On Hybrid Lagrangian-Eulerian Simulation Methods
Practical Notes and High-Performance Aspects

http://mpm.graphics

Yuanming Hu\textsuperscript{1}  Xinxin Zhang\textsuperscript{2}  Ming Gao\textsuperscript{2}  Chenfanfu Jiang\textsuperscript{3}

\textsuperscript{1}MIT CSAIL  \textsuperscript{2}Tencent  \textsuperscript{3}University of Pennsylvania
Speakers
Particle-Particle Particle-Mesh Method for fast N-Body dynamics in Eulerian-Lagrangian Computations
N-Body Dynamics

\[ u_i = \sum_{j=1, j \neq i}^{N} \frac{v_j \omega_j \times (x_i - x_j)}{4\pi \| x_i - x_j \|_2^3} \]

\[ f_i = -\epsilon \sum_{j=1, j \neq i}^{N} \frac{\rho_j v_j (x_i - x_j)}{4\pi \| x_i - x_j \|_2^3} \]
N-Body Dynamics

\[ u_i = \sum_{j=1, j \neq i}^{N} \frac{v_j \omega_j \times (x_i - x_j)}{4\pi \|x_i - x_j\|^3} \]

\[ f_i = -\epsilon \sum_{j=1, j \neq i}^{N} \frac{\rho_j v_j (x_i - x_j)}{4\pi \|x_i - x_j\|^3} \]

Given N particles and M evaluation position, direct computation requires \(O(NM)\) time!
On the Accurate Large-scale Simulation of Ferrofluids. Huang et. al. SIGGRAPH 2019

Surface reconstruction.

Fig. 5. An example of growing grass along the electric field.

Harmonic Parameterization by Electrostatics. Wang et. al. ACM TOG
Fast N-body Summation is Non-Trivial

Solve dynamics with only near-by influence is wrong

cut-off influence

full influence
Fast N-body Summations

• Solutions have been widely discovered to reduce this computation bottleneck.

<table>
<thead>
<tr>
<th>Accuracy</th>
<th>Efficiency</th>
<th>Code Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Excellent</td>
<td>Good</td>
<td>Fair</td>
</tr>
</tbody>
</table>

O(N) Fast Multipole Method (Rokhlin & Greengard)
Fast N-body Summations

• Solutions have been widely discovered to reduce this computation bottleneck.

O(G) Particle Mesh Methods, Vortex-In-Cell.

<table>
<thead>
<tr>
<th>Accuracy</th>
<th>Efficiency</th>
<th>Code Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fair</td>
<td>Excellent</td>
<td>Excellent</td>
</tr>
</tbody>
</table>
Fast N-body Summations: Particle-Particle, Particle-Mesh (PPPM)

Accuracy | Efficiency | Code Complexity
----------|------------|------------------
Good      | Excellent  | Excellent
**PPPM: Key idea**

\[ u_i = \sum_{j=1, j \neq i}^{N} \frac{v_j \omega_j \times (x_i - x_j)}{4\pi \|x_i - x_j\|_2^3} \]

\[ f_i = -\epsilon \sum_{j=1, j \neq i}^{N} \frac{\rho_j v_j (x_i - x_j)}{4\pi \|x_i - x_j\|_2^3} \]

\[ \nabla^2 \psi = -\omega \]

\[ u = \nabla \times \psi \]

\[ \nabla^2 \phi = \epsilon \rho \]

\[ f = \nabla \phi \]

Direct summation for the turbulent part

Poisson’s Equation for the smooth part
PPPM

• Fast solution uses near-far decomposition to get acceleration. Can we do similar thing on a particle-mesh setup?
PPPM

- Fast solution uses near-far decomposition to get acceleration. Can we do similar thing on a particle-mesh setup?
Figure 4: Performance of the PPPM fast summation. Computation time grows linearly with the number of computational elements.

Figure 5: Accuracy statistics of the PPPM fast summation.
Local Correction

• In 3D, for a correction window of size $K$ in each dimension, a local matrix of size $K^3 \times K^3$ can be precomputed to cancel the local influence from grid.

• $T(N) = O(c K^6 N)$
Local Correction
Local Correction

The influence made by neighbor cells.
Local Correction

• The matrix inverse reveals how the center cell’s value depends linearly on its neighbors (including itself).

\[ s_c = \sum_{j \in \eta} a_j r_j \]
One interesting finding

\[ a_j \approx G_{cj} = \frac{1}{4\pi \| x_c - x_j \|_2} \]
Even Simpler PPPM

• Decompose the velocity field as

\[ u = u_{\text{smooth}} + u_{\text{turbulent}} \]

• Where

\[ u_{\text{smooth}} = \text{Interpolate}(PM \ Solution) \]
\[ u_{\text{turbulent}} = \text{NearFieldSummation}(\omega_j - VIC) \]
Higher Order PPPM

• Compute Velocity Field not on Particles, but on SpGrid Nodes

• Allows tracking particle trajectories with higher order schemes, for example RK3.
Stable Vortex Segments

\[
|\omega^{n+1}_i|_2 = |\omega^n_i| L_i^{n+1} / L^n_i
\]

\[
|\omega^n_i|_2 = \frac{|\omega^0_i| L^n_i}{L^0_i} = \Gamma^0 L^n_i
\]
Better than remeshing

with remeshing only

with particle splitting and remeshing

Raising vortex ring simulation, camera moving with the target
Hybrid vortex-Eulerian simulation
Constitutive modeling: basic example
Constitutive modeling: basic example

- Keywords: constitutive relationship/behavior, strain stress curve, ...
Constitutive modeling: basic example

- Keywords: constitutive relationship/behavior, strain stress curve, ...
- General definition: local mechanical response under local kinematics
Constitutive modeling: basic example

- Keywords: constitutive relationship/behavior, strain stress curve, ...
- General definition: local mechanical response under local kinematics
- Example: Linear elasticity equilibrium (elastostatics)

**Kinematics describes strain and deformation**

infinitesimal strain (Cauchy strain):  
\[ \epsilon = \frac{1}{2}(\mathbf{F} + \mathbf{F}^T) - \mathbf{I} \]

Response comes from the constitutive model, relates to kinematics

linear elasticity:  
\[ \Psi(\mathbf{F}) = \mu \epsilon : \epsilon + \frac{\lambda}{2} \text{tr}(\epsilon)^2 \]
\[ \mathbf{P} = 2\mu \epsilon + \lambda \text{tr}(\epsilon) \mathbf{I} \]

**Governing equations are the the PDEs to be solved**

force balance (equilibrium):  
\[ \nabla^X \cdot \mathbf{P} + \mathbf{f}^{\text{ext}} = 0 \]
Constitutive modeling: basic example

- Keywords: constitutive relationship/behavior, strain stress curve, ...
- General definition: local mechanical response under local kinematics
- Example: Linear elasticity equilibrium (elastostatics)

Kinematics describes strain and deformation

infinitesimal strain (Cauchy strain):
\[ \epsilon = \frac{1}{2} (F + F^T) - I \]

Response comes from the constitutive model, relates to kinematics

linear elasticity:
\[ \Psi(F) = \mu \epsilon : \epsilon + \frac{\lambda}{2} \text{tr} (\epsilon)^2 \]
\[ P = 2\mu \epsilon + \lambda \text{tr} (\epsilon) I \]

Governing equations are the PDEs to be solved

force balance (equilibrium):
\[ \nabla^X \cdot P + f^{\text{ext}} = 0 \]

A core model in topology optimization
Continuum assumption of material kinematics

Continuum assumption

\[ x = \Phi(X, t) \]

\[ X = \Phi^{-1}(x, t) \]

\[ \Omega^0 \]

\[ \Omega^t \]
Continuum assumption of material kinematics

Continuum assumption

\[ \Omega^0 \]

\[ \Omega^t \]

Lagrangian kinematics

\[ \mathbf{V}(\mathbf{X}, t) = \frac{\partial \Phi}{\partial t} \]

\[ \mathbf{A}(\mathbf{X}, t) = \frac{\partial^2 \Phi}{\partial t^2} \]
Continuum assumption of material kinematics

Continuum assumption

\[ x = \Phi(X, t) \]
\[ X = \Phi^{-1}(x, t) \]

Lagrangian kinematics

\[ V(X, t) = \frac{\partial \Phi}{\partial t} \]
\[ A(X, t) = \frac{\partial^2 \Phi}{\partial t^2} \]

Eulerian velocity

\[ v(x, t) = V(\Phi^{-1}(x, t), t) \]
Continuum assumption of material kinematics

Continuum assumption

\[ \Omega^0 \]

\[ \Omega^t \]

Lagrangian kinematics

\[ \mathbf{V}(\mathbf{X}, t) = \frac{\partial \Phi}{\partial t} \]

\[ \mathbf{A}(\mathbf{X}, t) = \frac{\partial^2 \Phi}{\partial t^2} \]

Eulerian velocity

\[ \mathbf{v}(\mathbf{x}, t) = \mathbf{V}(\Phi^{-1}(\mathbf{x}, t), t) \]

Deformation gradient

\[ \mathbf{F}(\mathbf{X}, t) = \frac{\partial \Phi}{\partial \mathbf{X}} \]

\[ \mathbf{x} = \Phi(\mathbf{X}, t) \]

\[ \mathbf{X} = \Phi^{-1}(\mathbf{x}, t) \]
Continuum assumption of material kinematics

**Continuum assumption**

- Lagrangian kinematics:
  \[
  V(X, t) = \frac{\partial \Phi}{\partial t} \\
  A(X, t) = \frac{\partial^2 \Phi}{\partial t^2}
  \]

- Eulerian velocity:
  \[
  v(x, t) = V(\Phi^{-1}(x, t), t)
  \]

**Deformation gradient**

\[
F(X, t) = \frac{\partial \Phi}{\partial X}
\]

**Volume change**

\[
J(X, t) = \det(F(X, t))
\]
Stress and dynamics

\[ \Omega^t \]

\[ B^t \]
Stress and dynamics

\[ t = \sigma n \]

