OPTIMIZING THE LULESH STENCIL CODE USING CONCURRENT COLLECTIONS

WOLFHPC’15
Chenyang Liu
Milind Kulkarni
11-16-2015
Overview

• Background
• LULESH
• Concurrent Collections
• CnC LULESH
• Optimizations
• Experiments
• Conclusion
Scientific Programming…
LULESH Stencil

- **LULESH: Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics**
  - 2010 DARPA UHPC challenge problem
- 3D stencil
  - Hexahedral mesh with 2 centerings
  - Nodal Computations
  - Element Calculations
  - Highly data dependent algorithm
LULESH Algorithm

1. Calculate safe Timestep
2. Calculate Forces
3. Update Physical/Kinetic Properties
4. Calculate Thermodynamic Properties
LULESH Algorithm

1. Calculate safe Timestep
2. Update Physical/Kinetic Properties
3. Calculate Thermodynamic Properties
4. Calculate Forces

Diagram showing the flow of the algorithm with arrows connecting each step.
LULESH Algorithm

- Calculate safe Timestep
- Calculate Forces
- Update Physical/Kinetic Properties
- Calculate Thermodynamic Properties

**REDUCTION**
- Hourglass Partial Force
- Stress Partial Force
- Compute Total Force
- Compute Acceleration
- Compute Position
- Compute Velocity

**SCATTER/GATHER**
- Compute Volume
- Compute Characteristic Length
- Compute Gradients
- Compute Viscosity
- Compute Energy
- Compute Hydro

**SCATTER/GATHER**
- Compute Volume Derivative
- Compute Sound Speed
- Compute Courant
LULESH Challenges

- Programming the Stencil
  - Not a trivial problem
  - Complex dependencies
- Exploiting Parallelism
  - Managing communication patterns
- Data Layout and Tiling
  - Block sizing and cache
  - Optimizing memory performance
  - Element/Node size differences
Concurrent Collections (CnC)

- **Separation of Concerns Philosophy**
  - Scientists focus on Algorithm
  - Programmers focus on Performance
- **Programmability with Performance**
  - Programmer expresses computation, runtime exploits parallelism
  - Low level tuners for performance
- **Programming Model constraints:**
  - Must use CnC Collections
  - Extension of common programming languages
  - Explicit dependencies
The Collections

• CnC Model:
  • Step Collections
    • Stateless computation blocks
    • Dynamically instantiated using tags
  • Item Collections
    • Key/Value lookup
    • Dynamic Single Assignment
    • Explicit dependencies (Get/Put)
  • Tag Collections
    • Associated with every step collection
    • Dictates program control flow
CnC LULESH

- CnC simplifies:
  - Programming the Stencil
    - Domain expert can write the algorithmic program
  - High-level CnC specification
- Exploiting Parallelism
  - CnC handles the parallelism, maps to most platforms
- CnC Tuners (low-level)
- CnC does not solve:
  - Data Layout and Tiling
    - Optimizing block sizing
    - Achieve good granularity

```c
struct lulesh_context:public
  context<lulesh_context>
{
  // Step Collections
  step_collection<compute_dt>
    step_compute_dt;
  step_collection<reduce_force>
    step_reduce_force;
  ...

  // Item Collections
  // per node items
  item_collection<pair,vector>force;
  item_collection<pair,vertex>position;
  item_collection<pair,vector>velocity;
  // per element items
  ...

  // Tag Collections
  tag_collection<pair>iteration_node;
  tag_collection<pair>iteration_element;
  tag_collection<int>iteration;
  ...

  // Producer Dependencies
  step_compute_dt.consumes(dt);
  ...

  // Consumer Dependencies
  step_compute_dt.produces(dt);
  ...
```


Motivation

• Fully decomposed algorithm results in fine-grained parallelism
  • Performance suffers if steps lack sufficient computation
• Need to coarsen the computation
  • Modify CnC step/tag collections
  • Fusion: Combine multiple steps (graph level)
  • Tiling: Combine multiple step instances from multiple tags
• Challenge: Take the provided singleton, sequential, algorithmic stencil and produce a scalable parallel application
  • Perform high level transformations
    • Legality and detection
    • Program modifications
CnC computation space

