Parameterization and Control in Particle-based Fluid Dynamics

CSCI-7551 Parallel and Distributed Computing

Shane Transue
Introduction

- Parallel Fluid Simulation (Smoothed-particle Hydrodynamics)
  - **Objective:** Provide physically plausible non-viscous fluid behavior
    - Represent molecules by discrete particles
    - Utilize Newtonian dynamics (**emphasis:** physically plausible)
    - Introduce impulse-based collision resolution
    - Extract a surface that represents the fluid

- Composed of several algorithms that can be effectively parallelized
  - Parallel particle dynamics
  - Collision Detection/Resolution
  - Fluid surface generation (real-time tessellation)

- Prior techniques can be expanded to modern parallel architectures
  - SIMT (or grouped SIMD)
  - MIMD
Particle-based Fluid Dynamics
Particle-based Fluid Dynamics

Final Shaded Translucent Surface
Particle-based Fluid Dynamics

- **Research/Development Objective:**
  - Develop a physical simulation for particle-based fluid dynamics
    - Utilize particle dynamics to drive fluid (Simple parallel particle simulation)
    - Impulse-based rigid-body simulation (Mirtich, 96)
    - Similar approach in Particle-based Fluid Simulation by (Muller et al., 03)
    - Fluid Surface Generation: Marching Cubes (Lorensen and Cline, 87)

- **Physical Simulation Implementation**
  - **Objective:** Interactive/Real-time physical fluid simulation
  - Simultaneous GPU and Multi-core CPU execution
  - Development Languages/APIs
    - C++ (Main language)
    - OpenMP (MIMD Parallel language/API)
    - CUDA (SIMT Parallel language/API)
    - OpenGL (Graphics Interface API)
Approach Overview

- **Newtonian Particle Dynamics (GPU - SIMT)**
  - Explicit Ordinary Differential Equation (ODE) Solvers (Euler, Verlet, Runge-Kutta, etc.)
  - Rough approximation of molecules for plausible fluid behavior
  - Fluid behavior can be easily modified through impulses and external forces
  - Embarrassingly Parallel – What else can we do?

- **Collision Detection (Covered)**
  - Parallel Continuous Collision Detection (CCD)

- **Collision Resolution (CPU – MIMD)**
  - Parallel Impulse Solver (Bender and Schmitt, 06)
  - Cluster Graphs and particle contacts (MIMD [vs] SIMT)

- **Fluid Surface Extraction (GPU – Various Applications of SIMT)**
  - Parallel Marching Cubes
Approach Results
Newtonian Particle Dynamics (brief)

- **Objective:** Determine the velocity and position of a particle as a function of time utilizing Newton’s second law:

  \[ \vec{F}(t) = m \frac{d\vec{V}(t)}{dt} \]

- Formulate two initial value problems of the form: \( \frac{dy}{dx} = f(x, y) \), \( y(0) = y_0 \)

  **Velocity:** \( \frac{d\vec{V}(t)}{dt} = \frac{\vec{F}(t)}{m}, \vec{V}(t_0) = \vec{v}_0 \)

  \( \overrightarrow{v_0} = \text{Initial Velocity} \)

  **Position:** \( \frac{dX(t)}{dt} = \vec{V}(t), X(t_0) = x_0 \)

  \( x_0 = \text{Initial Position} \)

- This results in two initial value problems of which the solutions can be approximated using an ODE solver. Any solver can be exchanged simply by providing an alternative CUDA Kernel.

- Focus on parallelization not accuracy; therefore we can simply utilize Euler’s method for explicit forward integration.
Newtonian Particle Dynamics (cont’d)

- Euler’s Method (General Form): \( y_{n+1} = y_n + hf(x_n, y_n), \quad n \in \mathbb{N}^+ \)

- Time discretization: Form for both velocity and position using Euler’s method:

  **Velocity:** \( \frac{\tilde{v}_{i+1} - \tilde{v}_i}{t_{i+1} - t_i} = \frac{\vec{F}(t)}{m} \) \quad \text{Gives:} \quad \tilde{v}_{i+1} = \tilde{v}_i + h \frac{\vec{F}(t_i)}{m} \quad \text{where} \quad h = (t_{i+1} - t_i)

