out to understand the memory requirements associated with collisions in each bin
    - Finds out the total number of contacts owned by a bin
  - Key question: which bin does a contact belong to?
    - Answer: It belongs to bin containing the CM of the Contact Volume (CMCV)
Stage 7, Comments

- Two bodies can have multiple contacts, handled ok by the method.

- Easy to define the CMCV for two spheres, two ellipsoids, and a couple of other simple geometries.
  - In general finding CMCV might be tricky.
    - Notice picture below, CM of 4 is in A5, CM of 7 is in B4 and CMCV is in A4.

- Finding the CMCV is the subject of the so called “narrow phase collision detection.”
  - It’ll be simple in our case since we are going to work with simple geometry primitives.
Stage 8: Inclusive Prefix Scan

- Save to the side the number of contacts in the last bin (last entry of $D$) $d_{\text{last}}$
  - Last entry of $D$ will get overwritten

$$N_c = D[N_b] + d_{\text{last}}$$
Stage 9: Populate Array E

- From the host, allocate on the device memory for array E
  - Array E stores the required collision information: normal, two tangents, etc.
  - Number of entries in the array: \( N_c \) (see previous slide)

- In parallel, on a per bin basis (one thread/bin):
  - Populate the E array with required info

- Not discussed in greater detail, this is just like Stage 7, but now you have to generate actual collision info (stage 7 was the rehearsal)

- Thread for A4 will generate the info for contact “c”
- Thread for C2 will generate the info for “i” and “d”
- Etc.
Stage 9, details

- B, C, D required to populate array E with collision information

- C and B are needed to compute the collision information
- D is needed to understand where the collision information will be stored in E
In this stage, parallelism is on a per bin basis

- Each thread picks up one entry in the array C
- Based on info in C you pick up from B the bin id and bodies touching this bin
- Based on info in B you run brute force collision detection
  - You run brute force CD for as long as necessary to find the number of collisions specified by array D
  - Note that in some cases there are no collisions, so you exit without doing anything
- As you compute collision information, you store it in array E
Tests Performed

1. Benchmark against Bullet
2. Scaling as number of contacts increases
3. Standard testing with multi-body dynamics simulations
Speedup - GPU vs. CPU (Bullet library)

GPU: NVIDIA Tesla C1060
CPU: AMD Phenom II Black X4 940 (3.0 GHz)
Parallel Implementation: Number of Contacts vs. Detection Time

![Graph showing the relationship between number of contacts and detection time. The graph is a linear regression line with increasing time as the number of contacts increases.]
Multi-GPU Collision Detection

Assembled Quad GPU Machine

Processor: AMD Phenom II X4 940 Black
Memory: 16GB DDR2
Graphics: 4x NVIDIA Tesla C1060
Power supply 1: 1000W
Power supply 2: 750W
Processing Overview

Main Data Set

Results

16 GB RAM

Quad Core AMD Microprocessor

Tesla C1060 4x4 GB Memory 4x30720 threads

Open MP

CUDA
Multi-GPU Collision Detection

- Split and organize data into Chunks
- Relying on OpenMP threads, one for each GPU
- Divide chunks into groups, GPUs work on chunk after chunk
- Combine collision data per group
- Combine collision data for all groups
Results – Contacts vs. Time

Quad Tesla C1060 Configuration used

- Time (sec) vs. Contacts (Billions)
- Time axis ranges from 0 to 40 seconds
- Contacts axis ranges from 0 to 40 billion
Ongoing: TARDEC Project

- Dynamics analysis of tracked vehicle running on sand/gravel
  - Relying of spherical decomposition of bodies in the model
  - Multiscale problem: Integration of terrain and track models
Spherical Decomposition

- Represent complex geometry as a union of spheres
  - Fast parallel collision detection on GPU
  - Allows non-convex geometry
Spherical Decomposition Demo

- Cubit
- Spherical Padding
Examples...

- Chain model
  - 10 links
  - 7,797 spheres per link

- Plow model
  - 31,791 spheres in plow blade model
  - 15,000 spheres representing terrain
GPU CCP Implementation
Scallable Parallel Implementation

- Approach draws on NVIDIA’s CUDA environment

Goal: parallelize the main iteration of the CCP solver:

\[
\gamma_i^{r+1} = \lambda \prod_{i} \eta_i \left( \gamma_i^r - \omega_i \left( D_i^T v^r + b_i \right) \right) + (1 - \lambda) \gamma_i^r
\]

\[
\Delta \gamma_i^{r+1} = \gamma_i^{r+1} - \gamma_i^r
\]

\[
\Delta v_i = M^{-1} D_i \Delta \gamma_i^{r+1}
\]
Data Structures on the GPU

- The buffer of rigid bodies

![Diagram of GPU bodies buffer with data categories]