*traction: force per unit area*

*outward pointing normal*

\[ \sigma \]  *Cauchy stress tensor*
Stress and dynamics

Net force due to contact with exterior

\[ f_{B^t} = \int_{\partial B^t} t \, ds(x) = \int_{\partial B^t} \sigma n \, ds(x) = \int_{B^t} \nabla^x \cdot \sigma \, dx \]

traction: force per unit area

outward pointing normal

\( t = \sigma n \)

Cauchy stress tensor
Stress and dynamics

\[ t = \sigma n \]

traction: force per unit area

outward pointing normal

\[ \sigma \]

Cauchy stress tensor

Net force due to contact with exterior

\[ f_{B^t} = \int_{\partial B^t} t ds(x) = \int_{\partial B^t} \sigma n ds(x) = \int_{B^t} \nabla x \cdot \sigma dx \]

Conservation of momentum

\[ \rho \frac{Dv}{Dt} = \nabla \cdot \sigma + \rho g. \]
MPM and MLS-MPM from the weak form

**Strong form**

\[ \rho \frac{Dv}{Dt} = \nabla \cdot \sigma + \rho g \]  
(conservation of momentum)

**Weak form**

\[ \frac{1}{\Delta t} \int_{\Omega t^n} \rho(x, t^n) \left( \hat{\omega}_\alpha^{n+1}(x) - \omega^n_\alpha(x) \right) q_\alpha(x, t^n) \, dx \]

\[ = \int_{\partial \Omega t^n} q_\alpha(x, t^n) T_\alpha(x, t^n) \, ds - \int_{\Omega t^n} q_\alpha, \beta(x, t^n) \sigma_{\alpha \beta}(x, t^n) \, dx \]

**B-Spline MPM kernels**

\[ q_\alpha(x, t^n) = N_i(x) q^n_{i \alpha} \]

**MLS-MPM kernels**

\[ q_\alpha(x, t^n) = P^T(x - x^n_p) M^{-1}(x^n_p) \xi_i(x^n_p) P(x_i^n - x^n_p) \delta_{\alpha \hat{\alpha}} \]
Lagrangian Kinematics of Continuum
Lagrangian Kinematics of Continuum

Full Lagrangian kinematics

\[ \Phi(X, t) \]

\[ \Omega^0 \]

\[ \frac{dX}{dx} \]

\[ \Omega^t \]
Lagrangian Kinematics of Continuum

Full Lagrangian kinematics

Updated Lagrangian kinematics

The deformation flows through updating the current configuration.

\[ \hat{\Phi}(x^n) = \Phi(\Phi^{-1}(x^n, t^n), t^{n+1}) \]
Evolve the deformation

\[ F_p^{n+1} = (I + \Delta t \nabla v_p^{n+1}) F_p^n \]
Evolve the deformation

\[ \hat{\Phi}(x_p^n) = \Phi(\Phi^{-1}(x_p^n, t^n), t^{n+1}) \]

\[ \frac{\partial \hat{\Phi}}{\partial x_p^n} = \frac{\partial \Phi}{\partial X}(X, t^{n+1}) \left( \frac{\partial \Phi}{\partial X}(X, t^n) \right)^{-1} \]

\[ \hat{f} = F^{n+1}(F^n)^{-1} \]

\[ \hat{\Phi} = x_p^n + \Delta t \sum_i v_i^{n+1} w_{ip} \]

\[ \hat{f} = I + \Delta t \sum_i v_i^{n+1} \nabla w_{ip} \]

\[ F_{p}^{n+1} = (I + \Delta t \nabla v_{p}^{n+1})F_{p}^{n} \]

F is updated using velocity gradient
MPM is hybrid Lagrangian/Eulerian

**PARTICLE DOMAIN (LAGRANGIAN)**

1. Particle states
2. Trial elastic deformation gradient
3. Updated deformation gradient
4. Updated positions
5. Return mapping
6. Grid velocity and mass
7. Grid forces
8. Velocity update
9. Particles velocities

**GRID VIEW (EULERIAN)**
MPM is hybrid Lagrangian/Eulerian

Grid handles: the Galerkin DOFs, discretization, function space, ....
MPM is hybrid Lagrangian/Eulerian

- Grid handles: the Galerkin DOFs, discretization, function space, ….
- The transfer/embedding handles: coupling, quadrature rule, non-penetration, topology change, …
MPM is hybrid Lagrangian/Eulerian

✦ Grid handles: the Galerkin DOFs, discretization, function space, ....
✦ The transfer/embedding handles: coupling, quadrature rule, non-penetration, topology change, ...
✦ Each individual particle handles: constitutive modeling - the physics
Elasticity, hyperelasticity

- Hyperelasticity is convenient
  - Capturing real materials
  - Artistic design
  - Anisotropy, damping ...

\[
\Psi(F) = \frac{\mu}{2} (\text{tr}(FF^T) - 3) - \mu \log(J) + \frac{\lambda}{2} \log^2(J)
\]

\[
\Psi(F) = \mu \|F - R\|_F^2 + \frac{\lambda}{2} (\det(F) - 1)^2
\]

\[
\Psi(F) = \mu \text{tr}((\log \Sigma)^2) + \frac{\lambda}{2} (\text{tr}(\log \Sigma))^2
\]

\[
\Psi(F) = f(\sigma_1) + f(\sigma_2) + f(\sigma_3) + g(\sigma_1 \sigma_2) + g(\sigma_2 \sigma_3) + g(\sigma_3 \sigma_1) + h(\sigma_1 \sigma_2 \sigma_3)
\]
Hyperelasticity is convenient

- Capturing real materials
- Artistic design
- Anisotropy, damping …

\[ \Psi(\mathbf{F}) = \frac{\mu}{2} (\text{tr}(\mathbf{F}^2) - 3) - \mu \log(J) + \frac{\lambda}{2} \log^2(J) \]

\[ \Psi(\mathbf{F}) = \mu \|\mathbf{F} - \mathbf{R}\|_F^2 + \frac{\lambda}{2} (\text{det}(\mathbf{F}) - 1)^2 \]

\[ \Psi(\mathbf{F}) = \mu \text{tr}((\log \mathbf{\Sigma})^2) + \frac{\lambda}{2} (\text{tr}(\log \mathbf{\Sigma}))^2 \]

\[ \Psi(\mathbf{F}) = f(\sigma_1) + f(\sigma_2) + f(\sigma_3) + g(\sigma_1\sigma_2) + g(\sigma_2\sigma_3) + g(\sigma_3\sigma_1) + h(\sigma_1\sigma_2\sigma_3) \]
Inelasticity
[Jiang17] Anisotropic Elastoplasticity for Cloth, Knit and Hair Frictional Contact, Chenfanfu Jiang, Theodore Gast, Joseph Teran, SIGGRAPH 2017
[Jiang17] Anisotropic Elastoplasticity for Cloth, Knit and Hair Frictional Contact, Chenfanfu Jiang, Theodore Gast, Joseph Teran, SIGGRAPH 2017
Continuum damage mechanics
MPM is in Bifrost [Maya] starting yesterday

TRY IT NOW!
MPM is in Bifrost[Maya] starting yesterday

TRY IT NOW!
MPM is in Bifrost[Maya] starting yesterday

TRY IT NOW!
MPM Implementation & Performance Considerations

Yuanming Hu
MIT CSAIL
The Material Point Method (MPM)
The Material Point Method (MPM)

Particles (Constitutive models)
Snow [Stomakhin et al. 2013],
Foam [Ram et al. 2015, Yue et al. 2015]
Sand [Klar et al. 2015, Pradhana et al 2017]
The Material Point Method (MPM)

Particles (Constitutive models)
Snow [Stomakhin et al. 2013],
Foam [Ram et al. 2015, Yue et al. 2015]
Sand [Klar et al. 2015, Pradhana et al. 2017]

Grid
SPGrid [Setaluri et al. 2014],
OpenVDB [Museth 2013]
Multiple Grids
[Pradhana et al. 2015]
The Material Point Method (MPM)

Particles (Constitutive models)
Snow [Stomakhin et al. 2013],
Foam [Ram et al. 2015, Yue et al. 2015]
Sand [Klar et al. 2015, Pradhana et al. 2017]

Grid
SPGrid [Setaluri et al. 2014],
OpenVDB [Museth 2013]
Multiple Grids [Pradhana et al. 2015]

Particle to Grid (P2G)
Grid to Particle (G2P)
Transfer (Particle in Cell, PIC)
Affine PIC, APIC [Jiang et al. 2016]
Polynomial PIC, PolyPIC [Fu et al. 2017]
High-performance GIMP [Gao et al. 2017]
Moving Least Squares [Hu et al. 2018]
Compatible PIC [Hu et al. 2018]
The MPM Simulation Cycle

MPM Cycle

- Particle 2 grid (P2G)
- Grid Op
- Grid 2 particle (G2P)
The MPM Simulation Cycle

MPM Cycle

- Particle 2 grid (P2G)
- Grid Op
- Grid 2 particle (G2P)

For each particle:
Update particle
Scatter mass & momentum to nearby 3x3x3 nodes
The MPM Simulation Cycle

MPM Cycle

- Particle 2 grid (P2G)
- Grid Op
- Grid 2 particle (G2P)

For each particle:
- Update particle
- Scatter mass & momentum to nearby 3x3x3 nodes

For each grid node:
- Divide momentum by mass to get velocity; apply gravity.
The MPM Simulation Cycle

MPM Cycle

Particle 2 grid (P2G)

Grid Op

Grid 2 particle (G2P)

For each particle:
- Update particle
- Scatter mass & momentum to nearby 3x3x3 nodes

For each grid node:
- Divide momentum by mass to get velocity;
  apply gravity.

