Outline

- Finish discussion of patterns
- Molecular dynamics example
  - Problem description
  - Steps to solution
  - Build data structures; Compute forces; Integrate for new position; Check global solution; Repeat
  - Finding concurrency
  - Data and data decomposition; reductions
  - Algorithm structure
  - Supporting structures

Patterns for Parallelizing Programs

4 Design Spaces

- Algorithm Expression
  - Finding Concurrency
    - Expose concurrent tasks
  - Algorithm Structure
    - Map tasks to processes to exploit parallel architecture
  - Supporting Structures
    - Code and data structuring patterns
  - Implementation Mechanisms
    - Low level mechanisms used to write parallel programs

ILP, DLP, and TLP in SW and HW

- ILP
  - OOO
  - Data Flow
  - VLIW
- DLP
  - SIMD
  - Vector
- TLP
  - Essentially multiple cores with multiple sequencers

ILP, DLP, and TLP and Supporting Patterns

<table>
<thead>
<tr>
<th>Task</th>
<th>Order and Parallel Composition</th>
<th>Dynamic Data Decomposition</th>
<th>Data Reduction</th>
<th>Pipelining</th>
<th>Load Balancing Schedules</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DLP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TLP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ILP, DLP, and TLP and Supporting Patterns

<table>
<thead>
<tr>
<th>Task</th>
<th>Order</th>
<th>Pipeline</th>
<th>Dynamic Scheduling</th>
<th>Dynamic Composition</th>
<th>Data Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILP</td>
<td>small</td>
<td>small</td>
<td>narrow</td>
<td>small</td>
<td>small</td>
</tr>
<tr>
<td>DLP</td>
<td>narrow</td>
<td>small</td>
<td>narrow</td>
<td>narrow</td>
<td>small</td>
</tr>
<tr>
<td>TLP</td>
<td>natural</td>
<td>natural</td>
<td>narrow</td>
<td>narrow</td>
<td>natural</td>
</tr>
</tbody>
</table>
Outline

• Molecular dynamics example
  - Problem description
  - Steps to solution
    • Build data structures; compute forces; integrate for new positions; check global solution; repeat
  - Finding concurrency
    • Scans; data decomposition; reductions
  - Algorithm structure
  - Supporting structures

Outline

• Molecular dynamics example
  - Problem description
  - Steps to solution
    • Build data structures; compute forces; integrate for new positions; check global solution; repeat
  - Finding concurrency
    • Scans; data decomposition; reductions
  - Algorithm structure
  - Supporting structures

Credits

• Parallel Scan slides courtesy David Kirk (NVIDIA) and Wen-Mei Hwu (UIUC)
  - Taken from ES493-AI taught at UIUC in Spring 2007
• Reduction slides courtesy Dr. Rodric Rabbah (IBM)
  - Taken from 6.189 IAP taught at MIT in 2007

GROMACS

• Highly optimized molecular-dynamics package
  - Popular code
  - Specifically tuned for protein folding
  - Hand optimized loops for SSE3 (and other extensions)

Gromacs Components

• Non-bonded forces
  - Water-water with cutoff
  - Protein-protein tabulated
  - Water-water tabulated
  - Protein-water tabulated
• Bonded forces
  - Angles
  - Dihedrals
  - Boundary conditions
  - Verlet integrator
  - Constraints
    • SHAKE
    • SPC
  - Other
    • Temperature-pressure coupling
    • Virial calculation
GROMACS Water-Water Force Calculation

- Non-bonded long-range interactions
  - Coulomb
  - Lennard-Jones
  - 234 operations per interaction

Water-water interaction ~75% of GROMACS run-time

GROMACS Uses Non-Trivial Neighbor-List Algorithm

- Full non-bonded force calculation is $\mathcal{O}(n^2)$
- GROMACS approximates with a cutoff
  - Molecules located more than $r_c$ apart do not interact
  - $\mathcal{O}(n r_c^3)$

Efficient algorithm leads to variable rate input streams
Parallel Prefix Sum (Scan)

• Definition:
The all-prefix-sums operation takes a binary associative operator ⊕ with identity I, and an array of n elements \([a_0, a_1, ..., a_{n-1}]\) and returns the ordered set \([I, a_0, (a_0 ⊕ a_1), ..., (a_0 ⊕ a_1 ⊕ ... ⊕ a_{n-2})]\).

• Example:
  if ⊕ is addition, then scan on the set \([3, 1, 7, 0, 4, 1, 6, 3]\) returns the set \([0, 3, 4, 11, 11, 15, 16, 22]\).

Applications of Scan

• Scan is a simple and useful parallel building block
  – Convert recurrences from sequential:
    for (j = 1; j < n; j++)
    out[j] = out[j-1] + f(j);
  – into parallel:
    for all (j) { temp[j] = f(j); }
    scan(out, temp);

• Useful for many parallel algorithms:
  – radix sort
  – quicksort
  – String comparison
  – lexical analysis
  – Building data structures
  – Etc.

Building Data Structures with Scans

Scan on a serial CPU

```c
void scan ( float* scanned, float* input, int length )
{
    scanned[0] = 0; for (int i = 1; i < length; ++i)
    {
        scanned[i] = input[i-1] + scanned[i-1];
    }
}
```

• Just add each element to the sum of the elements before it.
  • Trivial, but sequential.
  • Exactly n adds optimal.

