General Transformations for GPU Execution of Tree Traversals

Michael Goldfarb*, Youngjoon Jo**, Milind Kulkarni
School of Electrical and Computer Engineering

* Now at Qualcomm; ** Now at Google
GPU execution of irregular programs

- GPUs offer promise of massive, energy-efficient parallelism
- Much success in mapping *regular* applications to GPUs
  - Regular memory accesses, predictable computation
- Much less success in mapping *irregular* applications
  - Pointer-based data structures
  - Unpredictable, input-dependent computation and memory accesses
Tree traversal algorithms

• Many irregular algorithms are built around tree-traversal
  • Barnes-Hut
  • Nearest-neighbor
  • 2-point correlation

• Numerous papers describing how to map tree traversal algorithms to GPUs
Point correlation

- Data mining algorithm
- Goal: given a set of $N$ points in $k$ dimensions and a point $p$, find all points within a radius $r$ of $p$
- Naïve approach: compare all $N$ points with $p$
- Better approach: build $kd$-tree over points, traverse tree for point $p$, prune subtrees that are far from $p$
Point correlation

- Data mining algorithm
- Goal: given a set of $N$ points in $k$ dimensions and a point $p$, find all points within a radius $r$ of $p$
- Naïve approach: compare all $N$ points with $p$
- Better approach: build $kd$-tree over points, traverse tree for point $p$, prune subtrees that are far from $p$
Point correlation

- Data mining algorithm
- Goal: given a set of $N$ points in $k$ dimensions and a point $p$, find all points within a radius $r$ of $p$
- Naïve approach: compare all $N$ points with $p$
- Better approach: build $kd$-tree over points, traverse tree for point $p$, prune subtrees that are far from $p$
Point correlation
Point correlation
Point correlation
Point correlation
Point correlation
Point correlation
Point correlation
Point correlation
Point correlation
Point correlation
Point correlation
Point correlation
Point correlation
Point correlation

```c
KDCell root = /* build kdtree */;
Set<Point> ps;
double radius;

foreach Point p in ps {
    recurse(p, root, radius);
}
...
void recurse(Point p, KDCell node, double r) {
    if (tooFar(p, node, r)) return;
    if (node.isLeaf() && (dist(node.point, p) < r))
        p.correlated++;
    else {
        recurse(p, node.left, r);
        recurse(p, node.right, r);
    }
}
```
Basic pattern

TreeNode root;
Set<Point> ps;

foreach Point p in ps {
    recurse(p, root, ...);
}

... recurse(Point p, KDCell node, ...) {
    if (truncate?(p, node, ...))
        { ... }
    recurse(p, node.child1, ...);
    recurse(p, node.child2, ...);
    ...
}
Basic pattern

TreeNode root;
Set<Point> ps;

foreach Point p in ps {
    recurse(p, root, ...);
}

... 
recurse(Point p, KDCell node, ...) {
    if (truncate?(p, node, ...))
    {
        ...
        recurse(p, node.child1, ...);
        recurse(p, node.child2, ...);
        ...
    }
}
Basic pattern

TreeNode root;
Set<Point> ps;

foreach Point p in ps {
    recurse(p, root, ...);
}
...

recurse(Point p, KDCell node, ...) {
    if (truncate?(p, node, ...))
        { ... }  
    recurse(p, node.child1, ...);
    recurse(p, node.child2, ...);
    ... 
}
Basic pattern

TreeNode root;
Set<Point> ps;

foreach Point p in ps {
    recurse(p, root, ...);
}

... recurse(Point p, KDCell node, ...) {
    if (truncate?(p, node, ...))
        { ... }
    recurse(p, node.child1, ...);
    recurse(p, node.child2, ...);
    ...
}
Basic pattern

TreeNode root;
Set<Point> ps;

foreach Point p in ps {
    recurse(p, root, ...);
}

... recurse(Point p, KDCell node, ...) {
    if (truncate?(p, node, ...)) {
        ...
    }
    recurse(p, node.child1, ...);
    recurse(p, node.child2, ...);
    ...
}

Lots of parallelism!
What’s the problem?