- Velocity
- Position in reduction buffer
- Angular velocity
- Position
- Rotation (quaternion)
- J inertia and mass
- Applied force
- Applied torque
Data Structures on the GPU

- The contact buffer

![Diagram showing data structures on the GPU with labels for Residual (penetration), Body A index, Body B index, Friction, Multipliers (contact force), and Jacobian (translation), Jacobian (rotation), and (auxiliary data).]
Data Structures on the GPU

- Optimization: compute contact Jacobians in-place

- Vectors and indexes are feed from the collision detection engine

- Jacobian computation is an embarrassingly-parallel operation
Data Structures on GPU

- The buffer of bilateral constraints

Example:

1 Revolute joint → 5 scalar bilateral constraints

- Jacobian (translation, A)
- Jacobian (translation, B)
- Jacobian (rotation, A)
- Jacobian (rotation, B)
- Body A index
- Body B index
- Multiplier (reaction force)
- (auxiliary data)
- Residual (violation)
Data Structures on GPU

- The reduction buffer

\[
\gamma_{i}^{r+1} = \lambda \prod_{i} \left( \gamma_{i}^{r} - \omega \xi_{i} \left( D_{i}^{T}v^{r} + b_{i} \right) \right) + (1 - \lambda) \gamma_{i}^{r}
\]

\[
\Delta \gamma_{i}^{r+1} = \gamma_{i}^{r+1} - \gamma_{i}^{r} - \gamma_{i}^{r} : \Delta v_{i} = M^{-1} D_{i} \Delta \gamma_{i}^{r+1}.
\]

- Primal (velocities) update may cause write-conflicts when done in parallel!

- A reduction buffer avoids that N constraints might write to the same body data during constraint-parallel GPU code.

- Multiple updates to the same body are stored in the reduction buffer, then a following “reduction kernel” will sum them efficiently.

- Note: same scheme for contact-parallel code.
Reduction

Can reduction be performed efficiently in parallel?

- Reduction roughly means ‘sum all the values in a vector’
- Reduction on parallel architecture is not as easy as in serial architectures
- To keep all stream processors busy, an efficient parallel implementation must exploit the ‘binary tree’ concept at the right:
- Avoiding uncoalesced / misaligned memory access (otherwise performance might be memory-bandwidth limited)
Reduction

- The parallel reduction, in case of many bodies

- For each body there could be a single parallel-reduction problem!

- Better perform all reduction problems at once, to keep all processors busy.

- RMVLA: *Reduction of Multiple Variable Length Arrays* - our custom algorithm

- Some parallel reductions could take more steps than others. Reason: some body might touch 8 other bodies, but another body might touch 100 other bodies, etc. This issue is automatically managed by the hardware thread scheduler, at a cost of few divergence.
GPU Algorithm Pseudocode

- For each time step do:

  **Step 1**
  *(Host, serial)* Perform collision detection between bodies, obtaining $n_A$ possible contact points within a distance $\delta$, as contact positions $s_{i,A}$, $s_{i,B}$ on the two touching surfaces, and normals $n_i$. If warm start is used, then fetch last reactions in contact point $\gamma^*_i$ (obtained in previous frame, if the contact is persistent) and set $\gamma_i = \gamma^*_i$; otherwise set $\gamma_i = 0$.

  **Step 2**
  *(Host, serial)* Copy contact and body data structures from host memory to GPU buffers. Copy also constraint data (residuals $b_i$ and jacobians) into the constraint buffer. Note: also compute and store $R_{i,A}$, $R_{i,B}$, $n_{i,A}$ and $n_{i,B}$ in contact and constraint structures.

  **Step 3**
  *(GPU, body-parallel)* **Force kernel.** For each body, compute forces $f(t^{(l)}, q^{(l)}, v^{(l)})$, if any. Store these forces and torques into $F_j$ and $C_j$. For example, apply the gravitational and gyroscopic forces.
GPU Algorithm Pseudocode

Step 4

\textit{(GPU, contact-parallel) Contact preprocessing kernel.} For each contact, given contact normal and position, compute in-place the matrices $D_{i,v_A}^T$, $D_{i,\omega_A}^T$ and $D_{i,\omega_B}^T$, then compute $\eta_i$ and the contact residual $b_i = \left\{ \frac{1}{h} \Phi_i(q), 0, 0 \right\}^T$.

