Active Data Structures on GPGPUs

John O'Donnell
Cordelia Hall
Stuart Monro
University of Glasgow

UCPHC
August 2013, Aachen

- What is an active data structure?
- Two examples
- What platforms are suitable?
- Why GPGPUs are imperfect but wonderful

What is an active data structure?

- Access it through higher level operations, don’t operate directly on elements
  - Search for a value in a set of tuples
  - Insert into a binary search tree
- Similar to an API, but
  - Use data parallelism to make the operations fast

Examples in processors

- Set associative cache controller
- Translation lookaside buffer

- Search algorithms that are critical to processor performance
  - Performs searching in one clock cycle
  - A comparison circuit for each memory word
  - A simple tree circuit to determine whether there is a match

A canonical representation

- Our family of active data structures shares a uniform approach
  - A vector of cells; each cell is a tuple of fields
  - A family of combinators; higher order functions that express regular patterns of computation and communication
- The algorithm comprises a set of instructions
  - Each instruction is a composition of combinators

The combinator families

- Map expresses an iteration across an array

  map f [] = []
  map f (x:xs) = f x : map f xs

- Fold combines cells to produce a singleton

  tfold f g (Cell x) = f x
  tfold f g (Node x y) = Node (g x y)

- Scan produces a vector of partial folds

  scanl (+) a [x0, x1, x2] =
  [a, a+x0, (a+x0)+x1, ((a+x0)+x1)+x2, …]

Virtual tree architecture

- The family of maps can be implemented in parallel—no data dependencies between the elements. O(1) time
- The folds can be implemented in parallel on a tree architecture. O(log n) time
- The scans can also run in log time, using recursive doubling or the tree architecture

Implementation

- The virtual tree architecture gives a portable, high level basis for the algorithms
- Several ways to implement it on real hardware
  - Coarse grain: multicore
  - Intermediate grain: GPU
  - Fine grain: FPGA, VLSI

Case study: Selection

- Take a set of unordered data xs=[x0, …, xn]
  - select k xs = the k'th largest (smallest) value
- Easy if k is close to 0 or n; harder as k approaches n/2
- Simple but inefficient: sort xs and index by k

ADS approach to selection

- Don’t represent the relative position of an element by its physical location
- Instead, associate an index interval (lo,hi) with each element
- (xi, lo, hi) means that if the array were sorted, then i <= lo <= hi

Flexible representation

- We can represent completely known information: (x, 9, 9) indicates that x is the 9’th largest element
- Partial information: (x, 12, 34)
- No information at all: (x, 0, n)

Basic instruction: Improve j

- Effect is to find every cell where lo <= j <= hi and refine its index interval—in parallel
  1. Locate cells where j is in the interval (map)
  2. Choose a “splitter” (fold)
  3. Count how many elements are smaller (fold)
  4. Refine every index interval (map)
- Total time is 4 cycles, O(1) time
- This improves many cells, all in O(1) time

Select k

While no cell has (x, k, k) — fold: 1 cycle

  Improve k — 4 cycles

- Each iteration is O(1) time
- Number of iterations is log n (worst case)
- Side effect: each selection refines many cells, so future selections are faster
- Can sort by repeated selection

Extensible sparse functional arrays

- Imperative arrays have two operations
  - Lookup: a[i]
  - Update: a[i] = e
- ESF arrays provide
  - empty – a constant array with no elements
  - lookup a i
  - update a i e

Update gives a new array a' where a'[i] = e

But the old array a is unchanged

Properties of ESF arrays

- An array is persistent
  - Update creates a new array, but doesn't disturb the old one
  - Useful in a pure functional language:
    - No side effects
- No need to allocate array
  - Every array is built by a sequence of updates, starting from empty

Hard to implement efficiently!

