Toward efficient GPU-accelerated \( N \)-body simulations

Mark J. Stock, Ph.D. and Adrin Gharakhani, Sc.D.

Applied Scientific Research
Santa Ana, CA

Funding: NASA SBIR Phase I Grant and NCRR SBIR Phase II Grant
Definitions

- GPU - Graphics Processing Unit
- GPGPU - General-Purpose programming for GPUs
- Flop - Floating point operation
- GFlop/s - Billion Flops per second

Two Nvidia 9800GX2 GPUs > 1 TFlop/s

[Tom’s Hardware]
Motivation

- GPU performance doubles ~1 year
- CPU performance doubles ~2 years

NVIDIA CUDA Programming Guide
Motivation

- GPU performance doubles ~1 year
- CPU performance doubles ~2 years
Lagrangian vortex methods

• Navier-Stokes equations in vorticity

\[ \frac{D\omega}{Dt} = \omega \cdot \nabla u + \nu \nabla^2 \omega \]

• Discretized onto Lagrangian particles

\[ \bar{\omega}(\bar{x}, t) = \sum_{i=1}^{N_v} \bar{\Gamma}_i(t) \phi_\sigma(\bar{x} - \bar{x}_i) \]

\[ \bar{\Gamma}_i = \bar{\omega}_i \Delta V_i \]
Why vortex methods?

- **Grid-free** in the fluid domain (only mesh surfaces)
- *Not limited* by CFL convective instability (time steps 10x)
- Computational elements needed only where vorticity present
- *Free* from numerical diffusion (must be explicitly added)
- *Continuity conserved* by construction
CPU-direct method

- Perform $N^2$ Biot-Savart evaluations

$$\bar{u}_i (\bar{x}_i) = \sum_{j=1}^{N_v} K_\sigma (\bar{x}_j - \bar{x}_i) \times \bar{\Gamma}_j + U_\infty$$

$$K_\sigma(\bar{x}) = K(\bar{x}) \int_0^{\bar{x}/\sigma} 4\pi g(r) r^2 dr$$

$$K(\bar{x}) = -\frac{\bar{x}}{4\pi |\bar{x}^3|}$$

$$g(r) = (3/4\pi) \exp (-r^3)$$

- 3-component $u$ and 9-component $\nabla u$ is 72 Flops
CPU-direct method

• Perform $N^2$ Biot-Savart evaluations

\[
\bar{u}_i (\bar{x}_i) = \sum_{j=1}^{N_v} K_\sigma (\bar{x}_j - \bar{x}_i) \times \bar{\Gamma}_j + U_\infty
\]
CPU-direct method

• Perform $N^2$ Biot-Savart evaluations

$$
\bar{u}_i (\bar{x}_i) = \sum_{j=1}^{N_v} K_\sigma (\bar{x}_j - \bar{x}_i) \times \bar{\Gamma}_j + U_\infty
$$

$$
K_\sigma (\bar{x}) = K (\bar{x}) \int_0^{||\bar{x}||/\sigma} 4\pi g(r) r^2 dr
$$

$$
K (\bar{x}) = -\frac{\bar{x}}{4\pi ||\bar{x}^3||}
$$

$$
g(r) = (3/4\pi) \exp (-r^3)
$$

• 3-component $u$ and 9-component $\nabla u$ is 72 Flops
Faster summations


• Principle: use simplified representation for faraway groups of particles

• Application: if box is too close, examine box’s children, repeat...

• $O(N)$ to $O(N \log N)$
Our fast multipole treecode

1) Create tree structure

2) Compute multipole moments for each box

3) For each particle:
   • Recurse through tree starting at root
   • Calculate influence from source box using either multipole multiplication or Biot-Savart, as necessary
   • Add freestream
Our fast multipole treecode

• VAMSplit k-d trees (White and Jain, 1996)

• Only real multipole moments (Wang, 2004)

• 7 or 9 orders of multipole moments

• Barnes-Hut box-opening criteria, with extensions
  Barnes and Hut (1986), Warren and Salmon (1994)

• Interaction lists made uniquely for each particle
Serial performance

CPU times, random vortex particles in a cube

- Direct method: \( N_v^2 \)
- Treecode, 2e-4 error: \( N_v^{1.2} \)

CPU seconds vs. Number of particles, \( N_v \)
Serial performance

Speedup, vortex particles in a cube

Speedup vs. direct method

Number of particles, $N_v$

Present method, random, $2e^{-4}$

Wang, 2004, random, $2e^{-4}$

Strickland et al., 1999, uniform, $e^{-4}$
Parallel performance

Parallel performance vs. problem size

- 41880 s theoretical
- 10M particles: 268 s, 122%
- 1M particles: 2931 s, 23.6 s, 97%
- 100k particles: 205.1 s, theoretical 205.1 s, 77%
- 1M particles: 23.6 s, 97%
- 10M particles: 122%

