GPU Acceleration of the Generalized Interpolation Material Point Method

Wei-Fan Chiang, Michael DeLisi, Todd Hummel, Tyler Prete, Kevin Tew, **Mary Hall**, Phil Wallstedt, and James Guilkey

Sponsored in part by NSF awards CSR-0615412 and OCI-0749360 and by hardware donations from NVIDIA Corporation.

Schlumberger THE UNIVERSITY OF UTAH

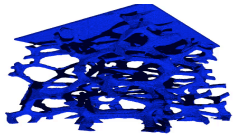
Outline

- What is Material Point Method and Generalized Interpolation Material Point Method?
- Suitability for GPU Acceleration
- Implementation Challenges
  - Inverse mapping from grids to particles (global synchronization)
  - I/O in sequential implementation
- Experimental Results
- Looking to the future:
  - Programming Tools and Auto-tuning

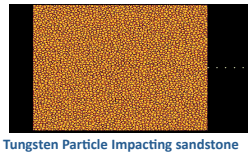
Schlumberger THE UNIVERSITY OF UTAH

Rigid, Soft Body and Fluid Simulations

- Breadth of applications
  - fluids and smoke in games, astrophysics simulation, oil exploration, and molecular dynamics
- MPM Part of Center for the Simulation of Accidental Fires and Explosions (C-SAFE) software environment



Compaction of a foam microstructure



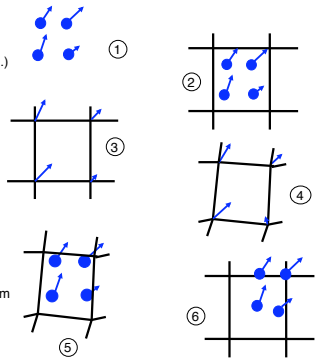
Tungsten Particle Impacting sandstone

Schlumberger THE UNIVERSITY OF UTAH

The Material Point Method (MPM)

1. Lagrangian material points carry all state data (position, velocity, stress, etc.)
2. Overlying mesh defined
3. Particle state projected to mesh, e.g.:
   $v_g = \sum_p S_{gp} m_p v_p / \sum_p S_{gp} m_p$
4. Conservation of momentum solved on mesh giving updated mesh velocity and (in principal) position.

   Stress at particles computed based on gradient of the mesh velocity.
5. Particle positions/velocities updated from mesh solution.
6. Discard deformed mesh. Define new mesh and repeat



Schlumberger THE UNIVERSITY OF UTAH

Approach

- Start with sequential library implementation of MPM and GIMP
  - And descriptions of parallel OpenMP and MPI implementations
- Profiling pinpointed key computations *(updateContribList and advance, >99%)*
- Two independent implementations (2-3 person teams)
- Some other aspects of mapping
  - Makes heavy use of C++ templates
  - Gnuplot used for visualization

Key Features of MPM and GIMP Computation

- Large amounts of data parallelism
- Particles mapped to discretized grid
  - Compute contribution of particles to grid nodes *(updateContribList)*
  - Compute <force, velocity, acceleration, stress> operations on grid nodes *(advance)*
- Each time step, the particles are moving
  - Compute stresses and recompute mapping
- Periodically, visualize or store results

Overview of Strategy for CUDA Implementation

- Partition particle data structure and mapping to grid across threads
- Build an inverse map from grid nodes to particles
  - Requires global synchronization
- Later phase partitions grid across threads
- Two implementations differ in strategy for this inverse map
  - V1: Sort grid nodes after every time step
  - V2: Replicate inverse map, using extra storage to avoid hotspots in memory (focus)

Global Synchronization for Inverse Map (CUDA Particle Project)

```
__device__ void addParticleToCell(int3 gridPos, uint
    index, uint* gridCounters, uint* gridCells)
{
  // calculate grid hash
  uint gridHash = calcGridHash(gridPos);

  // increment cell counter using atomics
  int counter = atomicAdd(&gridCounters[gridHash], 1);
  counter = min(counter, params.maxParticlesPerCell-1);

  // write particle index into this cell (uncoalesced!)
  gridCells[gridHash*params.maxParticlesPerCell +
    counter] = index;
}
```