Step Iteration Space

Tag Space
Step Fusion

Step Iteration Space
Tag Tiling

Step Iteration Space

Tag Space

Step1 Tile

Step2 → Step3 → Step4

Step2 → Step3 → Step4

Step2 → Step3 → Step4

Step2 → Step3 → Step4

Step2 → Step3 → Step4
Fusion Scenario

Step Iteration Space

Tag Space

Step 1

Step 2

Step 3

Step 4
Fusion Scenario

Step Iteration Space

Altered dependencies!!
Fusion and Tiling Legality

• **Step Fusion**
  • Step collections are prescribed by the same tag, all dependencies are within that tag space
  • Serialize steps, remove intermediate dependencies

• **Tag Tiling**
  • Step collection operates on multiple tags, performing the same work, independently
  • Coalesce computation from multiple tags
  • Opportunity to exploit reuse

• **Illegal Fusion**
  • Cannot fuse if a “get” depends on value “put” from a different tag

  Step1: `cnc.dataout.put(produced_data, my_index)`
  Step2: For(All neighbors) `{ cnc.dataout.get(consumed_data, neighbor_index) }`
Code changes

• Step Collections
  • Aggregate dependencies from fused routines
  • Coalesce work from multiple steps and tags
  • Maintain proper working set size and intermediate storage
  • Must maintain step-like behavior

• Tag Prescription
  • New smaller tag set

• Other Optimizations
  • Block size
  • Dependency re-use
  • Item Collection - coalescing
LULESH: Fused Algorithm
Experimental Results

- AMD Opteron 6176 SE system with four 12-core processors (48 cores total) running at 2.3 GHz.
- gcc 4.7, -o3, Intel CnC implementation

Experiments
  - Baseline
  - Fused-only
  - Tiled-only
  - Fused+Tiled Blocked (red)
  - Fused+Tiled Strided (blue)
Experimental Results cont.

### Table 1: Timing Results: 60³ Sized Mesh

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>32</th>
<th>48</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>53.180</td>
<td>52.900</td>
<td>71.852</td>
<td>108.531</td>
<td>110.515</td>
<td>110.572</td>
<td>119.466</td>
</tr>
<tr>
<td>Fusion Only</td>
<td>26.288</td>
<td>26.405</td>
<td>33.389</td>
<td>52.224</td>
<td>51.391</td>
<td>54.340</td>
<td>57.430</td>
</tr>
<tr>
<td>Tiling Only</td>
<td>12.67596</td>
<td>12.5693</td>
<td>6.677482</td>
<td>4.234</td>
<td>2.652</td>
<td>2.504</td>
<td>2.526</td>
</tr>
<tr>
<td>Blocked Fuse-Tile</td>
<td>10.749</td>
<td>10.699</td>
<td>5.110</td>
<td>2.883</td>
<td>1.845</td>
<td>1.557</td>
<td>1.768</td>
</tr>
</tbody>
</table>

### Speedup: Tiling vs Baseline Sequential

- Blue: Tiling Only
- Red: Tiling-Fusion Blocked
- Orange: Tiling-Fusion Strided

![Speedup Graph](image-url)
Tiling: Block Size

- Parameter: block size
  - Smaller size creates excess fine-grain parallelism
  - Larger size limits available parallelism
  - Tunable parameter

![Performance Impact: Block Size](chart.png)
Summary

• Contributions
  • Performance & scalability obtained with step fusion, tag tiling
  • Minimal underlying code and data layout changes
  • Automatic optimization in works

• Improving CnC
  • Tiling/Grain size is a huge problem for stencils
  • Tuners available, but do not operate at a high level

• Future Goals
  • High level collection tuning
  • Hierarchical CnC
Thank you!

Questions?