Number of processors, P

Processor-seconds

10000 –
1000 –
100 –

1 2 4 8 16 32 64 128
GPU-direct method

- **BrookGPU** (Buck *et al.* 2004)
  - Looks like C language, converted to Cg
  - Define *streams* of data
  - OpenGL driver on Linux returns one set of `float4` per kernel

- **CUDA**: Compute Unified Device Architecture (NVIDIA, 2007)
  - C-like syntax, compiled by `nvcc`
  - Explicit control of memory on device
  - Kernels have few limits
GPU-direct method

- NVIDIA 8800 GTX has 8 multiprocessors, each with:
  - 16 processing elements (PEs)
  - 8192 registers
  - 16 banks of 16kB shared memory
GPU-direct method

1) Put all particles and field points on GPU main memory

2) Start one thread per field point (8 x 64 = 512 at a time)

3) For each group of 64 threads:
   • Load 64 source particles from main to shared memory
   • Calculate influences using Biot-Savart
   • Repeat; when done, write resulting $u$ and $\nabla u$ to main GPU memory

4) Read all $u$ and $\nabla u$ back to CPU memory
GPU-direct method

Interactions per second, velocity and velocity gradient

Interactions per second, velocity and velocity gradient

8800 GTX with CUDA
3.03 billion/sec
218 GFLOPS

8800 GTX with BrookGPU
938 million/sec
120 GFLOPS

Opteron 2216HE dual-core CPU
22.3 million/sec
1.6 GFLOPS

Number of particles, $N_v$
GPU treecode

1) Create tree structure  (CPU, single-thread)

2) Compute multipole moments for each box (CPU, multithread)

3) Determine all interaction lists (CPU, single-thread)

4) Calculate all influences from *far-field* using multipole mult.  
   (GPU)

5) Calculate all influences from *near-field* using Biot-Savart  
   (GPU)
GPU treecode

Runtime breakdown, all-CPU method, 500k particles, $2 \times 10^{-4}$ error

- Lists By Particle
  - CPU 2: 251.5 s
  - CPU 1: 251.5 s

- Lists By Box
  - CPU 2: 330.1 s
  - CPU 1: 330.1 s

Running time: 0 - 400 s

Legend:
- Build tree
- Compute moments
- Far- and near-field interactions
- Far-field (multipoles)
- Near-field (Biot-Savart)
GPU-direct method -- far-field

1) Put all particles, field points, and interaction lists on GPU

2) Start one thread per field point (8 x 64 = 512 at a time)

3) For each group of 64 threads:
   • Iterate through that group’s far-field interaction list
   • Load 210 multipole moments from GPU main to shared memory
   • Calculate influences using multipole multiplication
   • Repeat; when done, write resulting $u$ and $\nabla u$ to GPU main mem.

4) Read all $u$ and $\nabla u$ back to CPU main memory
GPU-direct method -- near-field

1) Put all particles, field points, and interaction lists on GPU

2) Start one thread per field point (8 x 64 = 512 at a time)

3) For each group of 64 threads:
   • Iterate through that group’s near-field interaction list
   • Load 64 source particles from GPU main to shared memory
   • Calculate influences using Biot-Savart
   • Repeat; when done, write resulting $u$ and $\nabla u$ to GPU main mem.

4) Read all $u$ and $\nabla u$ back to CPU main memory
GPU treecode

Runtime breakdown, 500k particles, 2 x 10^{-4} error

All-CPU

<table>
<thead>
<tr>
<th></th>
<th>GPU</th>
<th>CPU 2</th>
<th>CPU 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running</td>
<td>330.1 s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Build tree</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compute moments</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Far-field (multipoles)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Near-field (Biot-Savart)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hybrid CPU-GPU

<table>
<thead>
<tr>
<th></th>
<th>GPU</th>
<th>CPU 2</th>
<th>CPU 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running</td>
<td>25.7 s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Build tree</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compute moments</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Far-field (multipoles)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Near-field (Biot-Savart)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
GPU treecode

- Changing *bucket size* shifts work between near- and far-field

![Runtime breakdown, 2-way trees, 500k particles, 2e-4 error](image)
GPU treecode

Runtime breakdown, 500k particles, $2 \times 10^{-4}$ error

2-way, $N_b = 64$
- GPU: 25.7 s
- CPU 2
- CPU 1

8-way, $N_b = 512$
- GPU: 14.9 s
- CPU 2
- CPU 1

Running time
- Build tree
- Compute moments
- Far-field (multipoles)
- Near-field (Biot-Savart)
Best performance, 500k particles, $2 \times 10^{-4}$ mean velocity error