For each particle:
- Gather velocity info from 3x3x3 nodes;
  Update particle.
The MPM Simulation Cycle

MPM Cycle

- Particle 2 grid (P2G)
- Grid Op
- Grid 2 particle (G2P)

For each particle:
- Update particle
- Scatter mass & momentum to nearby 3x3x3 nodes

For each particle:
- Gather velocity info from 3x3x3 nodes;
  Update particle.
main simulation loop:

```c++
void advance(float dt) {
  // Reset grid
  std::memset(grid, 0, sizeof(grid));
  // P2G
  // main simulation loop
  // g = GridOp
  // p = ParticleOp
  // advance
}
```

88 Lines of C++

https://github.com/yuanming-hu/taichi_mpm

- 88-Line MLS-PM (cross-platform)
- High-Performance 3D MLS-PM+CPIC Solver on CPUs
- MLS-PM Tetris Game
- niall's MLS-PM implementation and tutorial in Unity
- Roberto Toro made mls-mpm.js that runs in your browser
- David Medina contributed mls-mpm88-explained.cpp
Demos!
Particle 2 Grid

for (auto &p : particles) {
    Vector2i base_coord=(p.x*inv_dx-Vec(0.5_f)).cast<int>(); //element-wise floor
    Vec fx = p.x * inv_dx - base_coord.cast<real>();
    // Quadratic kernels [http://mpm.graphics Eqn. 123, with x=fx, fx-1,fx-2]
    Vec w[3]{Vec(0.5) * sqr(Vec(1.5) - fx), Vec(0.75) - sqr(fx - Vec(1.0)),
        Vec(0.5) * sqr(fx - Vec(0.5))};
    auto e = std::exp(hardening * (1.0_f - p.Jp)), mu=mu_0*e, lambda=lambda_0*e;
    real J = determinant(p.F); // Current volume
    Mat r, s; polar_decomp(p.F, r, s); //Polar decomp. for fixed corotated model
    auto stress = -4*inv_dx*inv_dx*dt*vol*(2*mu*(p.F-r) * transposed(p.F)+lambda*(J-1)*J);
    auto affine = stress+particle_mass*p.C;
    for (int i = 0; i < 3; i++) for (int j = 0; j < 3; j++) { // Scatter to grid
        auto dpos = (Vec(i, j) - fx) * dx;
        Vector3 mv(p.v * particle_mass, particle_mass); //translational momentum
        grid[base_coord.x + i][base_coord.y + j] +=
            w[i].x*w[j].y * (mv + Vec3D(affine*dpos, 0));
    }
}
for (auto &p : particles) {
    Vector2i base_coord=(p.x*inv_dx-Vec(0.5_f)).cast<int>(); //element-wise floor
    Vec fx = p.x * inv_dx - base_coord.cast<real>();
    // Quadratic kernels [http://mpm.graphics Eqn. 123, with x=fx, fx-1,fx-2]
    Vec w[3]{Vec(0.5) * sqr(Vec(1.5) - fx), Vec(0.75) - sqr(fx - Vec(1.0)),
                Vec(0.5) * sqr(fx - Vec(0.5))};
    auto e = std::exp(hardening * (1.0_f - p.Jp)), mu=mu_0*e, lambda=lambda_0*e;
    real J = determinant(p.F); // Current volume
    Mat r, s; polar_decomp(p.F, r, s); //Polar decomp. for fixed corotated model
    auto stress = 4*inv_dx*inv_dx*dt*vol*(2*mu*(p.F-r) * transposed(p.F)+lambda*(J-1)*J);
    affine = stress+particle_mass*p.C;
    for (int i = 0; i < 3; i++) for (int j = 0; j < 3; j++) { // Scatter to grid
        auto dpos = (Vec(i, j) - fx) * dx;
        Vector3 mv(p.v * particle_mass, particle_mass); //translational momentum
        grid[base_coord.x + i][base_coord.y + j] +=
            w[i].x*w[j].y * (mv + Vector3(affine*dpos, 0));
    }
}
Particle 2 Grid

for (auto &p : particles) {
    Vector2i base_coord=(p.x*inv_dx-Vec(0.5f)).cast<int>(); // element-wise floor
    Vec fx = p.x * inv_dx - base_coord.cast<int>();
    // Quadratic kernels [http://mpm.graphics Eqn. 123, with x=fx, fx-1,fx-2]
    Vec w[3]{Vec(0.5) * sqr(Vec(1.5) - fx), Vec(0.75) - sqr(fx - Vec(1.0)),
        Vec(0.5) * sqr(fx - Vec(0.5))};
    auto e = std::exp(hardening * (1.0_f - p.Jp)), mu=mu_0*e, lambda=lambda_0*e;
    real J = determinant(p.F); // Current volume
    Mat r, s; polar_decomp(p.F, r, s); // Polar decomp. for fixed corotated model
    auto stress = -4*inv_dx*inv_dx*dt*vol*(2*mu*(p.F-r) * transposed(p.F)+lambda*(J-1)*J);
    auto affine = stress+particle_mass*p.c;
    for (int i = 0; i < 3; i++) for (int j = 0; j < 3; j++) { // Scatter to grid
        auto dpos = (Vec(i, j) - fx) * dx;
        Vector3 mv(p.v * particle_mass, particle_mass); // translational momentum
        grid[base_coord.x + i][base_coord.y + j] +=
            w[i].x*w[j].y * (mv + Vector3(affine*dpos, 0));
    }
}
for (auto &p : particles)  // Grid to particle
    Vector2i base_coord=(p.x*inv_dx-Vec(0.5_f)).cast<int>(); // element-wise floor
    Vec fx = p.x * inv_dx - base_coord.cast<real>();
    Vec w[3]{Vec(0.5) * sqr(Vec(1.5) - fx), Vec(0.75) - sqr(fx - Vec(1.0)),
               Vec(0.5) * sqr(fx - Vec(0.5))};
    p.C = Mat(0); p.v = Vec(0);
    for (int i = 0; i < 3; i++) for (int j = 0; j < 3; j++) {
        auto dpos = (Vec(i, j) - fx),
        grid_v = Vec(grid[base_coord.x + i][base_coord.y + j]);
        auto weight = w[i].x * w[j].y;
        p.v += weight * grid_v;  // Velocity
        p.C += 4 * inv_dx * Mat::outer_product(weight * grid_v, dpos); // APIC C
    }
    p.x += dt * p.v;  // Advection
auto F = (Mat(1) + dt * p.C) * p.F;  // MLS-MPM F-update
Mat svd_u, sig, svd_v; svd(F, svd_u, sig, svd_v);
for (int i = 0; i < 2 * int(plastic); i++)  // Snow Plasticity
    sig[i][i] = clamp(sig[i][i], 1.0_f - 2.5e-2_f, 1.0_f + 7.5e-3_f);
real oldJ = determinant(F); F = svd_u * sig * transposed(svd_v);
real Jp_new = clamp(p.Jp * oldJ / determinant(F), 0.6_f, 20.0_f);
Grid 2 Particle

for (auto &p : particles) {
    // Grid to particle
    Vector2i base_coord = (p.x * inv_dx - Vec(0.5_f)).cast<int>(); // element-wise floor
    Vec fx = p.x * inv_dx - base_coord.cast<real>();
    Vec w[3] {Vec(0.5) * sqr(Vec(1.5) - fx), Vec(0.75) * sqr(fx - Vec(1.0)), Vec(0.5) * sqr(fx - Vec(0.5))};
    p.C = Mat(0); p.v = Vec(0);
    for (int i = 0; i < 3; i++) for (int j = 0; j < 3; j++) {
        auto dpos = (Vec(i, j) - fx),
        grid_v = Vec(grid[base_coord.x + i][base_coord.y + j]);
        auto weight = w[i].x * w[j].y;
        p.v += weight * grid_v; // Velocity
        p.C += 4 * inv_dx * Mat::outer_product(weight * grid_v, dpos); // APIC C
    }
    p.x += dt * p.v; // Advection
    auto F = (Mat(1) + dt * p.C) * p.F; // MLS-MPM F-update
    Mat svd_u, sig, svd_v; svd(F, svd_u, sig, svd_v);
    for (int i = 0; i < 2 * int(plastic); i++) // Snow Plasticity
        sig[i][i] = clamp(sig[i][i], 1.0_f - 2.5e-2_f, 1.0_f + 7.5e-3_f);
    real oldJ = determinant(F); F = svd_u * sig * transposed(svd_v);
    real Jp_new = clamp(p.Jp * oldJ / determinant(F), 0.6_f, 20.0_f);
}
Grid 2 Particle

Particle Processing

Grid Gathering

(3^3=27 nodes in 3D!)
Grid 2 Particle

```cpp
for (auto &p : particles) {  // Grid to particle
    Vector2i base_coord=(p.x*inv_dx-Vec(0.5_f)).cast<int>(); // element-wise floor
    Vec fx = p.x * inv_dx - base_coord.cast<real>();
    Vec w[3]{Vec(0.5) * s2r(V(1.5) - fx), Vec(0.75) - s2r(fx - V(1.0)),
              Vec(0.5) * s2r(fx - Vec(0.5))};
    p.C = Mat(0); p.v = Vec(0);
    for (int i = 0; i < 3; i++) { for (int j = 0; j < 3; j++) {
        auto dpos = (V(i, j) - fx),
        grid_v = V(grid[base_coord.x + i][base_coord.y + j]);
        auto weight = w[i].x * w[j].y;
        p.v += weight * grid_v; // Velocity
        p.C += 4 * inv_dx * Mat::outer_product(weight * grid_v, dpos); // APIC-C
        p.x += dt * p.v; // Advection
        auto F = (Mat(1) + dt * p.C) * p.F; // MLS-PMF F-update
        Mat svd_u, sig, svd_v; svd(F, svd_u, sig, svd_v);
        for (int i = 0; i < 2 * int(plastic); i++) // Snow Plasticity
            sig[i][i] = clamp(sig[i][i], 4.0_f - 2.5e-2_f, 1.0_f + 7.5e-3_f);
        real oldJ = determinant(F); F = svd_u * sig * transposed(svd_v);
        real Jp_new = clamp(p.Jp * oldJ / determinant(F), 0.6_f, 20.0_f);
    }
}
```
Key Computational Patterns:
  a) Streaming Particle Data
  b) Particle-Grid Interaction
How much do implementation practices affect performance?

40%?
How much do implementation practices affect performance?