- GPUs add high overhead for recursion
- GPUs work best when memory accesses are regular and strided, but irregular algorithms have unpredictable memory accesses
- Status quo: ad hoc solutions
- New algorithm? New GPU techniques!
What’s the problem?

- GPUs add high overhead for recursion.
- GPUs work best when memory accesses are regular and strided, but irregular algorithms have unpredictable memory accesses.
- Status quo: ad hoc solutions.
- New algorithm? New GPU techniques!

Want generally applicable techniques for mapping irregular applications to GPUs.
Contributions

• Two general techniques for mapping tree-traversals to GPUs
  • **Autoropes**: eliminates recursion overhead
  • **Lockstepping**: promotes memory coalescing
• Compiler pass to automatically apply techniques to recursive tree-traversal code
• Significant GPU speedups on 5 tree-traversal algorithms
Naïve GPU implementation

- **Warp**-based *SIMT* (single-instruction, multiple-thread) execution
- 32 points put in a single warp
- Warp traverses tree
- All points in warp must execute same instruction
- If points *diverge*, some points sit idle while other threads execute
Naïve GPU implementation
Naïve GPU implementation
Naïve GPU implementation
Naïve GPU implementation
Naïve GPU implementation
Naïve GPU implementation
Naïve GPU implementation
Naïve GPU implementation
Naïve GPU implementation
Naïve GPU implementation
Naïve GPU implementation
Naïve GPU implementation
Naïve GPU implementation
Naïve GPU implementation
Naïve GPU implementation
Naïve GPU implementation
Naïve GPU implementation
Naïve GPU implementation
Lots of accesses to tree

- Many accesses just moving up the tree in order to later move down again
- Lots of function stack manipulation
- Trees are very large, cannot be stored in GPU’s fast memory
- Want to minimize accesses to tree
How to avoid extra accesses to tree?

- Typical technique: *ropes*
- Pointers in each tree node that let a traversal jump to the next part of the tree
- Effectively linearizes traversal
How to avoid extra accesses to tree?

- Typical technique: *ropes*
- Pointers in each tree node that let a traversal jump to the next part of the tree
- Effectively linearizes traversal
How to avoid extra accesses to tree?

- Typical technique: *ropes*
- Pointers in each tree node that let a traversal jump to the next part of the tree
- Effectively linearizes traversal
How to avoid extra accesses to tree?

- Typical technique: *ropes*
- Pointers in each tree node that let a traversal jump to the next part of the tree
- Effectively linearizes traversal
How to avoid extra accesses to tree?

- Installing ropes into a tree requires complex, application-specific preprocessing.
- Using ropes correctly during execution requires complex, application-specific logic.
Autoropes

• General technique for “linearizing” tree traversal for *arbitrary traversal algorithms*

• Achieve generality, simplicity and space-efficiency at the cost of overhead

• Key insight: *recursive tree algorithms are just depth-first traversals of a tree; can transform into iterative algorithm*
Autoropes
Autoropes
Autoropes
Autoropes
Autoropes
Autoropes

Rope stack

Thursday, November 21, 13
Autoropes
Autoropes

- Ropes stored on *rope stack* instead of in tree
- No application-specific code to use ropes
- Ropes instantiated *dynamically*
- No preprocessing required
- Same access patterns as with manual ropes
- Extra pushes and pops on rope stack add some overhead
void recurse(Point p, KDCell node, double r) {
    if (tooFar(p, node, r)) return;
    if (node.isLeaf() && (dist(node.point, p) < r))
        p.correlated++;
    else {
        recurse(p, node.left, r);
        recurse(p, node.right, r);
    }
}
ropeStack.push(root);
while (!ropeStack.isEmpty()) {
    node = ropeStack.pop();
    if (tooFar(p, node, r)) continue;
    if (node.isLeaf() && (dist(node.point, p) < r))
        p.correlated++;
    else {
        ropeStack.push(node.right);
        ropeStack.push(node.left);
    }
}

See paper for details of how to transform more complex code
Unintended consequence

