EE382V: Principles in Computer Architecture
Parallelism and Locality
Fall 2008

Lecture 9 – Example of Using Parallel Constructs

Mattan Erez

UT ECE

The University of Texas at Austin

Outline

- Finish discussion of patterns
- 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

Patterns for Parallelizing Programs

4 Design Spaces

Algorithm Expression

- Finding Concurrency
  - Expose concurrent tasks
- Algorithm Structure
  - Map tasks to processes to exploit parallel architecture

Software Construction

- Supporting Structures
  - Code and data structuring patterns
- Implementation Mechanisms
  - Low level mechanisms used to write parallel programs

Patterns for Parallel Programming. Mattson, Sanders, and Massingill (2005).

ILP, DLP, and TLP in SW and HW

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

- ILP
  - Within straight-line code
- DLP
  - Parallel loops
  - Tasks operating on disjoint data
    - No dependencies within parallelism phase
- TLP
  - All of DLP +
  - Producer-consumer chains

ILP, DLP, and TLP and Supporting Patterns

	Task parallelism	Divide and conquer	Geometric decomposition	Recursive data	Pipeline	Event-based coordination
ILP						
DLP						
TLP						

ILP, DLP, and TLP and Supporting Patterns

	Task parallelism	Divide and conquer	Geometric decomposition	Recursive data	Pipeline	Event-based coordination
ILP	inline / unroll	inline	unroll	inline	inline / unroll	inline
DLP	natural or local-conditions	after enough divisions	natural	after enough branches	difficult	local- conditions
TLP	natural	natural	natural	natural	natural	natural

ILP, DLP, and TLP and Implementation Patterns

	SPMD	Loop Parallelism	Mater/Worker	Fork/Join
ILP				
DLP				
TLP				

ILP, DLP, and TLP and Implementation Patterns

	SPMD	Loop Parallelism	Mater/Worker	Fork/Join
ILP	pipeline	unroll	inline	inline
DLP	natural or local-conditional	natural	local-conditional	after enough divisions + local-conditional
TLP	natural	natural	natural	natural

Outline

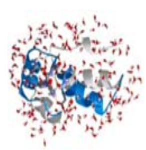
- Molecular dynamics example
  - Problem description
  - Steps to solution
    - Build data structures; Compute forces; Integrate for new; positions; Check global solution; Repeat
  - Finding concurrency
    - Scans; data decomposition; reductions
  - Algorithm structure
  - Supporting structures

Credits

- Parallel Scan slides courtesy David Kirk (NVIDIA) and Wen-Mei Hwu (UIUC)
  - Taken from EE493-AI taught at UIUC in Sprig 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)

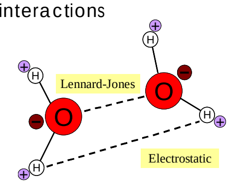


Gromacs Components

- 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
  - SETTLE
- Other
  - Temperature–pressure coupling
  - Virial calculation

GROMACS Water-Water Force Calculation

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

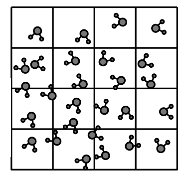


$$V_{nb} = \sum_{i,j}\left[\frac{1}{4\pi\epsilon_0}\frac{q_i q_j}{r_{ij}} + \left(\frac{C_{12}}{r_{ij}^{12}} - \frac{C_6}{r_{ij}^{6}}\right)\right]$$

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

GROMACS Uses Non-Trivial Neighbor-List Algorithm

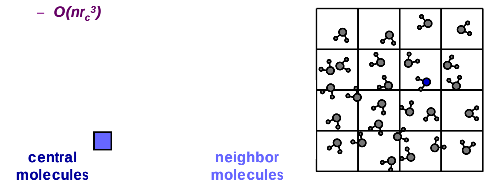
- Full non-bonded force calculation is ***o(n²)***
- GROMACS approximates with a cutoff
  - Molecules located more than r_c apart do not interact
  - ***O(nr_c^3)***



Efficient algorithm leads to variable rate input streams

GROMACS Uses Non-Trivial Neighbor-List Algorithm

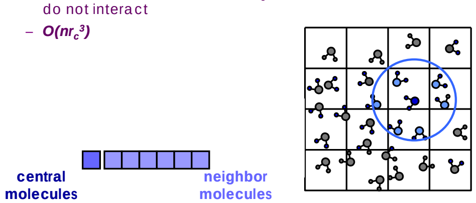
- Full non-bonded force calculation is ***o(n²)***
- GROMACS approximates with a cutoff
  - Molecules located more than r_c apart do not interact
  - ***O(nr_c^3)***