  **Position:** \( \frac{x_{i+1} - x_i}{t_{i+1} - t_i} = \vec{V}(t) \) \quad \text{Gives:} \quad x_{i+1} = x_i + h\vec{V}(t_{i+1}) \quad \text{where} \quad h = (t_{i+1} - t_i) \)
Newtonian Particle Dynamics (cont’d)

- Time-step duration is fixed to 30 or 60[fps]
  - i.e. $h = 1/30$ or $1/60$ respectively
- The following provides an implementation of this method in a CUDA kernel for $n$ particles:

```c
// All particles have an initial position, velocity, and some net external force
// Assume n = total threads for the kernel, inverse mass could be stored, etc.
__global__ void Euler(vec3* p, vec3* v, vec3* f, float* m, float h, int n) {
    unsigned int i = (blockIdx.x * blockDim.x) + threadIdx.x;
    v[i] = v[i] + (h * f[i] * (1/m[i]));
    p[i] = p[i] + (h * v[i]);
}
```
Parallel Particle Dynamics

- Embarrassingly Parallel nature of particle dynamics:
  - No interdependencies (check)
  - Direct mapping to device memory (check)
  - 1:1 Thread execution (check)

- Is there anything else we can do to improve performance?
  - Host to device memory transfers are expensive
  - How can the generated data be effectively reinterpreted for visualization?
  - How can we effectively utilize CPU/GPU parallelism?

- CUDA/OpenGL Interportability can be utilized to minimize these concerns
Objective: Minimize memory transfers between host and GPU when simulating and rendering the state of a physical simulation.

Memory Interportability – Why involve the host?
- CUDA Kernel: Generate new simulation state
- OpenGL Buffers/Texture: Render current simulation state
- Both are performed on the GPU

Interportability with CUDA and OpenGL
- Dedicate global device memory for both calculations in CUDA kernels and OpenGL reinterpretation for rendering – no host transfer
- Time-step: Dynamics update, then render (immediately)
- API synchronization

(Side note) Several Variants Exist
- CUDA with OpenGL/DirectX: Implementation is relatively simplistic
- OpenCL with OpenGL: Rather involved implementation – requires vendor OpenGL Extensions
CUDA/OpenGL Interportability Example

Initialize OpenGL/CUDA Context() {
    cudaGLSetGLDevice(0);
    glGenBuffers(1, &bufferName);
    glBindBuffer(GL_ARRAY_BUFFER, bufferName);
    glBufferData(GL_ARRAY_BUFFER, size, data, GL_DYNAMIC_DRAW);
    glVertexPointer(3, GL_FLOAT, 0, nullptr);
    glEnableClientState(GL_VERTEX_ARRAY);
...
    cudaGraphicsGLRegisterBuffer(&resource, bufferName, flags);
}
CUDA/OpenGL Interoperability Example

```c
Render() {
    cudaGraphicsMapResources(1, &resource);
    {
        float* bufferDev = nullptr;
        size_t size;
        cudaGraphicsResourceGetMappedPointer(&bufferDev, &size, resource);
        CUDA_KernelCall<<<Blocks, Threads>>> (bufferDev, count);
    }
    cudaGraphicsUnmapResources(1, &resource);
    ...
    glDrawArrays (GL_POINTS, 0, PARTICLE_COUNT);
}
```
Approach Overview

- **Newtonian Particle Dynamics (GPU - SIMT)**
  - Explicit Ordinary Differential Equation (ODE) Solvers (Euler, Verlet, Runge-Kutta, etc.)
  - Rough approximation of molecules for plausible fluid behavior
  - Fluid behavior can be easily modified through impulses and external forces
  - Embarrassingly Parallel – What else can we do?

- **Collision Detection (Covered)**
  - Parallel Continuous Collision Detection (CCD)

- **Collision Resolution (CPU – MIMD)**
  - Parallel Impulse Solver (Bender and Schmitt, 06)
  - Cluster Graphs and particle contacts (MIMD [vs] SIMT)

- **Fluid Surface Extraction (GPU – Various Applications of SIMT)**
  - Parallel Marching Cubes (Lorensen and Cline, 87)
Collision Resolution (High-level Overview)

- **Penalty-based Methods**
  - Continuous Penalty Forces (Tang et al., 12)
  - Requires penetration depth estimation (Heidelberger et al., 04)
  - Visible surface penetrations
  - Many techniques still rely on impulses for collision anyways (Drumwright, 08).