• Recursive calls naturally lead to thread divergence

• If some threads make recursive calls, other threads wait until calls return

• Does not happen for iterative code

• All threads reconverge at beginning of loop

ropeStack.push(root);
while (!ropeStack.isEmpty()) {
    node = ropeStack.pop();
    if (tooFar(p, node, r))
        continue;
    if (...)
        p.correlated++;
    else {
        ropeStack.push(node.right);
        ropeStack.push(node.left);
    }
}
Unintended consequence

- Recursive calls naturally lead to thread divergence
- If some threads make recursive calls, other threads wait until calls return
- Does not happen for iterative code
- All threads reconverge at beginning of loop

```java
ropeStack.push(root);
while (!ropeStack.isEmpty()) {
    node = ropeStack.pop();
    if (tooFar(p, node, r))
        continue;
    if (...)
        p.correlated++;
    else {
        ropeStack.push(node.right);
        ropeStack.push(node.left);
    }
}
```
Autoropones on GPU

Diagram of a tree structure with nodes labeled A, B, C, D, E, F, G, H, I, J, and K.
Autoropes on GPU
Autoropes on GPU
Autoropes on GPU
Autoropes on GPU
Autoropes on GPU
Autoropes on GPU
Autoropes on GPU

Threads no longer diverge in execution
But do diverge in tree!
Thread divergence vs. memory coalescing

• Memory accesses on GPU only well behaved if accesses by all threads in warp can be coalesced

• Same memory or strided access

• Bad memory behavior of autoropes outweighs lack of thread divergence

• Goal: benefits of autoropes while maintaining memory coalescing
Lockstepping

- Essentially, force GPU to let threads diverge
  - If *any* thread in a warp wants to visit a node’s children, *all* threads in a warp visit the child
  - Threads that are “dragged along” are programmatically masked out
- Warp execution takes longer (proportional to union of threads’ traversals, rather than longest traversal), but improved memory performance makes up for it
- Automatically implemented during autoropes compiler pass
Dynamic lockstepping

• Some algorithms allow different traversal orders
  • Some points visit left child before right, and others visit right before left
  • Optimization reduces traversal size
  • Inherently bad memory access patterns
Dynamic lockstepping
Dynamic lockstepping
Dynamic lockstepping
Dynamic lockstepping
Dynamic lockstepping
Dynamic lockstepping

- Dynamic lockstepping allows all points in a warp to “vote” on which traversal order to use
- Maintains memory coalescing
- Some points do more work than in original algorithm
- Tradeoff can still be worth it!
Engineering details

• Transform point data from array of structures format to structure of arrays

• Use analysis from [PACT 2013] to prove safety and transform automatically

• Copy tree data to GPU in linearized fashion

• Lay out fields of tree and point according to use (more commonly-accessed fields placed in shared memory)

• Interleave rope stacks for points in warp to allow strided access
Results

- Two platforms
  - GPU platform: NVIDIA Tesla C2070 (6GB global memory, 14 SMs)
  - CPU platform: 32-core, 2.3 GHz Opteron
- Five benchmarks
  - Barnes-Hut, Point correlation, Nearest neighbor, k-Nearest neighbor, Vantage point trees
- Multiple inputs per benchmark
- Used sorted and unsorted points
High-level takeaways

• Autoropes+lockstep always faster than simple recursive GPU implementation (up to 14x faster)

• For most benchmarks/inputs, best GPU implementation faster than CPU implementation up to 16 threads

• Speedups comparable to hand-written implementations
Barnes-Hut

CPU Performance vs. GPU

# Threads

Random  Plummer
Point correlation
Nearest-neighbor
Conclusions

• Mapping irregular applications to GPUs is very difficult

• Developed two general techniques, autoropes and lockstepping, that can achieve significant speedup on GPU

• vs. baseline GPU code and CPU implementations

• Automatic approaches competitive with previous hand-written implementations
General Transformations for GPU Execution of Tree Traversals

Michael Goldfarb*, Youngjoon Jo**, Milind Kulkarni
School of Electrical and Computer Engineering

* Now at Qualcomm; ** Now at Google