- Naïve solution 1
  - On update, make a new copy and modify at the index
  - Lookup is O(1), but update takes O(n) time and space for array of size n
- Naïve solution 2
  - Build a binary tree of updates
  - Now update is fast but lookup is slow
- Clever algorithms with rebalanced trees
  - Rebalancing has high overhead; both lookup and update are slow

Solution with active data structure

- Every operation takes a fixed number of instructions—worst case time O(1)
- This is faster than the long-believed lower bound of O(log n)
- Solution runs on a smart memory
  - Same circuit complexity as a RAM
  - Same cycle time complexity as a RAM

Change of perspective

- A conventional array “knows” the locations of its elements, found by address arithmetic
- This would be hopeless for ESFA
  - We must share each element with every array that contains it
  - Yet we can’t use a linked data structure (tree) to capture the sharing (no time to follow pointers)
- Idea 1: An array is just a code; each element knows the set of arrays that contain it

Inclusion sets

- An array code c is a natural number that identifies the array
- An element contains an inclusion set: codes of all the arrays this element belongs to
- Idea 2: we can represent inclusion sets with an index interval (lo, hi) provided that we readjust array codes

For details, see references…

- The instruction set supports extensibility, sparseness, searching, inverse searching
- Every instruction takes O(1) time in the worst case, with no restrictions on usage
- When an array is deleted, the memory manager reclaims every inaccessible cell in O(1) time. Small-constant time garbage collector!

Granularity

- Extremely coarse grain
  - Multicore with RAM
- Intermediate
  - GPU: typical 512 processor cores, shared memory
- Fine grain
  - FPGA with configurable logic blocks and local memories
- VLSI
  - Can mix logic and memory freely

Tradeoffs

- A VLSI chip would be fastest
  - But high cost
- FPGA is fast and cheaper
  - But interfacing is hard
- GPUs have ~1000 processing elements
  - Much less parallelism than FPGA
  - But ubiquitous and interfacing is easy

Challenges

- Five memory spaces
  - Shared memory is fast but limited access
  - Global memory is flexible but 100 times slower
- Blocks
  - Threads are organised into blocks
  - Can communicate and synchronise with a block, but not directly across blocks
- Coordination
  - Kernel is an SPMD style algorithm that runs in threads

What a GPU needs to do

- Keep persistent data in fast shared memory
- Long-running kernel
- Synchronise threads (within blocks and across blocks)
- Communicate across blocks
- Protocol for communication between CPU and GPU
- Combinators for computation and communication

Kernel implements an instruction set

- Request from CPU
  - Opcode, fixed number of operands
  - One opcode is “initialise the GPU”
  - Another is “terminate the kernel”
- Kernel
  - Receives and decodes request
  - Executes
  - Responds with a tuple of results

Barrier synchronisation

- Within a block, there is a CUDA primitive sync_threads
- Across blocks, we use the lock-free algorithm by Xiao and Feng (IPDPS 2010)

Combinators

- We have improved implementations of the fold and scan combinators
  - They work with each other (consistent representation)
  - They are higher order (but we have to generate first order CUDA code)
  - They are correct ☺

Testing

- The virtual tree architecture is synchronous, and easy to debug
- But GPU code is prone to deadlock and race conditions
- We have an elaborate test harness with automatically generated test data
- The result of every operation is checked for correctness by the CPU
- Have run ~30M operations without error

Results

- The GPU implementation of ESFA is 480 times faster than CPU implementation
  - The GPU timing includes everything: communication, coordination, checking every result for correctness
  - This is a real speedup of 480 (wall clock time)

For more details

- See the paper
- Further information on the web
  - www.dcs.gla.ac.uk/~jtod/research/gpu
  - www.dcs.gla.ac.uk/~jtod/research/esfa
  - Includes the source code, including test generation. Working C+Cuda code, and some Haskell

Conclusion
Why GPUs are imperfect but wonderful

- The downside
  - GPUs are coarse grain compared to VLSI, and offer less parallelism
  - GPU architecture is not specified; formal methods are impossible!
- But they are wonderful for active data structures
  - High performance
  - Easy to install and interface to CPU