Step 5

\textit{(GPU, body-parallel) CCP force kernel.} For each body $j$, initialize body velocities: $\mathbf{r}_i^{(l+1)} = h \mathbf{m}_i^{-1} \mathbf{F}_j$ and $\mathbf{\omega}_i^{(l+1)} = h \mathbf{J}_i^{-1} \mathbf{C}_j$. 
GPU Algorithm Pseudocode

Substep 6 a

*(GPU, contact-parallel) CCP contact iteration kernel.* For each contact $i$, do
\[
\gamma_i^{r+1} = \lambda \Pi_{T_i} \left( \gamma_i^r - \omega \frac{\gamma_i^r}{\gamma_i^r} \left( \mathbf{D}^T \mathbf{v}^r + \mathbf{b}_i \right) \right) + (1 - \lambda) \gamma_i^r.
\]
Note that $\mathbf{D}_i^T \mathbf{v}^r$ is evaluated with sparse data, using Eq. (24). Store $\Delta \gamma_i^{r+1} = \gamma_i^{r+1} - \gamma_i^r$ in contact buffer. Compute sparse updates to the velocities of the two connected bodies $A$ and $B$, that is the four vectors of Eq. (26), and store them in the $R_{i,A}$ and $R_{i,B}$ slots of the reduction buffer. Also copy $n_{i,A}$ and $n_{i,B}$ in the same slots.

Substep 6 b

*(GPU, constraint-parallel) CCP constraint iteration kernel.* For each constraint $i$, do
\[
\gamma_i^{r+1} = \lambda \left( \gamma_i^r - \omega \frac{\gamma_i^r}{\gamma_i^r} \left( \mathbf{V}_i^T \mathbf{v}^r + \mathbf{b}_i \right) \right) + (1 - \lambda) \gamma_i^r.
\]
Note that $\mathbf{V}_i^T \mathbf{v}^r$ is evaluated with sparse data, using Eq. (27). Store $\Delta \gamma_i^{r+1} = \gamma_i^{r+1} - \gamma_i^r$ in contact buffer. Compute sparse updates to the velocities of the two connected bodies $A$ and $B$, that is the four vectors of Eq. (28), and store them in the $R_{i,A}$ and $R_{i,B}$ slots of the reduction buffer. Also copy $n_{i,A}$ and $n_{i,B}$ in the same slots.

Substep 6 c

*(GPU, reduction-slot-parallel) RMVLA binary reduction kernel.* Do an inner loop with this kernel, starting with $k = n_R$ and ending with $k = 1$. At the $k$-th inner iteration, for each slot of the reduction kernel, if the slot repetition counter $n \geq 2^{k-1}$, add slot values to the slot whose index is rearranged $2^{k-1}$ positions, and set counter to 0.

Substep 6 d

*(GPU, body-parallel) Body velocity updates kernel.* For each $j$ body, add the cumulative velocity updates which can be fetched from the reduction buffer, using the index $R_j$.

Repeat substep 6 until convergence or number of CCP steps satisfies $r > r_{max}$.59
GPU Algorithm Pseudocode

Step 7

*(GPU, body-parallel)* Time integration kernel. For each \( j \) body, perform time integration as:

\[
q_j^{(l+1)} = q_j^{(l)} + hL(q_j^{(l)})v_j^{(l+1)}
\]

Step 8

*(Host, serial)* Copy body data structures from GPU memory to host memory. Copy contact multipliers from GPU memory to host memory.

end
Numerical Experiments
Dynamics Engine

- All simulations performed using Chrono::Engine
  - Chrono::Engine Developed in Italy
  - GPU version of solver – joint project with Wisconsin
    - GPU Collision Detection
    - GPU CCP solver
Mixing 40000 Spheres
Tacked Vehicle Mobility

- Interested in dynamics of tracked vehicles (collaboration with TARDEC)
  - Mobility
  - Reaction forces for durability/fatigue analysis

Simulation in Chrono::Engine
1 Million spheres floating
Numerical Results: Pebble Bed Nuclear Reactor

- Two types of tests were run

- On the GPU
  - CD: NVIDIA 8800 GT
  - CCP: NVIDIA Tesla C870

- On the CPU
  - Single threaded
  - Quad Core Intel Xeon E5430
  - 2.66 GHz

- The reactor contains spheres which flow out the bottom the nozzle and are recycled to the top of the reactor

- Performed simulations with 16K, 32K, 64K, and 128K bodies
# Speedup, GPU vs CPU