<table>
<thead>
<tr>
<th></th>
<th>Total time</th>
<th>Tree-building</th>
<th>Multipole moments</th>
<th>Velocity solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU direct</td>
<td>11242 s</td>
<td>-</td>
<td>-</td>
<td>11242 s</td>
</tr>
<tr>
<td>CPU treecode</td>
<td>251.5 s</td>
<td>2.0 s</td>
<td>11.7 s</td>
<td>237.8 s</td>
</tr>
<tr>
<td>GPU direct</td>
<td>88.7 s</td>
<td>-</td>
<td>-</td>
<td>88.7 s</td>
</tr>
<tr>
<td>GPU treecode</td>
<td>14.9 s</td>
<td>1.1 s</td>
<td>2.3 s</td>
<td>11.4 s</td>
</tr>
</tbody>
</table>
Dynamic simulation step

• Inviscid Flow - **Lagrangian Vortex Element Method**

\[
\frac{\partial \bar{x}_i}{\partial t} = \bar{u}_i
\]

\[
\frac{\partial \bar{\Gamma}_i}{\partial t} = \bar{\Gamma}_i \cdot \nabla \bar{u}_i
\]

• Diffusion - **Vorticity Redistribution** (Shankar and van Dommelen, 1996)

\[
\frac{\partial \bar{\Gamma}_i}{\partial t} = \nu \nabla^2 \bar{\Gamma}_i
\]
Dynamic simulation step

- Wall B.C. satisfied by generating vorticity at the wall - Boundary Element Method

\[
\frac{1}{2} (\bar{\gamma}A)_i \times \hat{n} + \sum_{m \neq i} (\bar{\gamma}A)_m \times \int_{S_i} K(\bar{x} - \bar{x}_m) \, d\bar{x} = \bar{Q}_s
\]

\[
\bar{Q}_s = \int_{S_i} \bar{U}_\infty(\bar{x}) \, d\bar{x} - \sum_{j=1}^{N_v} \bar{\Gamma}_j \times \int_{S_i} K(\bar{x} - \bar{x}_j) \, d\bar{x}
\]
Flow over 10:10:1 ellipsoid at 60°
Re_D=1000

SURFACE MESH
Flow over 10:10:1 ellipsoid at 60°

Runtime breakdown, GPU-CPU, 8-way trees, 8e-4 error

- CPU trees and moments
- GPU velocity calculation
- CPU VRM
- CPU BEM

Nv, number of particles

Calculation time (s)
Flow over 10:10:1 ellipsoid at 60°
$Re_D=1000$
Flow over 10:10:1 ellipsoid at 60°
Re_D=1000
Flow over 10:10:1 ellipsoid at 60°
Re_D=1000
Flow over 10:10:1 ellipsoid at 60°
Re_D=1000
Flow over 10:10:1 ellipsoid at 60°
Re_D=1000
Flow over 10:10:1 ellipsoid at 60°
Re_D=1000
Flow over 10:10:1 ellipsoid at 60°
Re$_D$=1000
Flow over 10:10:1 ellipsoid at 60°
\( \text{Re}_D = 1000 \)
Flow over 10:10:1 ellipsoid at 60°
$Re_D=1000$
Flow over 10:10:1 ellipsoid at 60°
Re_D=1000
Flow over 10:10:1 ellipsoid at 60°
$Re_D=1000$
Flow over 10:10:1 ellipsoid at 60°
$Re_D=1000$
Flow over 10:10:1 ellipsoid at 60°
Re_D=1000
Flow over 10:10:1 ellipsoid at 60°
Re_D=1000
Flow over 10:10:1 ellipsoid at 60°
Re_D=1000
Flow over 10:10:1 ellipsoid at 60°
Re_D=1000
Flow over 10:10:1 ellipsoid at 60°
Re_D=1000
Flow over 10:10:1 ellipsoid at 60°
$Re_D=1000$
Flow over 10:10:1 ellipsoid at 60°
Re_D=1000
Flow over 10:10:1 ellipsoid at 60°
Re_D=1000
Flow over 10:10:1 ellipsoid at 60°
$Re_D=1000$
Conclusions

• Vortex particle methods adapt to hybrid CPU-GPU systems
• 140x speedup for direct summations on GPU
• 10x-15x speedup for GPU treecode velocity-finding
• 3.4x speedup for full dynamic timestep
Conclusions

• GPU performance has increased rapidly...and should continue
Future Work

• Reorganize to more efficiently use CPU & GPU
• Put VRM diffusion onto GPU
• Test on distributed-memory CPU-GPU clusters
Future Work

• Reorganize to more efficiently use CPU & GPU
• Put VRM diffusion onto GPU
• Test on distributed-memory CPU-GPU clusters

Thank you for your attention! Any questions?

mstock@applied-scientific.com