- **Constraint-based Methods**
  - Linear Complementary Problem (LCP)
    - System of nonlinear equations to determine constraint forces that cancel interpenetrations between each rigid-body
    - Computationally Expensive compared to iterative impulse approaches

- **Iterative Impulse-based Methods**
  - Constraint-based collision and contact handing using impulses (Bender and Schmitt, 06)
  - **Instantaneous impulse**: instantaneous change in velocity after a collision
  - *Fast!* (At the cost of accuracy – but for fluids the apparent difference is negligible)
  - Presents nontrivial challenges in parallelization
Iterative Impulse-based Collision Response

- **Continuous Collision Detection (Collision resolution input)**
  - Provides collision events tied to each pair of objects that are in contact (CCD result)
  - Physical state is provided at the instant of collision
  - CCD Guarantees no interpenetrations between colliding objects
  - Positions and velocities of each particle are known, collision events are generated for each contact point with a collision normal:

  CCD Resulting Contact States
  CCD Collision Events (Collision point and normal are illustrated in green)
Iterative Impulse-based Collision Response

- Constraint-based Collision and Contact Handling using Impulses (Bender and Schmitt, 06)
  - Calculate Relative Velocities (between particle pairs) to Separate Particles
    - Analyze the relative velocity in the direction of the collision normal:
      \[ v_{rel} = v_a - v_b \]
      \[ v_{rel,n} = v_{rel} \cdot \hat{n} \]
    - This leads to the following evaluation (later utilized to terminate iteration):
      \[ v_{rel,n} \begin{cases} < 0 & \text{Objects are Approaching} \\ > 0 & \text{Objects are Separating} \\ = 0 & \text{Objects are in contact} \end{cases} \]
  - Impulse Application
    - Using the evaluation above, apply an impulse \( j \) to particle \( a \) and \( -j \) to particle \( b \) such that we obtain: \( v_{rel,n} \geq 0 \)
    - This impulse dictates that the objects will then be separating and will not incur another collision in the next time-step
Iterative Impulse-based Collision Response

- Pair-wise sequential application of impulses
  - **Obvious problem:** Cannot handle multiple contacts
    - Separating two objects will cause new penetrations
    - Red region indicates this new interpenetration
    - Violation of the CCD constraint (no penetrations)
    - All contact points must be handled within a single simulation time-step

- **Solution:** Iterate until all pair-wise relative velocities are greater than or equal to 0
  - **Key:** Positions are not modified, only velocities
  - CCD Ensures the positions are correct (at contact)
  - Iterative process ends when all velocities are separating (or) no pair of contact points will satisfy the criterion for a collision in the next time-step
Parallel Impulse Collision Response

- **Problem:** This iterative process cannot be executed in parallel
  - Synchronization is required, reducing this to a sequential process
  - Current Approach: Group objects to iteratively solve in parallel (Schornbaum, 2010)
  - Introduced through: **Cluster Graphs**

- **What is a Cluster Graph?**
  - An undirected graph that describes collision contacts (as a cluster of particles in contact)
  - Particles that have one or multiple contacts qualify as nodes within a cluster graph
  - Edges represent collision events between the associated particles
  - The smallest cluster graph is defined as a pair-wise collision:

  ![Diagram](image)

  **Particles** $a$ and $b$ in contact (detected by CCD algorithm)
  **Cluster graph** representing the collision between particles $a$ and $b$
Parallel Impulse Collision Response

- What do we gain from using Cluster Graphs?
  - **We know:** Individual clusters contain interdependencies (cannot be calculated in parallel)
  - **Current Approach:** Calculate impulse response for each cluster graph

- What can go **wrong** with this approach?
  - All particles are in contact (degrades to sequential)
  - Stacks, Newton’s Cradle, Enclosed Volumes, etc.
Parallel Impulse Collision Response