**10x higher performance!**

- Instruction-level parallelism
- Vectorization
- High-quality instruction generation
- Lock-free parallelization
- Bandwidth-saving data structures
- ...

without multithreading
Recap:
Modern Computer Architecture

- CPU
  - Float point unit
  - (a+b, a x b)

  Page Table & TLB

  Physical Memory

  \[ c = a + b \]
Recap:
Modern Computer Architecture

CPU
Float point unit
(a+b, a x b)

Page Table & TLB

Physical Memory

\[ c = a + b \]

1 FLoat Point OPeration (FLOP)
AVX2: 8 single precision/4 double precision float point number operations in a row
AVX512: 16 single precision/8 double precision float point number operations in a row
CPU

Float point unit
vectorized $a + b$, $a \times b$

FMA $a \times b + c$

Page Table & TLB

Physical Memory

Fused multiply add (FMA) operations
CPU

Float point unit
2xFMA a x b + c

Page Table & TLB

Physical Memory

4.2G Hz x 2 FMA/cycle x 16 FLOPs/FMA x 4 cores = 538G FLOPs
CPU

Float point unit
2xFMA a x b + c

Page Table & TLB

Cache

Physical Memory
CPU
- Float point unit
  - 2xFMA a x b + c

Page Table & TLB

L1 Cache

L2 Cache

L3 Cache

Physical Memory
(Part of) the Memory Hierarchy

- Execution Engine:
  - 4.20 GHz
  - 134.4 G FLOP/s, i.e. 806.4 GB/s bandwidth requirement

- Main Memory:
  - 35.8 GB/s
  - 256 cyc latency

- L3 cache 2M/core:
  - 134.4 GB/s
  - 42 cyc latency

- L2 cache 256KB:
  - 268.8 GB/s
  - 12 cyc latency

- L1 data cache 32KB:
  - 403.2 GB/s
  - 4 cyc latency

- L2 Unified TLB (STLB):
  - 4 KB/2MB pages - 1536 entries
  - 1G pages - 16 entries

- L1 Data TLB:
  - 4 KB pages - 64 entries
  - 2/4 MB pages - 32 entries
  - 1G pages - 4 entries

- Integer Physical Registers:
  - 8 bytes per entry, 180 entries
  - 1 cyc latency

- Vector Physical Registers:
  - 32 byte entries, 168 entries
  - 1 cyc latency

* Figures are not drawn to scale.
* Instruction caches are omitted.
* Main memory BW is shared by all cores.
Main Memory
35.8 GB/s
256 cyc latency

L3 cache 2M/core
134.4 GB/s
42 cyc latency

L2 cache 256KB
268.8 GB/s
12 cyc latency

L1 data cache 32KB
403.2 GB/s
4 cyc latency

L1 Data TLB
4 KB/2MB pages - 64 entries
2/4 MB pages - 32 entries
1G pages - 4 entries

L2 Unified TLB (STLB)
4 KB/2MB pages - 1536 entries
1G pages - 16 entries

CPU core

Integer Physical Registers
8 bytes per entry, 180 entries
1 cyc latency

Vector Physical Registers
32 byte entries, 168 entries
1 cyc latency

Execution Engine
4.20 GHz
134.4 G FLOP/s, i.e. 806.4 GB/s bandwidth requirement

* Figures are not drawn to scale.
* Instruction caches are omitted.
* Main memory BW is shared by all cores.

(Part of) the Memory Hierarchy

closer to CPU,
smaller capacity,
lower latency,
higher bandwidth.
CPU Core
- L1 data cache 32KB
  - 403.2 GB/s
  - 4 cyc latency
- L2 cache 256KB
  - 268.8 GB/s
  - 12 cyc latency
- L3 cache 2M/core
  - 134.4 GB/s
  - 42 cyc latency
- Main Memory
  - 35.8 GB/s
  - 256 cyc latency

* Caches are not drawn to scale.
* Data collected from the Intel Skylake architecture, single core.
* There can be multiple data transfers happening simultaneously.
* Access to slower memory is invoked by faster memory cache miss.
Main Memory
35.8 GB/s
256 cyc latency

L3 cache 2M/core
134.4 GB/s
42 cyc latency

L2 cache 256KB
268.8 GB/s
12 cyc latency

L1 data cache 32KB
403.2 GB/s
4 cyc latency

* Caches are not drawn to scale.
* Data collected from the Intel Skylake architecture, single core.
* There can be multiple data transfers happening simultaneously.
* Access to slower memory is invoked by faster memory cache miss.
Locality

✦ **Spatial locality**: try to access spatially neighboring data in main memory
  - Higher cacheline utilization
  - Fewer Cache/TLB misses
  - Better hardware prefetching on CPUs

✦ **Temporal locality**: reuse the data as much as you can
  - Higher cache-hit rates
  - Lower main memory bandwidth pressure

✦ Shrink the working set, so that data resides in lower-level (higher throughput, lower latency) memory
Main Memory
35.8 GB/s in total, 9 GB/s per core
256 cyc latency

Execution Engine
4.20 GHz
134.4 G FLOP/s, i.e. 806.4 GB/s bandwidth requirement
Main Memory
35.8 GB/s in total, **9 GB/s per core**
256 cyc latency

Without data reuse (temporal locality),
Processors need 100x higher bandwidth than what they actually have!

Execution Engine
4.20 GHz
134.4 G FLOP/s, **i.e. 806.4 GB/s bandwidth requirement**
The era of slow memory…

Figure from *High Performance by Exploiting Information Locality through Reverse Computing*, Bahi & Eisenbeis
...and parallelism.

35 YEARS OF MICROPROCESSOR TREND DATA

https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/

Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore
...and parallelism.
…and parallelism.

35 YEARS OF MICROPROCESSOR TREND DATA

- Transistors (thousands)
- Single-thread Performance (SpecINT)
- Frequency (MHz)
- Typical Power (Watts)
- Number of Cores

https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/

Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

#transistors keeps growing
Single-core performance stops growing
...and parallelism.

35 YEARS OF MICROPROCESSOR TREND DATA

#transistors keeps growing

Single-core performance stops growing

Instead of “better” cores, we have “more” cores (with wider SIMD)

https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/

Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore
...and parallelism.

Unlike 20 years ago, No “Free lunch” anymore!

Single-core performance stops growing
Instead of “better” cores, we have “more” cores (with wider SIMD)

https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/

Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore
Takeaways:

1. **[Architecture-aware programming]** will become increasingly important in the future since processors and memory are stopping getting faster.

2. Processors are more capable than you thought.  
   **[Parallelism: Computation is cheap]**

3. Memory bandwidth is a more scarce resource nowadays.  
   **[Locality: Communication is expensive]**
Takeaways:

1. [Architecture-aware programming] will become increasingly important in the future since processors and memory are stopping getting faster.

2. Processors are more capable than you thought. [Parallelism: Computation is cheap]

3. Memory bandwidth is a more scarce resource nowadays. [Locality: Communication is expensive]

Same rules apply to GPUs
The “Moore’s Law” of MPM

Performance 50x boost

Lines of Code 10x reduction

MpHz: Million particle x (Explicit Substep) Hertz
The Design Space for High Performance

✦ Data structures
  ○ For particles:
    ‣ array of structure (AOS)/ structure of arrays (SOA), sorting/reordering
  ○ For grid:
    ‣ Dense/sparse (VDB/SPGrid), dynamic/fixed hierarchy, leaf block data layout, Z-indexing…

✦ Parallelization:
  ○ what does each vector lane/thread/core/warp/block/streaming multiprocessor do?
  ○ How to avoid data race and cacheline sharing?
Organizing Particle Data
AOS v.s. SOA Layout

✧ Array of Structures (AOS)

```c
struct Particle {float x, y, z};
Particle particles[8192];
```

✧ Structure of Arrays (SOA)

```c
struct particles {
    float particle_x[8192];
    float particle_y[8192];
    float particle_z[8192];
};
```
Array of Structures

Linear Memory

Each particle’s fields are continuous in memory
Array of Structures: Sequential Access

Linear Memory

Cacheline size: x86_64: 64B; NVIDIA GPU: 128B
Array of Structures: Sequential Access

Linear Memory

Cacheline size: x86_64: 64B; NVIDIA GPU: 128B
Array of Structures: Sequential Access

Linear Memory

Cacheline

Cacheline size: x86_64: 64B; NVIDIA GPU: 128B
Array of Structures: Sequential Access

Linear Memory

Cacheline

Cacheline size: x86_64: 64B; NVIDIA GPU: 128B
Array of Structures: Sequential Access

Linear Memory

Cacheline

Cacheline size: x86_64: 64B; NVIDIA GPU: 128B
Array of Structures: Sequential Access

Linear Memory

Cacheline size: x86_64: 64B; NVIDIA GPU: 128B
Array of Structures: Sequential Access

Linear Memory

Cacheline size: x86_64: 64B; NVIDIA GPU: 128B
Array of Structures: Sequential Access

Linear Memory

Cacheline

Cacheline size: x86_64: 64B; NVIDIA GPU: 128B
Array of Structures: Sequential Access

Linear Memory

Cacheline

Cacheline size: x86_64: 64B; NVIDIA GPU: 128B
Array of Structures: Sequential Access

Linear Memory

Cacheline size: x86_64: 64B; NVIDIA GPU: 128B
Array of Structures: Sequential Access

Linear Memory

Cacheline

Cacheline size: x86_64: 64B; NVIDIA GPU: 128B

Cacheline Utilization: 100%
Array of Structures: Random Access

Linear Memory

Cacheline
Array of Structures: Random Access

Linear Memory

[x1 x2 y1 y2 z1 z2]

Cacheline
Array of Structures: Random Access

Linear Memory

Cacheline
Array of Structures: Random Access

Linear Memory

Cacheline
Array of Structures: Random Access

Linear Memory

Cacheline
Array of Structures: Random Access

Linear Memory

Cacheline
Array of Structures: Random Access

Linear Memory

Cacheline
Array of Structures: Random Access

Linear Memory

x1  y1  z1  x2  y2  z2

Cacheline
Array of Structures: Random Access

Linear Memory

Cacheline
Array of Structures: Random Access

Linear Memory

x1  y1  z1  x2  y2  z2

Unlikely used later

Cacheline
Array of Structures: Random Access

Linear Memory

Unlikely used later

Cacheline Utilization: 75%
Array of Structures

Linear Memory

Each particle’s fields are continuous in memory
Structure of Arrays

Linear Memory

Each field’s particle instances are continuous in memory
Structure of Arrays: Sequential Access

Linear Memory

x1  x2  y1  y2  z1  z2
Structure of Arrays: Sequential Access

Linear Memory

x1 x2 y1 y2 z1 z2
Structure of Arrays: Sequential Access
Structure of Arrays: Sequential Access