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>16000</td>
<td>7.11</td>
<td>0.25</td>
<td>7.50</td>
<td>0.57</td>
<td>0.05</td>
<td>0.66</td>
<td>12.59</td>
<td>4.67</td>
<td>11.29</td>
</tr>
<tr>
<td>32000</td>
<td>16.01</td>
<td>0.67</td>
<td>16.98</td>
<td>1.00</td>
<td>0.11</td>
<td>1.20</td>
<td>16.07</td>
<td>6.14</td>
<td>14.11</td>
</tr>
<tr>
<td>64000</td>
<td>34.60</td>
<td>2.46</td>
<td>37.69</td>
<td>1.97</td>
<td>0.24</td>
<td>2.42</td>
<td>17.58</td>
<td>10.35</td>
<td>15.60</td>
</tr>
<tr>
<td>128000</td>
<td>76.82</td>
<td>11.28</td>
<td>89.44</td>
<td>4.55</td>
<td>0.52</td>
<td>5.55</td>
<td>16.90</td>
<td>21.71</td>
<td>16.12</td>
</tr>
</tbody>
</table>

Table 1: Timing results for PBR tests. Times reported represent average computation time per timestep of $\Delta t=0.01$ sec. Results obtained on a machine with an Intel Xeon E5430 2.66 GHz CPU and NVIDIA TESLA C1060.
\[ \Delta t = 0.01[\text{s}] \]

150 iterations
Results: Average Duration for Taking One Integration Step $[\Delta t=0.01 \text{ s}]$

Pebble Bed Reactor Demo, GPU vs CPU

- GPU Average Total Time
- CPU average Total Time
- Linear (GPU Average Total Time)
- Linear (CPU average Total Time)

- $y = 0.0007x - 6.3447$
  - $R^2 = 0.996$
- $y = 6E-05x - 0.51$
  - $R^2 = 0.9936$
Validation
Validation

- Carried out at the
  - At “macroscale” – Parma, Italy
  - At “microscale” – Wisconsin, US
    - Work in progress
Experimental Setups

- Two test beds built:
  - One ‘basic’ flat hopper
  - One 1:10 PBR reactor (half model)
Experimental Test Beds

- **Materials**
  - Plexiglas (for photo)
  - 180,000 plastic ABS spheres (6mm diameter)
  - Spheres—spheres and spheres—plexiglass friction found experimentally

- **Data acquisition**
  - Videocamera
  - High-speed videocamera (1000 fps)
  - High resolution cameras
  - Matlab™ image post-processing

- **Coloring**
  - Spheres are layered in different colors to ease image processing
  - Spheres can be washed and recolored with a chemical process
Flat Hopper Tests

Video recording from a test (a case that starts from high crystallization)
Flat Hopper Tests

3D rendering from a simulation (4x slower than real-time)
Flat Hopper Tests

- Example of output data from simulation
- Contact forces near the outlet
Flat Hopper Tests

- Comparison experimental - simulated
Flat Hopper Tests

Automated image processing for motion computation:

- Crop
- Color curve equalization
- Color area extraction
- Threshold
- Morphological operation (open/close)
- Speed profile computation
Flat Hopper Tests

- Agreement between experimental and simulated
Reactor Tests

- Granular flow tests in half reactor
  - We compare a simulated 1:10 half-PBR with the experimental 1:10 half-PBR testbed
  - Both cannot be compared exactly to the real full-PBR (the vertical wall friction affects the flow)
  - 180'000 pebbles, on a dual-core P4
  - Timesteps tested: from 0.01s to 0.001s
Reactor Tests

- Granular flow tests in half reactor
  - Cases: 30°, 60° funnel.
Reactor Tests

- Select results
  - Velocity speed profiles
  - Postprocessing in MATLAB

Non-averaged, with wall friction
Reactors Tests

- Vertical velocity iso-values, near the outlet
Reactor Tests

- Velocity profiles, in radial direction
- Validation at different heights
- Compared also with results of 1:10 MIT testbed
Reactor Tests

- Velocity iso values in the PBR core

- Very low radial dispersion, as shown in experiments and DEM simulations
Reactor Tests

- Volume fraction (1-porosity) as isovalues
Reactor Tests

- Comparing the porosity with results from DEM simulations at SANDIA
  - Porosity at different heights - good matching with DEM results (but significantly smaller computation times)
Conclusions

- Developed GPU parallel solver for multi-body systems

- Methods underlying numerical solution map well on GPU architecture

- Future improvements: AMG multigrid, multiple GPUs for CCP

- Applications:
  - Ground vehicle mobility on granular terrain (sand)
  - Material mixing for pharmaceuticals
  - Pebble bed nuclear reactors
  - Real-time simulators (VR, games)
Thank You.