Efficient algorithm leads to variable rate input streams

GROMACS Uses Non-Trivial Neighbor-List Algorithm

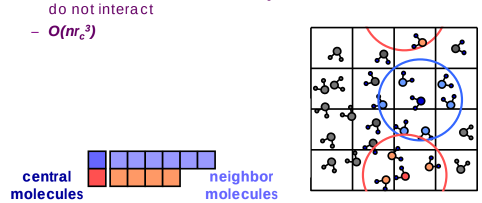
- Full non-bonded force calculation is ***o(n²)***
- GROMACS approximates with a cutoff
  - Molecules located more than r_c apart do not interact
  - ***O(nr_c^3)***



Efficient algorithm leads to variable rate input streams

GROMACS Uses Non-Trivial Neighbor-List Algorithm

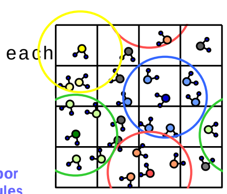
- Full non-bonded force calculation is ***o(n²)***
- GROMACS approximates with a cutoff
  - Molecules located more than r_c apart do not interact
  - ***O(nr_c^3)***

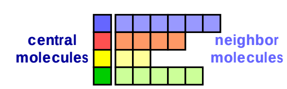


Efficient algorithm leads to variable rate input streams

GROMACS Uses Non-Trivial Neighbor-List Algorithm

- Full non-bonded force calculation is ***o(n²)***
- GROMACS approximates with a cutoff
  - Molecules located more than r_c apart do not interact
  - ***O(nr_c^3)***
- Separate neighbor-list for each molecule
  - Neighbor-lists have variable number of elements





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

  $[a_0, a_1, \ldots, a_{n-1}]$

  and returns the ordered set

  $[I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})]$.

- 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]

Exclusive scan: last input element is not included in the result

(From Blelloch, 1990, "Prefix Sums and Their Applications)

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:

```
forall(j) { temp[j] = f(j) };
scan(out, temp);
```

- Useful for many parallel algorithms:
  - radix sort
  - quicksort
  - String comparison
  - Lexical analysis
  - Stream compaction
  - Polynomial evaluation
  - Solving recurrences
  - Tree operations
  - **Building data structures**
  - Etc.