A First-Attempt Parallel Scan Algorithm

1. Read input to shared memory. Set first element to zero and shift others right by one.

2. Iterate log(n) times: UEs stride to n:
   - Add pairs of elements stride elements apart.
   - Double stride at each iteration. (note must double buffer shared memory arrays)

Each UE reads one value from the input array in device memory into shared memory array T0. UE 0 writes 0 into shared memory array.

Iteration #1

Stride = 1

UE 0

<table>
<thead>
<tr>
<th>In</th>
<th>3</th>
<th>1</th>
<th>7</th>
<th>0</th>
<th>4</th>
<th>1</th>
<th>6</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>

Active UEs: stride to n-1 (n-stride UEs)

- UE i adds elements j and j-stride from T0 and writes result into shared memory buffer T1 (ping-pong).
A First-Attempt Parallel Scan Algorithm

1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.

2. Iterate log(n) times: UEs stride to n:
   - Add pairs of elements stride elements apart.
   - Double stride at each iteration. (note must double buffer shared mem arrays)

3. Write output.

What is wrong with our first-attempt parallel scan?

- **Work Efficient**: A parallel algorithm is work efficient if it does the same amount of work as an optimal sequential complexity.
- **Scan executes log(n) parallel iterations**
  - The steps do n, n/2, n/4, ..., n/2 adds each
  - Total adds n * (log(n) - 1) + 1 = O(n log(n)) work
- **This scan algorithm is NOT work efficient**
  - Sequential scan algorithm does n adds
  - A factor of log(n) hurts 20x for 10^6 elements!

Improving Efficiency

- A common parallel algorithm pattern: Balanced Trees
  - Build a balanced binary tree on the input data and sweep it to and from the root.
  - Tree is not an actual data structure, but a concept to determine what each UE does at each step.

  - For scan:
    - Traverse down from leaves to root building partial sums at internal nodes in the tree.
    - Root holds sum of all leaves.
    - Traverse back up the tree building the scan from the partial sums.

Build the Sum Tree

- Assume array is already in shared memory.

**Build the Sum Tree**

Stride 1

- Iteration 1, n/2 UEs
  
  \[ T \begin{array}{cccccccc} 
  3 & 1 & 7 & 0 & 4 & 1 & 6 & 3 
  \end{array} \]

- Each corresponds to a single UE.

Stride 2

- Iteration 2, n/4 UEs
  
  \[ T \begin{array}{cccccccc} 
  3 & 4 & 7 & 1 & 1 & 1 & 4 & 5 & 6 \end{array} \]

- Each corresponds to a single UE.

**Build the Sum Tree**

Stride 1

- Iteration 1, n/2 UEs
  
  \[ T \begin{array}{cccccccc} 
  3 & 1 & 7 & 0 & 4 & 1 & 6 & 3 
  \end{array} \]

- Each corresponds to a single UE.

Stride 2

- Iteration 2, n/4 UEs
  
  \[ T \begin{array}{cccccccc} 
  3 & 4 & 7 & 1 & 1 & 4 & 5 & 6 \end{array} \]

- Each corresponds to a single UE.

**Build Scan From Partial Sums**

- Iteration 1
  
  \[ T \begin{array}{cccc} 
  3 & 4 & 7 & 11 
  \end{array} \]

- Iteration 2
  
  \[ T \begin{array}{cccc} 
  3 & 4 & 7 & 11 
  \end{array} \]

- Iteration 3
  
  \[ T \begin{array}{cccc} 
  3 & 4 & 7 & 11 
  \end{array} \]

**Zero the Last Element**

- We now have an array of partial sums. Since this is an exclusive scan, set the last element to zero. It will propagate back to the first element.

- Iteration 1
  
  \[ T \begin{array}{cccc} 
  3 & 4 & 7 & 11 
  \end{array} \]

**Build Scan From Partial Sums**

- Iteration 1
  
  \[ T \begin{array}{cccc} 
  3 & 4 & 7 & 11 
  \end{array} \]
Build Scan From Partial Sums

Done! We now have a completed scan that we can write out to device memory.

Serial Reduction

- When reduction operator is not associative
- Usually followed by a broadcast of result

Tree-based Reduction

- n steps for 2^n units of execution
- When reduction operator is associative
- Especially attractive when only one task needs result

Reductions

- Many to one
- Many to many
  - Simply multiple reductions
  - Also known as scatter-add and subset of parallel prefix sums
- Use
  - Histograms
  - Superposition
    - Physical properties

Recursive-doubling Reduction

- n steps for 2^n units of execution
- If all units of execution need the result of the reduction
Recursive-doubling Reduction

- Better than tree-based approach with broadcast
  - Each unit of execution has a copy of the reduced value at the end of n steps
  - In tree-based approach with broadcast:
    - Reduction takes n steps
    - Broadcast cannot begin until reduction is complete
    - Broadcast can take n steps (architecture dependent)

Other Examples

- More patterns
  - Reductions
  - Scans
  - Building a data structure
- More examples
  - Search
  - Sort
  - FFT as divide and conquer
  - Structured meshes and grids
  - Sparse algebra
  - Unstructured meshes and graphs
  - Trees
  - Collections
    - Particles
    - Rays