Linear Memory

x1 x2

y1 y2

z1 z2
Structure of Arrays: Sequential Access

Linear Memory

x1 x2 y1 y2 z1 z2
Structure of Arrays: Sequential Access

Linear Memory

- x1
- x2
- y1
- y2
- z1
- z2
Structure of Arrays: Sequential Access

Linear Memory

x1 x2 | y1 y2 | z1 z2
Structure of Arrays: Sequential Access

Linear Memory

\[
\begin{align*}
x_1 & \quad x_2 \\
y_1 & \quad y_2 \\
z_1 & \quad z_2
\end{align*}
\]
Structure of Arrays: Sequential Access

Linear Memory

x1 x2

y1 y2

z1 z2
Structure of Arrays: Sequential Access
Structure of Arrays: Sequential Access

Linear Memory

x1 x2

y1 y2

z1 z2
Structure of Arrays: Sequential Access

Linear Memory

x1 x2

y1 y2

z1 z2
Structure of Arrays: Sequential Access

Linear Memory

x1 x2

y1 y2

z1 z2
Structure of Arrays: Sequential Access

Cacheline Utilization: 100%
Structure of Arrays: Random Access

Linear Memory

Assuming cache size=3
Structure of Arrays: Random Access

Linear Memory

Assuming cache size=3
Structure of Arrays: Random Access

Assuming cache size=3
Structure of Arrays: Random Access

Assuming cache size=3
Structure of Arrays: Random Access

Assuming cache size=3
Structure of Arrays: Random Access

Linear Memory

x1 x2 y1 y2 z1 z2

Assuming cache size=3
Structure of Arrays: Random Access

Assuming cache size=3
Structure of Arrays: Random Access

Linear Memory

Assuming cache size=3
Structure of Arrays: Random Access

Linear Memory

x1 x2  y1  y2  z1  z2

Assuming cache size=3
Structure of Arrays: Random Access

Linear Memory

x1 x2 y1 y2 z1 z2

Assuming cache size=3
Structure of Arrays: Random Access

Linear Memory

x1 x2 y1 y2 z1 z2

Assuming cache size=3

Cacheline Utilization: 25%
Data Structures for MPM Particles

✦ SOA: very efficient when sorted
  ◢ Coalesced access on GPUs
  ◢ Large number of streams (e.g. 20): may invalidate prefetchers on CPU
    ‣ not a problem for GPU - GPUs are designed for streaming and have no prefetching
  ◢ Random access: low cacheline utilization

✦ AOS: efficient even unsorted
  ◢ Random access: much better than SOA but should still be avoided if possible (cache/TLB misses)
  ◢ Sequential access: Slightly inferior than AOS
    ‣ Vector lane shuffling on CPUs
    ‣ Non-coalesced access on GPUs
  ◢ No sorting needed
Real world 3D MPM: \(x(3\ \text{floats}),\ v(3\ \text{floats}),\ F(9\ \text{floats}),\ C(9\ \text{floats})\)…

<table>
<thead>
<tr>
<th>Particle Layout</th>
<th>Ordered</th>
<th>Randomly Shuffled</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOA</td>
<td>3.52ms</td>
<td>21.23 ms</td>
</tr>
<tr>
<td>AOS</td>
<td>3.15ms</td>
<td>4.28 ms</td>
</tr>
</tbody>
</table>

- SOA 6x slower when random shuffled
  - Periodic reordering is key to high performance for SOA
- AOS is less sensitive to particle order
Data Structures for MPM Grids

- Multi-Level Hierarchical Sparse Grids
  - e.g. VDB, SPGrid
- Dense blocks (e.g. 4x4x4 grid nodes) organized in tree structures
  - Grid nodes store velocity (momentum) and mass
Organizing Grid Data & Parallelization based on Grid Blocks
Partition Particle for **Parallelization**

Particle data

Naive Partition

Each core touches all grid nodes: **Poor locality!**
Partition Particle for **Parallelization**

Data race! Needs locks on CPUs: **Poor parallelism!**
Partition Particle for Parallelization

Particle data

Blockwise Partition

Particles in each block touches only a small number of grid nodes:
Good locality!
Partition Particle for Parallelization

Particle data

Blockwise Partition

4x4 grid nodes
Partition Particle for Parallelization

Particle data

Blockwise Partition
Partition Particle for **Parallelization**

Particle data

Blockwise Partition

One color at a time, in parallel
No data race - **good parallelism!**
Differentiable MLS-MPM Solver for Soft Robotics [Hu et al., ICRA 2019]

Direct gradient-based optimization of robots and their controllers

- By the “Chain” rule, hence the name “ChainQueen”
- Orders of magnitude faster than reinforcement learning!
On Hybrid Lagrangian-Eulerian Simulation Methods: Practical Notes and High-Performance Aspects

Yuanming Hu, Xinxin Zhang, Ming Gao*, Chenfanfu Jiang

Tencent America & University of Pennsylvania
On Hybrid Lagrangian-Eulerian Simulation Methods: Practical Notes and High-Performance Aspects

Yuanming Hu, Xinxin Zhang, Ming Gao*, Chenfanfu Jiang

Tencent America & University of Pennsylvania
On Hybrid Lagrangian-Eulerian Simulation Methods: Practical Notes and High-Performance Aspects

Yuanming Hu, Xinxin Zhang, Ming Gao*, Chenfanfu Jiang

Tencent America & University of Pennsylvania
GPU MPM - explicit time integration

Initial states $m_p, x_p, v_p^n$

Updated velocity $v_p^{n+1}$

Adveected position $x_p^{n+1}$

Transfer

Apply forces $m_i^n, v_i^n$

Transfer

Update sparse grid

Sparse grid

GPU
GPU MPM - explicit time integration

Initial states: $m^p_n, x^p_n, v^p_n$

Advected position: $x^p_{n+1}$

Updated velocity: $v^p_{n+1}$

Transfer

Apply forces: $m^i_n, v^i_n$

Transfer

Sparse grid

Update sparse grid

GPU
Pipeline

1: **procedure** GPUMPM()  
2: \( P \leftarrow \) Initialize particle positions  
3: \( P \leftarrow \) Sort and reorder \((P)\)  
4: **for** each time step **do**  
5: \( \text{dt} \leftarrow \) Compute \( \text{dt} \) \((P)\)  
6: \( G \leftarrow \) Refresh GSPGrid \((P)\)  
7: \( M \leftarrow \) Build particle-grid mapping \((P, G)\)  
8: \( G \leftarrow \) Transfer from particles to grid \((P, M)\)  
9: \( G \leftarrow \) Apply external forces \((G)\)  
10: \( G \leftarrow \) Solve on the grid \((G, \text{dt})\)  
11: \( P \leftarrow \) Transfer from grid to particles \((G, M)\)  
12: \( P \leftarrow \) Update particle attributes \((P, \text{dt})\)  
13: \( P \leftarrow \) Resort and reorder \((P)\)
Pipeline

1: procedure GPUMPM( )
2:     P ← Initialize particle positions
3:     P ← Sort and reorder (P)
4:     for each time step do
5:         dt ← Compute dt (P)
6:         G ← Refresh GSPGrid (P)
7:         M ← Build particle-grid mapping (P, G)
8:         G ← Transfer from particles to grid (P, M)
9:         G ← Apply external forces (G)
10:        G ← Solve on the grid (G, dt)
11:        P ← Transfer from grid to particles (G, M)
12:        P ← Update particle attributes (P, dt)
13:        P ← Resort and reorder (P)
Pipeline

1: procedure GPUMPM( )
2: P ← Initialize particle positions
3: P ← Sort and reorder (P)
4: for each time step do
5:   dt ← Compute dt (P)
6:   G ← Refresh GSPGrid (P)
7:   M ← Build particle-grid mapping (P, G)
8:   G ← Transfer from particles to grid (P, M)
9:   G ← Apply external forces (G)
10:  G ← Solve on the grid (G, dt)
11:  P ← Transfer from grid to particles (G, M)
12:  P ← Update particle attributes (P, dt)
13:  P ← Resort and reorder (P)
Pipeline

1: \textbf{procedure} GPUMPM( )
2: \hspace{1em} P ← Initialize particle positions
3: \hspace{1em} P ← Sort and reorder (P)
4: \hspace{1em} \textbf{for} each time step \textbf{do}
5: \hspace{1em} \hspace{1em} dt ← Compute dt (P)
6: \hspace{1em} \hspace{1em} G ← Refresh GSPGrid (P)
7: \hspace{1em} \hspace{1em} M ← Build particle-grid mapping (P, G)
8: \hspace{1em} \hspace{1em} G ← Transfer from particles to grid (P, M)
9: \hspace{1em} \hspace{1em} G ← Apply external forces (G)
10: \hspace{1em} \hspace{1em} G ← Solve on the grid (G, dt)
11: \hspace{1em} \hspace{1em} P ← Transfer from grid to particles (G, M)
12: \hspace{1em} \hspace{1em} P ← Update particle attributes (P, dt)
13: \hspace{1em} P ← Resort and reorder (P)
procedure GPUMPM()

1. P ← Initialize particle positions
2. P ← Sort and reorder (P)
3. for each time step do
4.   dt ← Compute dt (P)
5.   G ← Refresh GSPGrid (P)
6.   M ← Build particle-grid mapping (P, G)
7.   G ← Transfer from particles to grid (P, M)
8.   G ← Apply external forces (G)
9.   G ← Solve on the grid (G, dt)
10. P ← Transfer from grid to particles (G, M)
11. P ← Update particle attributes (P, dt)
12. P ← Resort and reorder (P)
Pipeline

1: **procedure** GPUMPM()
2: \( P \leftarrow \text{Initialize particle positions} \)
3: \( P \leftarrow \text{Sort and reorder (P)} \)
4: **for** each time step **do**
5: \( dt \leftarrow \text{Compute dt (P)} \)
6: \( G \leftarrow \text{Refresh GSPGrid (P)} \)
7: \( M \leftarrow \text{Build particle-grid mapping (P, G)} \)
8: \( G \leftarrow \text{Transfer from particles to grid (P, M)} \)
9: \( G \leftarrow \text{Apply external forces (G)} \)
10: \( G \leftarrow \text{Solve on the grid (G, dt)} \)
11: \( P \leftarrow \text{Transfer from grid to particles (G, M)} \)
12: \( P \leftarrow \text{Update particle attributes (P, dt)} \)
13: \( P \leftarrow \text{Resort and reorder (P)} \)
Pipeline