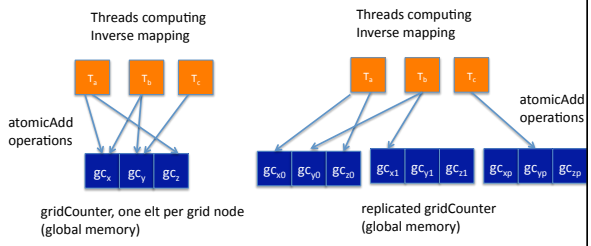
index refers to index of particle

gridPos represents grid cell in 3-d space

gridCells is data structure in global memory for the inverse mapping

What this does:
Builds up gridCells as array limited by max # particles per grid
atomicAdd gives how many particles have already been added to this cell

Optimized Version: Replicate gridCounters to avoid Contention



- Results of this optimization:
  - 2x speedup on *updateContribList*

Summary of Other Optimizations

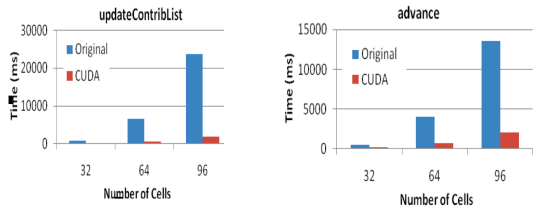
- Global memory coalescing
  - gridHash and gridCounters organization
  - Use of float2 and float4 data types
  - CUDA Visual Profiler pinpointed these!
- Maintain data on GPU across time steps
- Fuse multiple functions from sequential code into single, coarser grained GPU kernel
- Replace divides by multiples of inverse and cache

Experiment Details

- Architectures
  - Original = Intel Core2 Duo E8400 (3.00 GHz)
  - CUDA = nVIDIA GeForce 9600 GT (8 SMs)
- Input data set

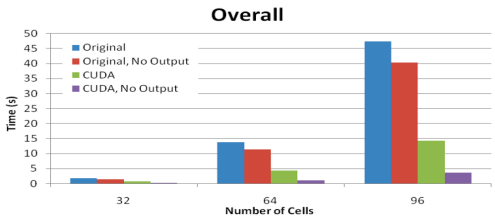
Cell	Grid Nodes	Particles
32	1,352	2,553
64	5,356	9,177
96	12,012	19,897

Results on Key Computations



- All results use 128 threads
- Speedups of 12.5x and 6.6x, respectively, over sequential implementation

Overall Speedup Results



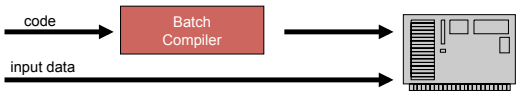
- No output, speedup of 10.7x
- With output, speedup only 3.3x
- Obvious future work: Open GL for visualization

Shifting Gears: Programmability and Auto-tuning

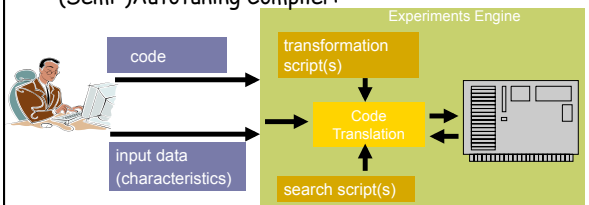
- Midterm extra credit question:
  - "If you could invest in tool research for GPUs, in what areas would you like to see progress?"
- Tools
  - Assistance with partitioning across threads/ blocks
  - Assistance with selecting numbers of threads/ blocks
  - Assistance with calculating indexing relative to thread/block partitioning

Auto-Tuning "Compiler"

Traditional view:



(Semi-)Autotuning Compiler:



Current Research Activity

- Automatically generate CUDA from sequential code and transformation script, with CUDAize(loop,TI,TJ,kernnm)
- Advantages of auto-tuning
  - Tradeoffs between large number of threads to hide latency and smaller number to increase reuse of data in registers
  - Detect ordering sensitivities that impact coalescing, bank conflicts, etc.
  - Evaluate alternative memory hierarchy optimizations
- Addresses challenges from earlier slide
  - Correct code generation, including indexing
  - Auto-tuning to select best thread/block partitioning
  - Memory hierarchy optimizations and data movement

Summary

- Three areas of improvement for MPM/GIMP
  - Used single precision, which may not always be sufficiently precise
  - Wanted more threads but constrained by register limits
  - OpenGL visualization of results
- Newer GPUs and straightforward extensions ameliorate these challenges
- Future work on programmability and auto-tuning

Schlumberger THE UNIVERSITY OF UTAH