Building Data Structures with Scans

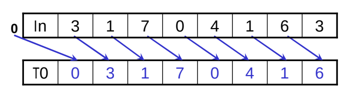
- Fun on the board

Scan on a serial CPU

```
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

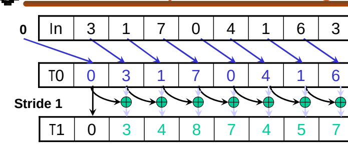
A First-Attempt Parallel Scan Algorithm



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.

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

A First-Attempt Parallel Scan Algorithm

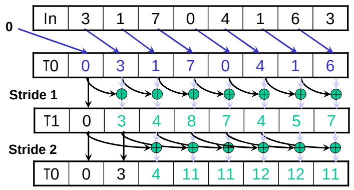


1. (previous slide)
2. Iterate log(n) times: UEs *stride* to *n*: Add pairs of elements *stride* elements apart. Double *stride* at each iteration. (note must double buffer shared mem arrays)

Iteration #1
Stride = 1

- Active UEs: *stride* to *n*-1 (*n-stride* UEs)
- UE *j* adds elements *j* and *j-stride* from T0 and writes result into shared memory buffer T1 (ping-pong)

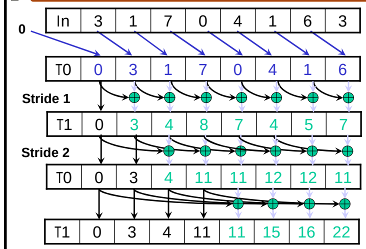
A First-Attempt Parallel Scan Algorithm



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)

Iteration #2
Stride = 2

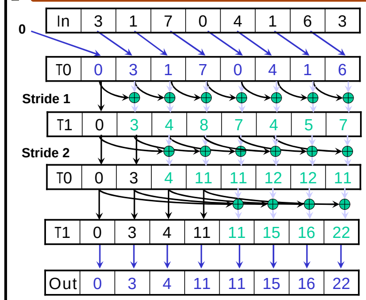
A First-Attempt Parallel Scan Algorithm



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)

Iteration #3
Stride = 4

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-1, 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:

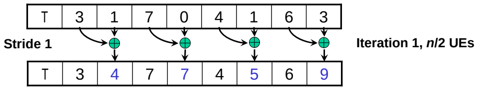
  Balanced Trees
  - Build a balanced binary tree on the input data and sweep it to and from the root
  - Tree is not an actual data structure, but a concept to determine what each UE does at each step
- For scan:
  - Traverse down from leaves to root building partial sums at internal nodes in the tree
    - Root holds sum of all leaves
  - Traverse back up the tree building the scan from the partial sums

Build the Sum Tree

T	3	1	7	0	4	1	6	3

Assume array is already in shared memory

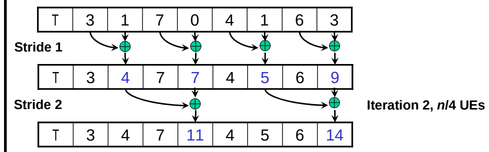
Build the Sum Tree



Each $\oplus$ corresponds to a single UE.

Iterate log(n) times. Each UE adds value *stride* elements away to its own value

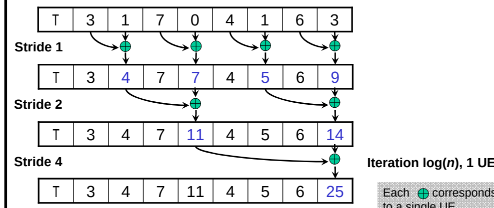
Build the Sum Tree



Each $\oplus$ corresponds to a single UE.

Iterate log(n) times. Each UE adds value *stride* elements away to its own value

Build the Sum Tree



Each $\oplus$ corresponds to a single UE.

Iterate log(n) times. Each UE adds value *stride* elements away to its own value.

Note that this algorithm operates in-place: no need for double buffering

Zero the Last Element

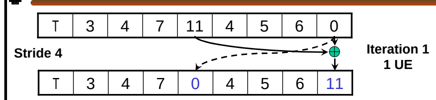
T	3	4	7	11	4	5	6	0

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.

Build Scan From Partial Sums

T	3	4	7	11	4	5	6	0

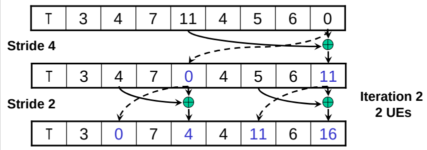
Build Scan From Partial Sums



Each $\oplus$ corresponds to a single UE.

Iterate log(n) times. Each UE adds value *stride* elements away to its own value, and sets the value *stride* elements away to its own *previous* value.

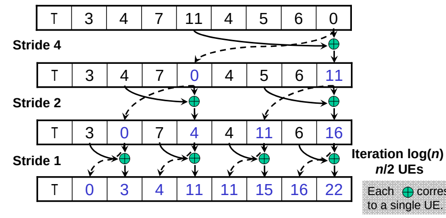
Build Scan From Partial Sums



Each ⊕ corresponds to a single UE.

Iterate log(n) times. Each UE adds value *stride* elements away to its own value, and sets the value *stride* elements away to its own *previous* value.

Build Scan From Partial Sums



Each ⊕ corresponds to a single UE.

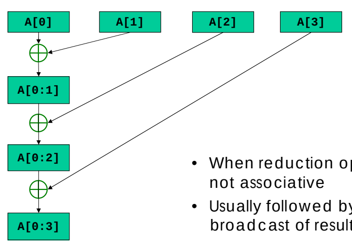
Done! We now have a completed scan that we can write out to device memory.

Total steps: 2 * log(n).
Total work: 2 * (n-1) adds = $O(n)$ **Work Efficient!**

Reductions

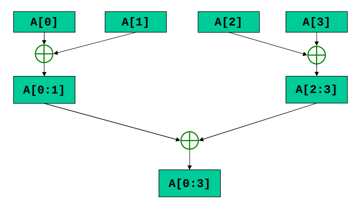
- Many to one
- Many to many
  - Simply multiple reductions
    - Also known as scatter-add and subset of parallel prefix sums
- Use
  - Histograms
  - Superposition
    - Physical properties

Serial Reduction



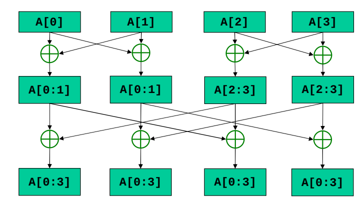
- When reduction operator is not associative
- Usually followed by a broadcast of result

Tree-based Reduction



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

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 units 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