1: **procedure** GPUMPM()
2: \( P \leftarrow \) Initialize particle positions
3: \( P \leftarrow \) Sort and reorder \((P)\)
4: **for** each time step **do**
5: \( dt \leftarrow \) Compute \( dt \) \((P)\)
6: \( G \leftarrow \) Refresh GSPGrid \((P)\)
7: \( M \leftarrow \) Build particle-grid mapping \((P, G)\)
8: \( G \leftarrow \) Transfer from particles to grid \((P, M)\)
9: \( G \leftarrow \) Apply external forces \((G)\)
10: \( G \leftarrow \) Solve on the grid \((G, dt)\)
11: \( P \leftarrow \) Transfer from grid to particles \((G, M)\)
12: \( P \leftarrow \) Update particle attributes \((P, dt)\)
13: \( P \leftarrow \) Resort and reorder \((P)\)
1: procedure GPUMPM( )
2: P ← Initialize particle positions
3: P ← Sort and reorder (P)
4: for each time step do
5:     dt ← Compute dt (P)
6:     G ← Refresh GSPGrid (P)
7:     M ← Build particle-grid mapping (P, G)
8:     G ← Transfer from particles to grid (P, M)
9:     G ← Apply external forces (G)
10:    G ← Solve on the grid (G, dt)
11:    P ← Transfer from grid to particles (G, M)
12:    P ← Update particle attributes (P, dt)
13:    P ← Resort and reorder (P)
Pipeline

1: `procedure GPUMPMP( )`
2: \hspace{1cm} P ← Initialize particle positions
3: \hspace{1cm} P ← Sort and reorder (P)
4: \hspace{1cm} for each time step do
5: \hspace{2cm} dt ← Compute dt (P)
6: \hspace{2cm} G ← Refresh GSPGrid (P)
7: \hspace{2cm} M ← Build particle-grid mapping (P, G)
8: \hspace{2cm} G ← Transfer from particles to grid (P, M)
9: \hspace{2cm} G ← Apply external forces (G)
10: \hspace{2cm} G ← Solve on the grid (G, dt)
11: \hspace{2cm} P ← Transfer from grid to particles (G, M)
12: \hspace{1cm} P ← Update particle attributes (P, dt)
13: \hspace{1cm} P ← Resort and reorder (P)
Pipeline

1: procedure GPUMPM()
2: P ← Initialize particle positions
3: P ← Sort and reorder (P)
4: for each time step do
5:   dt ← Compute dt (P)
6:   G ← Refresh GSPGrid (P)
7:   M ← Build particle-grid mapping (P, G)
8:   G ← Transfer from particles to grid (P, M)
9:   G ← Apply external forces (G)
10:  G ← Solve on the grid (G, dt)
11:  P ← Transfer from grid to particles (G, M)
12:  P ← Update particle attributes (P, dt)
13:  P ← Resort and reorder (P)
Pipeline

1: procedure GPUMPM( )
2: P ← Initialize particle positions
3: P ← Sort and reorder (P)
4: for each time step do
5:     dt ← Compute dt (P)
6:     G ← Refresh GSPGrid (P)
7:     M ← Build particle-grid mapping (P, G)
8:     G ← Transfer from particles to grid (P, M)
9:     G ← Apply external forces (G)
10:    G ← Solve on the grid (G, dt)
11:    P ← Transfer from grid to particles (G, M)
12:    P ← Update particle attributes (P, dt)
13:    P ← Resort and reorder (P)
Pipeline

1: `procedure GPUMPM( )`
2: P ← Initialize particle positions
3: P ← Sort and reorder (P)
4: for each time step do
5:    dt ← Compute dt (P)
6:    G ← Refresh GSPGrid (P)
7:    M ← Build particle-grid mapping (P, G)
8:    G ← Transfer from particles to grid (P, M)
9:    G ← Apply external forces (G)
10:   G ← Solve on the grid (G, dt)
11:   P ← Transfer from grid to particles (G, M)
12:   P ← Update particle attributes (P, dt)
13:   P ← Resort and reorder (P)
Pure particle operations

1: `procedure GPUMPM( )`
2: \( P \leftarrow \text{Initialize particle positions} \)
3: \( P \leftarrow \text{Sort and reorder} \ (P) \)
4: `for each time step do`
5: \( dt \leftarrow \text{Compute dt} \ (P) \)
6: \( G \leftarrow \text{Refresh GSPGrid} \ (P) \)
7: \( M \leftarrow \text{Build particle-grid mapping} \ (P, G) \)
8: \( G \leftarrow \text{Transfer from particles to grid} \ (P, M) \)
9: \( G \leftarrow \text{Apply external forces} \ (G) \)
10: \( G \leftarrow \text{Solve on the grid} \ (G, dt) \)
11: \( P \leftarrow \text{Transfer from grid to particles} \ (G, M) \)
12: \( P \leftarrow \text{Update particle attributes} \ (P, dt) \)
13: \( P \leftarrow \text{Resort and reorder} \ (P) \)
Compute dt

1: procedure GPUMPM( )
2: P ← Initialize particle positions
3: P ← Sort and reorder (P)
4: for each time step do
5: "dt ← Compute dt (P)
6: G ← Refresh GSPGrid (P)
7: M ← Build particle-grid mapping (P, G)
8: G ← Transfer from particles to grid (P, M)
9: G ← Apply external forces (G)
10: G ← Solve on the grid (G, dt)
11: P ← Transfer from grid to particles (G, M)
12: P ← Update particle attributes (P, dt)
13: P ← Resort and reorder (P)
Compute dt

1: procedure GPUMPM( )
2: P ← Initialize particle positions
3: P ← Sort and reorder (P)
4: for each time step do
5:   dt ← Compute dt (P)
6:   G ← Refresh GSPGrid (P)
7:   M ← Build particle-grid mapping (P, G)
8:   G ← Transfer from particles to grid (P, M)
9:   G ← Apply external forces (G)
10:  G ← Solve on the grid (G, dt)
11:  P ← Transfer from grid to particles (G, M)
12:  P ← Update particle attributes (P, dt)
13:  P ← Resort and reorder (P)
Compute dt

1: procedure GPUMPM( )
2: P ← Initialize particle positions
3: P ← Sort and reorder (P)
4: for each time step do
5:   dt ← Compute dt (P)
6:   G ← Refresh GSPGrid (P)
7:   M ← Build particle-grid mapping (P, G)
8:   G ← Transfer from particles to grid (P, M)
9:   G ← Apply external forces (G)
10:  G ← Solve on the grid (G, dt)
11:  P ← Transfer from grid to particles (G, M)
12:  P ← Update particle attributes (P, dt)
13:  P ← Resort and reorder (P)

1. CFL
2. Max dt
Update particle properties

1: procedure GPUMPM()
2: P ← Initialize particle positions
3: P ← Sort and reorder (P)
4: for each time step do
5:  dt ← Compute dt (P)
6:  G ← Refresh GSPGrid (P)
7:  M ← Build particle-grid mapping (P, G)
8:  G ← Transfer from particles to grid (P, M)
9:  G ← Apply external forces (G)
10: G ← Solve on the grid (G, dt)
11: P ← Transfer from grid to particles (G, M)
12: P ← Update particle attributes (P, dt)
13: P ← Resort and reorder (P)
Update particle properties

1: procedure GPUMPM() 
2: P ← Initialize particle positions 
3: P ← Sort and reorder (P) 
4: for each time step do 
5:   dt ← Compute dt (P) 
6:   G ← Refresh GSPGrid (P) 
7:   M ← Build particle-grid mapping (P, G) 
8:   G ← Transfer from particles to grid (P, M) 
9:   G ← Apply external forces (G) 
10:  G ← Solve on the grid (G, dt) 
11:  P ← Transfer from grid to particles (G, M) 
12:  P ← Update particle attributes (P, dt) 
13:  P ← Resort and reorder (P)
Update particle properties

1: **procedure** GPUMPM()  
2: \( P \leftarrow \text{Initialize particle positions} \)  
3: \( P \leftarrow \text{Sort and reorder} (P) \)  
4: **for** each time step **do**  
5: \( dt \leftarrow \text{Compute dt} (P) \)  
6: \( G \leftarrow \text{Refresh GSPGrid} (P) \)  
7: \( M \leftarrow \text{Build particle-grid mapping} (P, G) \)  
8: \( G \leftarrow \text{Transfer from particles to grid} (P, M) \)  
9: \( G \leftarrow \text{Apply external forces} (G) \)  
10: \( G \leftarrow \text{Solve on the grid} (G, dt) \)  
11: \( P \leftarrow \text{Transfer from grid to particles} (G, M) \)  
12: \( P \leftarrow \text{Update particle attributes} (P, dt) \)  
13: \( P \leftarrow \text{Resort and reorder} (P) \)
SVD computation

Computing the Singular Value Decomposition of 3x3 matrices with minimal branching and elementary floating point operations

A. McAdams, A. Selle, R. Tamstorf, J. Teran and E. Sifakis
SVD computation

Computing the Singular Value Decomposition of 3x3 matrices with minimal branching and elementary floating point operations

A. McAdams, A. Selle, R. Tamstorf, J. Teran and E. Sifakis

0.37 ns per 3x3 matrix
SVD computation

Computing the Singular Value Decomposition of 3x3 matrices with minimal branching and elementary floating point operations

A. McAdams, A. Selle, R. Tamstorf, J. Teran and E. Sifakis

0.37 ns per 3x3 matrix
SVD computation

Computing the Singular Value Decomposition of 3x3 matrices with minimal branching and elementary floating point operations

A. McAdams, A. Selle, R. Tamstorf, J. Teran and E. Sifakis

0.37 ns per 3x3 matrix

Store them in memory???
SVD computation

![Graph showing SVD computation times for different scenarios. The graph compares 'Precompute-SVD', 'Stress', and 'Stress-derivative' with 'SVD on-the-fly' and 'precompute SVD' methods.](image-url)
SVD computation