- Architectural Implications of Cluster graph Collision resolution
  - **Problem:** Degradation of this algorithm can result in sequential execution
    - Terrible performance within SIMT (GPU)
    - One thread would be responsible for iterating through all collisions to generate impulses
    - Sequential computation cannot be avoided utilizing this method (all particles may collide)
  
  - **Solution:** MIMD with OpenMP
    - Assume simulation domain doesn’t always generate one cluster graph
    - 1 thread/process per cluster graph
    - Dynamic scheduling can divide thread load

- Like many things in computer graphics: it depends on the domain/application
- **Exploration:** Is there a better alternative for parallelization?
Approach Overview

- Newtonian Particle Dynamics (GPU - SIMT)
  - Explicit Ordinary Differential Equation (ODE) Solvers (Euler, Verlet, Runge-Kutta, etc.)
  - Rough approximation of molecules for plausible fluid behavior
  - Fluid behavior can be easily modified through impulses and external forces
  - Embarrassingly Parallel – What else can we do?

- Collision Detection (Covered)
  - Parallel Continuous Collision Detection (CCD)

- Collision Resolution (CPU – MIMD)
  - Parallel Impulse Solver (Bender and Schmitt, 06)
  - Cluster Graphs and particle contacts (MIMD [vs] SIMT)

- Fluid Surface Extraction (GPU – Various Applications of SIMT)
  - Parallel Marching Cubes (Lorensen and Cline, 87)
Fluid Surface Extraction

- Marching Cubes: A High Resolution 3D Surface Construction Algorithm
  - **Brief:** Generate 3D surfaces from multiple slices of computed tomography (CT Scans), magnetic resonance (MR), and single-photon emission computed tomography (SPECT)
  - **Purpose:** Generate 3D models to help physicians to understand the complex anatomy
  - Developed at General Electric (GE), (Lorensen and Cline, 87).

- Marching Squares
  - Provides a simplistic overview of the algorithm
  - Easy to understand (algorithm overview)

- Parallelization of Marching Cubes
  - Application of parallel prefix (sum) algorithm
    - Stream Compaction
    - Memory Address Offset Calculation
Marching Squares

- 2-Dimensional Analogue to Marching Cubes
  - Predefined 2D planar grid (controllable resolution)
  - The algorithm marches through each square providing an estimate of the contour

- **Input:** Grid resolution, Boolean function $f(x, y)$ that defines interior/exterior points within the sampled area (does not have to be a bounded region, the grid defines bounds)

- **Output:** Approximation of the contour that bounds the region (composed of simple primitives – in this instance, we use line segments)

- **Example:**

![Bounded Region and Approximated Contour](image)
Marching Squares (cont’d)

- Square-based Contour Approximation
  - Generates edges that approximate a 2D contour
  - $2^4$ Total possible combinations
    - Enumerated using inclusion (black) / exclusion (white)
    - Each identified with a unique binary code
  - Reducible to 6 through symmetry and rotation
    - Each defines a unique contour estimate (blue)
    - Inclusion patterns still exhibit ambiguous contours

All possible marching square node patterns
Marching Squares (cont’d)

- **Algorithm Overview**
  1. March through squares and lookup the appropriate pattern code
     i. Identify nodes within the bounded region (interior/exterior points)
     ii. Lookup appropriate code from predefined lookup table assign to this square
  2. Generate Geometry (line segments)
     i. Naïve Approach: Use the midpoint of the squares side
     ii. Better Approximation: Estimate the curves intersection with the squares side
  3. Render the result
Marching Cubes for Particles

- Dynamic System is composed of Particles
  - Need to approximate surface based on particle positions
  - Each particle contains an associated radius, this provides the 3D bounds function
  - If a grid point is within the radius of a particle, it will contribute to the generated fluid surface

Marching Cubes Result for Different Grid/Voxel Resolutions

- Grid size=10: 70 Facets
- Grid size=5: 220 Facets
- Grid size=2: 1700 Facets
- Grid size=1: 6800 Facets
- Grid size=0.5: 27000 Facets
Marching Cubes for Particles (cont’d)

☐ Algorithm Overview (Sequential)