On-the-fly/Load SVD Benchmark

ms

Precompute-SVD
Stress
Stress-derivative

SVD on-the-fly
precompute SVD
SVD computation

On-the-fly/Load SVD Benchmark

- Precompute-SVD
- Stress
- Stress-derivative

- SVD on-the-fly
- precompute SVD
SVD computation
SVD computation

On-the-fly/Load SVD Benchmark

- Precompute-SVD
- Stress
- Stress-derivative

SVD on-the-fly
precompute SVD
Resorting && reordering

1: procedure GPUMPM()
2: P ← Initialize particle positions
3: P ← Sort and reorder (P)
4: for each time step do
5:   dt ← Compute dt (P)
6: G ← Refresh GSPGrid (P)
7: M ← Build particle-grid mapping (P, G)
8: G ← Transfer from particles to grid (P, M)
9: G ← Apply external forces (G)
10: G ← Solve on the grid (G, dt)
11: P ← Transfer from grid to particles (G, M)
12: P ← Update particle attributes (P, dt)
13: P ← Resort and reorder (P)
Particle storage

Lexicographical curve

Morton coding curve

SPGrid
Setaluri et al 2014
Particle storage

Lexicographical curve

Morton coding curve

SPGrid
Setaluri et al 2014

Particles in one block

GPU memory

Particles in one cell
Radix vs histogram

\[(x,y,z) \rightarrow (i,j,k) \rightarrow 64\text{-bit offset}\]
Radix vs histogram

$(x, y, z) \rightarrow (i, j, k) \rightarrow 64$-bit offset
Radix vs histogram

\[(x,y,z) \rightarrow (i,j,k) \rightarrow 64\text{-}bit\, offset\]
Radix vs histogram

\[(x,y,z) \rightarrow (i,j,k) \rightarrow \text{64-bit offset}\]

52 bits + 12 bits

Continuous bins + 12 bits
Radix vs histogram

\[(x,y,z) \rightarrow (i,j,k) \rightarrow \text{64-bit offset}\]

- 52 bits + 12 bits
- Sparsely occupied

- Continuous bins + 12 bits
Radix vs histogram

- Radix
- Histogram (ours)

Time in ms:
- 0.4 million particles: Radix 0.4ms, Histogram 0.5ms
- 1.3 million particles: Radix 1.3ms, Histogram 1.4ms
- 3 million particles: Radix 3ms, Histogram 3ms
- 10 million particles: Radix 12ms, Histogram 12ms
Radix vs histogram
Radix vs histogram

Radix vs histogram (ours)
Reordering - memory movement
Reordering - memory movement

Old P: 1 2 3 4 5
New P: 5 4 2 3 1
Reordering - memory movement

Old P: 1 2 3 4 5
New P: 5 4 2 3 1
Reordering - memory movement

Old P: 1 2 3 4 5
New P: 5 4 2 3 1

Old V: [Diagram of colored squares]
Reordering - memory movement

Old P: 1 2 3 4 5

New P: 5 4 2 3 1

Old V

Old F
Reordering - memory movement

Old P: 1, 2, 3, 4, 5
New P: 5, 4, 2, 3, 1

Old V
Old F
Old E
Reordering - memory movement

<table>
<thead>
<tr>
<th>Old P</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>New P</th>
<th>5</th>
<th>4</th>
<th>2</th>
<th>3</th>
<th>1</th>
</tr>
</thead>
</table>
Reordering - memory movement

Old P
1 2 3 4 5

New P
5 4 2 3 1

Old V

New V

Old F

New F

Old E

New E

?
Reordering - memory movement

Old P: 1 2 3 4 5

New P: 5 4 2 3 1

Old V: ?? ?? ?? ?? ??

New V: ?? ?? ?? ?? ??

Old F: ?? ?? ?? ?? ??

New F: ?? ?? ?? ?? ??

Old E: ?? ?? ?? ?? ??

New E: ?? ?? ?? ?? ??
Pure memory operation - slow
Pure memory operation - slow

![Diagram showing comparison between reorder and no reorder in memory operations.](image-url)
Delayed reordering

Scattered inputs

1 2 3 4 5

Some function

Ordered outputs

5 4 2 3 1
Delayed reordering

![Bar chart showing delayed reordering times for different tasks.](chart.png)
Pure grid operations

1: procedure GPUMPM(
2:    P ← Initialize particle positions
3:    P ← Sort and reorder (P)
4:    for each time step do
5:        dt ← Compute dt (P)
6:        G ← Refresh GSPGrid (P)
7:        M ← Build particle-grid mapping (P, G)
8:        G ← Transfer from particles to grid (P, M)
9:        G ← Apply external forces (G)
10:       G ← Solve on the grid (G, dt)
11:       P ← Transfer from grid to particles (G, M)
12:       P ← Update particle attributes (P, dt)
13:       P ← Resort and reorder (P)
Pure grid operations

1: procedure GPUMPM( )
2: P ← Initialize particle positions
3: P ← Sort and reorder (P)
4: for each time step do
5:   dt ← Compute dt (P)
6:   G ← Refresh GSPGrid (P)
7:   M ← Build particle-grid mapping (P, G)
8:   G ← Transfer from particles to grid (P, M)
9:   G ← Apply external forces (G)
10:  G ← Solve on the grid (G, dt)
11:  P ← Transfer from grid to particles (G, M)
12:  P ← Update particle attributes (P, dt)
13:  P ← Resort and reorder (P)
Particle grid communications

1: **procedure** GPUMPM( )
2: P ← Initialize particle positions
3: P ← Sort and reorder (P)
4: **for** each time step **do**
5: dt ← Compute dt (P)
6: G ← Refresh GSPGrid (P)
7: M ← Build particle-grid mapping (P, G)
8: G ← Transfer from particles to grid (P, M)
9: G ← Apply external forces (G)
10: G ← Solve on the grid (G, dt)
11: P ← Transfer from grid to particles (G, M)
12: P ← Update particle attributes (P, dt)
13: P ← Resort and reorder (P)
Procedure GPUMPM() 

1. Initialize particle positions
2. Sort and reorder (P)
3. for each time step do
4. dt ← Compute dt (P)
5. G ← Refresh GSPGrid (P)
6. M ← Build particle-grid mapping (P, G)
7. G ← Transfer from particles to grid (P, M)
8. G ← Apply external forces (G)
9. G ← Solve on the grid (G, dt)
10. P ← Transfer from grid to particles (G, M)
11. P ← Update particle attributes (P, dt)
12. P ← Resort and reorder (P)
Build particle-grid mapping

1: procedure GPUMPM(
2:     P ← Initialize particle positions
3:     P ← Sort and reorder (P)
4: for each time step do
5:     dt ← Compute dt (P)
6:     G ← Refresh GSPGrid (P)
7:     M ← Build particle-grid mapping (P, G)
8:     G ← Transfer from particles to grid (P, M)
9:     G ← Apply external forces (G)
10:    G ← Solve on the grid (G, dt)
11:    P ← Transfer from grid to particles (G, M)
12:    P ← Update particle attributes (P, dt)
13:    P ← Resort and reorder (P)
Build particle-grid mapping

1: `procedure GPUMPMP( )`
2: \( P \leftarrow \text{Initialize particle positions} \)
3: \( P \leftarrow \text{Sort and reorder (P)} \)
4: `for each time step do`
5: \( dt \leftarrow \text{Compute dt (P)} \)
6: \( G \leftarrow \text{Refresh GSPGrid (P)} \)
7: \( M \leftarrow \text{Build particle-grid mapping (P, G)} \)
8: \( G \leftarrow \text{Transfer from particles to grid (P, M)} \)
9: \( G \leftarrow \text{Apply external forces (G)} \)
10: \( G \leftarrow \text{Solve on the grid (G, dt)} \)
11: \( P \leftarrow \text{Transfer from grid to particles (G, M)} \)
12: \( P \leftarrow \text{Update particle attributes (P, dt)} \)
13: \( P \leftarrow \text{Resort and reorder (P)} \)
Build particle-grid mapping

1: procedure GPUMPM( )
2:   P ← Initialize particle positions
3:   P ← Sort and reorder (P)
4: for each time step do
5:   dt ← Compute dt (P)
6:   G ← Refresh GSPGrid (P)
7:   M ← Build particle-grid mapping (P, G)
8:   G ← Transfer from particles to grid (P, M)
9:   G ← Apply external forces (G)
10:  G ← Solve on the grid (G, dt)
11:  P ← Transfer from grid to particles (G, M)
12:  P ← Update particle attributes (P, dt)
13:  P ← Resort and reorder (P)
Particle to grid (P2G)

1: procedure GPUMPM()
2: \hspace{5pt} P ← Initialize particle positions
3: \hspace{5pt} P ← Sort and reorder \((P)\)
4: \hspace{5pt} for each time step do
5: \hspace{10pt} \(\text{dt} ← \text{Compute } \text{dt} \text{ (P)}\)
6: \hspace{10pt} \(G ← \text{Refresh GSPGrid (P)}\)
7: \hspace{10pt} \(M ← \text{Build particle-grid mapping (P, G)}\)
8: \hspace{10pt} \(G ← \text{Transfer from particles to grid (P, M)}\)
9: \hspace{10pt} \(G ← \text{Apply external forces (G)}\)
10: \hspace{10pt} \(G ← \text{Solve on the grid (G, dt)}\)
11: \hspace{10pt} \(P ← \text{Transfer from grid to particles (G, M)}\)
12: \hspace{10pt} \(P ← \text{Update particle attributes (P, dt)}\)
13: \hspace{10pt} \(P ← \text{Resort and reorder (P)}\)
Particle to grid (P2G)

1: procedure GPUMPM()
2:     P ← Initialize particle positions
3:     P ← Sort and reorder (P)
4:     for each time step do
5:         dt ← Compute dt (P)
6:         G ← Refresh GSPGrid (P)
7:         M ← Build particle-grid mapping (P, G)
8:         G ← Transfer from particles to grid (P, M)
9:         G ← Apply external forces (G)
10:        G ← Solve on the grid (G, dt)
11:        P ← Transfer from grid to particles (G, M)
12:        P ← Update particle attributes (P, dt)
13:        P ← Resort and reorder (P)
Particle V.S. node per thread

CUDA thread - grid node
Particle V.S. node per thread

CUDA thread - grid node  v.S.
Particle V.S. node per thread

CUDA thread - grid node

V.S.