1. March through cubes and lookup the appropriate pattern code
   i. Identify nodes within the bounded region (interior/exterior points)
      ▪ Boundary region defined by spheres (particles)
      ▪ Squared Euclidean Distance between a grid point \( g_i \) and particle \( p \)
        \[ d^2(p, g_i) = (p_x - g_{i,x})^2 + (p_y - g_{i,y})^2 + (p_z - g_{i,z})^2 \leq p_{radius} \]
      ▪ If this expression is true then the grid point is within the particle bounds
   ii. Lookup appropriate code from predefined lookup table assign to this cube

2. Generate Geometry (triangle sets)
   i. Naïve Approach: Use the midpoint of the cubes side
   ii. Better Approximation: Estimate the curves intersection with the cubes side

3. Render the result
Marching Cubes Unique Triangle Sets
Sequential Marching Cubes Visualization
Parallel Marching Cubes

[Step 1]: Each thread processes an individual cube:

- **Input**: Voxel grid and all particles

- Generate cube code and occupied flag
  - **Cube Code** – Index into lookup table that defines the triangles for this cube
  - **Occupied Flag** – If this cube has any triangles the value is true; otherwise false

- **Output**:  
  1. Array containing cube codes:  
     - 0x8 0x0 0xA 0xC 0x0 0x0 0x4 0x0  
     - 1 0 1 1 0 0 1 0
Parallel Marching Cubes (cont’d)

[Step 2] Cube Code Array **Compaction:**

- **Input:**
  - Cube code array
  - Occupied Array

- **Stream Compaction Requirements:**
  - Data Array
  - Predicate (Boolean mask – occupied array)
  - Implemented utilizing an exclusive parallel prefix sum
  - Index is offset by 1 to account for array indexing
  - Remove all data elements where the predicate is false (occupied array is the mask)

- **Output:** Compressed Cube Code Array
  - We only want to generate triangles for this set of cubes
Parallel Marching Cubes (cont’d)

[Step 3] Triangle Array Allocation

- Need to calculate the total number of triangles (for allocation)
- Need the offset for each triangle set (associated with a cube)
- These can both be calculated using a parallel prefix sum

**Input:**
- Array of Cube Codes
- Array to hold prefix result

1. Compute parallel prefix sum on the cube code array by looking up the number of vertices for each code:
2. The resulting offsets are calculated by the parallel prefix sum
3. The total number of vertices is $15+9 = 24$

We can now properly allocate and generate the geometry for all surface triangles
Parallel Marching Cubes (cont’d)

[Step 4] Render All Triangles

- OpenGL can interpret each vertex triplet as a triangle to rasterize
- We have duplicate vertices (more memory – removal takes time)

- **Optimization**: Utilize CUDA/OpenGL interportability
  - Similar to particle update/render pipeline
  - Memory for triangle vertices allocated on the GPU
  - Triangles reside within GPU memory
  - OpenGL can reinterpret the vertices as triangle primitives
Particle-based Fluid Visualization

- Marching Cubes generates a triangular surface that can be rendered using any graphics technique:
  - Alpha with depth functions
  - OpenGL Shaders (reflection, refraction, environment maps, etc)
  - Ray-tracing
Particle-based Fluid Visualization

- Volume Thickness
- Transparency (based on thickness)
- Final Opaque Surface with Reflections
- Final Shaded Translucent Surface
Parameterization and Control

- **Parameterization (Shannon’s Application)**
  - Derive fluid behavior from external input (very cool!)

- **Fluid Control**
  - Arbitrary external forces can impose fluid behaviors
    - Pattern-based (splash, push, lift, etc.)
    - Object-based (rigid-bodies can influence fluid flow)
    - **Function-based** \(-x^2 + 1\)
  - Control of incompressible and compressible fluid are both challenging
    - Can we exert pressure for fluid control?
    - Collision nightmare!
Conclusion

- Physically plausible fluid dynamics can be achieved through
  - Particle dynamics
  - Collision Detection/Resolution
  - Iso-surface generation (Marching Cubes)
  - Clever rendering techniques

- Additional Important Aspects
  - Parallel Prefix has several useful applications (building block for parallelization)
  - CUDA/OpenGL Interportability is easy to use and removes host-GPU transfers
  - MIMD [vs] SIMT
  - Application of OpenGL shaders/visual techniques can be used to effectively model fluid appearance
Conclusion: Fluid Dynamics!
References