CUDA thread - particle
Scattering V.S. gathering

Particle
Grid node

CUDA thread -
Scattering V.S. gathering

Particle
Grid node

CUDA thread -
Scattering V.S. gathering

Particle
      Grid node

CUDA thread -
Scattering V.S. gathering

CUDA thread - •

Particle
Grid node
Scattering V.S. gathering

CUDA thread - •

Particle
Grid node
Scattering V.S. gathering

CUDA thread - •

Particle

Grid node
Scattering V.S. gathering

Particle
Grid node

CUDA thread -
Scattering V.S. gathering

Particle
Grid node

CUDA thread -
Scattering V.S. gathering

Particle
Grid node

CUDA thread -
Scattering V.S. gathering

CUDA thread - •
Scattering V.S. gathering

CUDA thread - •

CUDA thread - •

Particle

Grid node
Scattering V.S. gathering
Scattering V.S. gathering

CUDA thread - •

Particle list for a node - • • • • • • • • • • • • • • • • • •
Gathering - thread divergence

Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6

Particle
Grid node
Write hazard
Write hazard
Write hazard

Particle
Grid node

Thread 1
Thread 2
Write hazard
Write hazard

Particle

Grid node

Thread 1
Thread 2
Thread 3
Write hazard

Thread 1
Thread 2
Thread 3

Particle
Grid node
Hierarchy

Particle
Grid node

Cell

Geometric block
Hierarchy
Hierarchy
Hierarchy
Hierarchy

Hierarchy diagram showing the relationship between SMs, Warps, and Threads.
Hierarchy
Hierarchy

SM  SM

Warp  Warp  Warp

Thread  Thread  Thread

Hazard!

Hazard!
Hierarchy

Hazard!
Naive solution
Naive solution
Naive solution
Naive solution
Naive solution
Naive solution
Naive solution
Naive solution
Naive solution
Naive solution
Naive solution

Naive scattering
P2G - 90%
Naive solution

Naive scattering
P2G - 90%
Naive solution

Naive scattering
P2G - 90%
<table>
<thead>
<tr>
<th>lane id</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>node id</td>
<td>n</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
<td>n+2</td>
<td>n+3</td>
</tr>
</tbody>
</table>

iteration 0, stride 1

iteration 1, stride 2

mass sum $m_0 (m_1 + m_2) + (m_3 + m_4) (m_2 + m_3) + m_4 m_3 + m_4 m_5 + m_6 m_6 + m_7$
<table>
<thead>
<tr>
<th>lane id</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>node id</td>
<td>n</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
<td>n+2</td>
<td>n+3</td>
</tr>
<tr>
<td>lane id</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>---------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>node id</td>
<td>n</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
<td>n+2</td>
<td>n+3</td>
</tr>
<tr>
<td>lane id</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>---------</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>node id</td>
<td>n</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
<td>n+2</td>
<td>n+3</td>
</tr>
<tr>
<td>lane id</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>---------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>node id</td>
<td>n</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
<td>n+2</td>
<td>n+3</td>
</tr>
<tr>
<td>lane id</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>---------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>node id</td>
<td>n</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
<td>n+2</td>
<td>n+3</td>
</tr>
<tr>
<td>boundary mark</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>lane id</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>---------</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>node id</td>
<td>n</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
<td>n+2</td>
<td>n+3</td>
</tr>
<tr>
<td>boundary mark</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>region interval</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

CUDA intrinsic - ballot

region 0  region 1  region 2  region 3
<table>
<thead>
<tr>
<th>lane id</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>node id</td>
<td>n</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
<td>n+2</td>
<td>n+3</td>
</tr>
<tr>
<td>boundary mark</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>region interval</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**CUDA intrinsic - ballot**

**CUDA intrinsic - shfl**

**region 0**

**region 1**

**region 2**

**region 3**

<table>
<thead>
<tr>
<th>attribute mass</th>
<th>$m_0$</th>
<th>$m_1$</th>
<th>$m_2$</th>
<th>$m_3$</th>
<th>$m_4$</th>
<th>$m_5$</th>
<th>$m_6$</th>
<th>$m_7$</th>
</tr>
</thead>
</table>

**iteration 0, stride 1**

| mass sum | $m_0$ | $m_1 + m_2$ | $m_2 + m_3$ | $m_3 + m_4$ | $m_4$ | $m_5 + m_6$ | $m_6$ | $m_7$ |
CUDA intrinsic - ballot

CUDA intrinsic - shfl

CUDA intrinsic - shfl
### Lane ID and Node ID

<table>
<thead>
<tr>
<th>Lane ID</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node ID</td>
<td>n</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
<td>n+2</td>
<td>n+3</td>
</tr>
</tbody>
</table>

### Boundary Mark

| Node | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |

### Region Interval

| Region | 0 | 3 | 2 | 1 | 0 | 1 | 0 | 0 |

### Attribute Mass

<table>
<thead>
<tr>
<th>Region</th>
<th>attribute mass</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$m_0$</td>
</tr>
<tr>
<td>1</td>
<td>$m_1$ $m_2$</td>
</tr>
<tr>
<td>2</td>
<td>$m_3$ $m_4$</td>
</tr>
<tr>
<td>3</td>
<td>$m_5$ $m_6$ $m_7$</td>
</tr>
</tbody>
</table>

### Mass Sum

<table>
<thead>
<tr>
<th>Region</th>
<th>mass sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$m_0$</td>
</tr>
<tr>
<td>1</td>
<td>$m_1 + m_2$</td>
</tr>
<tr>
<td>2</td>
<td>$m_2 + m_3$</td>
</tr>
<tr>
<td>3</td>
<td>$m_3 + m_4$</td>
</tr>
<tr>
<td></td>
<td>$m_4$</td>
</tr>
<tr>
<td></td>
<td>$m_5 + m_6$</td>
</tr>
<tr>
<td></td>
<td>$m_6$</td>
</tr>
<tr>
<td></td>
<td>$m_7$</td>
</tr>
</tbody>
</table>

#### CUDA Intrinsic - ballot

#### CUDA Intrinsic - shfl

#### CUDA Intrinsic - shfl
CUDA intrinsic - ballot

<table>
<thead>
<tr>
<th>lane id</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>node id</td>
<td>n</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
<td>n+2</td>
<td>n+3</td>
</tr>
<tr>
<td>boundary mark</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>region interval</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

attribute mass

region 0 | region 1 | region 2 | region 3

| attribute mass | $m_0$ | $m_1$ | $m_2$ | $m_3$ | $m_4$ | $m_5$ | $m_6$ | $m_7$ |

CUDA intrinsic - shfl

iteration 0, stride 1

mass sum

| mass sum | $m_0$ | $m_1 + m_2$ | $m_2 + m_3$ | $m_3 + m_4$ | $m_4$ | $m_5 + m_6$ | $m_6$ | $m_7$ |

CUDA intrinsic - shfl

iteration 1, stride 2

mass sum

| mass sum | $m_0$ | $\frac{(m_1 + m_2)}{(m_3 + m_4)}$ | $\frac{(m_2 + m_3)}{m_4}$ | $m_3 + m_4$ | $m_4$ | $m_5 + m_6$ | $m_6$ | $m_7$ |
CUDA intrinsic - ballot

<table>
<thead>
<tr>
<th>lane id</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>node id</td>
<td>n</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+1</td>
<td>n+2</td>
<td>n+2</td>
<td>n+3</td>
</tr>
<tr>
<td>boundary mark</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>region interval</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

CUDA intrinsic - shfl

iteration 0, stride 1

<table>
<thead>
<tr>
<th>attribute mass</th>
<th>$m_0$</th>
<th>$m_1$</th>
<th>$m_2$</th>
<th>$m_3$</th>
<th>$m_4$</th>
<th>$m_5$</th>
<th>$m_6$</th>
<th>$m_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>mass sum</td>
<td>$m_0$</td>
<td>$m_1 + m_2$</td>
<td>$m_2 + m_3$</td>
<td>$m_3 + m_4$</td>
<td>$m_4$</td>
<td>$m_5 + m_6$</td>
<td>$m_6$</td>
<td>$m_7$</td>
</tr>
</tbody>
</table>

iteration 1, stride 2

<table>
<thead>
<tr>
<th>mass sum</th>
<th>$m_0$</th>
<th>$(m_1 + m_2)$</th>
<th>$(m_2 + m_3)$</th>
<th>$m_3 + m_4$</th>
<th>$m_4$</th>
<th>$m_5 + m_6$</th>
<th>$m_6$</th>
<th>$m_7$</th>
</tr>
</thead>
</table>

shared memory

node $n$, node $n+1$, node $n+2$, node $n+3$
NVIDIA TITAN Xp
Particles #: 7M
Grid res: 128^3

Benchmark

- CPU [Gao et al. 2017]
- CPU [Hu et al. 2018]
- GPU Naive Scattering
- GVDB [Wu et al. 2018]
- GPU Ours
Benchmark

NVIDIA TITAN Xp
Particles #: 7M
Grid res: 128^3

Ours is 15× faster than gathering and 23× faster than scattering with atomics
procedure GPUMPM()

1. P ← Initialize particle positions
2. P ← Sort and reorder (P)
3. for each time step do
4.    dt ← Compute dt (P)
5.    G ← Refresh GSPGrid (P)
6.    M ← Build particle-grid mapping (P, G)
7.    G ← Transfer from particles to grid (P, M)
8.    G ← Apply external forces (G)
9.    G ← Solve on the grid (G, dt)
10.   P ← Transfer from grid to particles (G, M)
11.   P ← Update particle attributes (P, dt)
12.   P ← Resort and reorder (P)
Particles: 4.2 M
Grid resolution: 256^3
Simulation: 10.48 secs/frame
Particles: 4.2 M
Grid resolution: 256^3
Simulation: 10.48 secs/frame
Thank you!

Source code - https://github.com/kuiwuchn/GPUMPM

Contact - ming.gao07@gmail.com