ABSTRACT
With the advent of programmer-friendly GPU computing environments, there has been much interest in offloading workloads that can exploit the high degree of parallelism available on modern GPUs. Exploiting this parallelism and optimizing for the GPU memory hierarchy is well-understood for regular applications that operate on dense data structures such as arrays and matrices. However, there has been significantly less work in the area of irregular algorithms and even less so when pointer-based dynamic data structures are involved. Recently, irregular algorithms such as Barnes-Hut and kd-tree traversals have been implemented on GPUs, yielding significant performance gains over CPU implementations. However, the implementations often rely on exploiting application-specific semantics to get acceptable performance. We argue that there are general-purpose techniques for implementing irregular algorithms on GPUs that exploit similarities in algorithmic structure rather than application-specific knowledge. We demonstrate these techniques on several tree traversal algorithms, achieving speedups of up to 38× over 32-thread CPU versions.

Categories and Subject Descriptors
D.3.4 [Processors]: Compilers, Optimization

General Terms
Languages, Algorithms, Performance

Keywords
vectorization, tree traversals, GPU, irregular programs

1 Introduction
With the increasing capabilities of graphics processing units (GPUs), and their noticeable performance per watt advantages over CPUs, there has been significant interest in mapping various types of computational problems to GPUs. The main success stories of GPU parallelization are regular applications, such as dense linear algebra programs, which are characterized by predictable, structured accesses to memory, and large amounts of data parallelism. These properties lend themselves to the GPU’s single-instruction, multiple-thread (SIMT) execution model. In recent years, there have been many techniques to automatically map such regular programs to GPUs, often with great success (e.g., [13, 14]).

In contrast, there has been less attention paid to irregular programs, which perform unpredictable, data-driven accesses. The complexity and variety of irregular programs, especially the lack of regularity in their memory access patterns, has meant that most attempts at mapping irregular programs to GPUs have been ad hoc, hand-tuned efforts [2, 6, 17, 18, 21]. Many of these approaches take advantage of specific application semantics to deal with irregular memory accesses and enable effective GPU performance. What is missing are general-purpose techniques that can be used to implement a broad class of irregular algorithms on GPUs.

Because irregular algorithms as a whole span a wide range of applications, we choose to focus on a subset of programs to exploit common structures and patterns. In recent work, Jo and Kulkarni have identified tree traversal algorithms as an interesting class of irregular algorithms that have some commonality [9–11]. Tree traversal algorithms arise in varied domains, from data mining (nearest-neighbor searches) to graphics (accelerated object-ray intersection tests) to simulation (Barnes-Hut n-body simulations), and exhibit the same basic pattern: a set of points (e.g., rays) each traverse a single tree (e.g., a bounding-volume hierarchy that captures the spatial distribution of objects in a scene) to calculate some object value (e.g., which object a ray intersects). Section 2.1 discusses these algorithms in more detail.

Tree traversal algorithms represent an interesting target for GPU parallelization. As naturally parallel algorithms (the points’ traversals of the tree are independent), they exhibit the massive parallelism that GPUs excel at. However, because the tree structures are irregular, and the points’ traversals are input-dependent, simply running multiple traversals simultaneously on the GPU cannot take advantage of efficient memory accesses, seriously hindering performance (Section 2.2 discusses GPU architectures and the GPU performance model in more detail). Due to these characteristics, there have been several attempts to run tree-traversal algorithms on GPUs, but they have largely relied on algorithmic tweaks and application-specific optimizations to achieve reasonable performance [2, 5–7, 21]. This paper addresses the open question of whether there are general, systematic, semantics-agnostic techniques to map traversal codes to GPUs.

General transformations for traversal algorithms
In this paper, we show that tree traversal algorithms have several common properties that can be exploited to produce efficient GPU implementations. Crucially, we argue that these properties arise not from semantic properties of the algorithms, or implementation-
specific details, but instead emerge from common structural features of tree traversal algorithms. As a result, we can develop a catalog of techniques to optimize traversal algorithms for GPUs and provide a set of easily-checked structural constraints that govern when and how to apply these techniques. Thus, we address the problem of finding systematic, general techniques for implementing traversal codes on GPUs.

The primary transformation we develop is autoroping. One of the primary costs of performing general tree traversals on GPUs is the cost of repeatedly moving up and down the tree during traversal. There have been numerous attempts to develop “stackless” traversals that encode the traversal orders directly into the tree via auxiliary pointers, called “ropes,” obviating the stack-management that is otherwise necessary to traverse the tree \[6, 21\]. Unfortunately, encoding the ropes into the tree requires developing algorithm- and implementation-specific preprocessing passes, sacrificing generality for efficiency. Autoroping is a generalization of ropes that can be applied to any recursive traversal algorithm. In Section 3 we describe how autoropes work, elaborate on its utility for GPU implementations, and explain how traversal algorithms can be systematically transformed to support autoropes.

While autoropes is a wholesale transformation of traversal codes to better suit the GPU performance model, there are a number of other structural properties that affect optimization decisions. Section 4 describes how we identify and leverage various structural characteristics to improve memory access behaviors and minimize load imbalance. As with autoropes, these techniques rely not on semantic knowledge but only a structural analysis of the algorithm, and hence are generally applicable. Section 5 discusses how to correctly engineer and implement traversal algorithms for GPUs; these techniques apply for all traversal algorithms.

We demonstrate in Section 6 that our transformations are effective at extracting significant performance from GPU implementations of a number of tree-traversal benchmarks. In particular, we show that our GPU implementations perform well compared to multithreaded CPU implementations, and dramatically outperform simple recursive GPU implementations, despite not exploiting application-specific semantic properties. Section 7 surveys related work, and Section 8 concludes.

2 Background

In this section we discuss the structure of recursive traversals, which our transformations address, and present an overview of GPU architectures and programming models.

2.1 Traversal algorithms

Recursive traversal problems arise in a number of domains. In astrophysical simulation, the Barnes-Hut n-body code builds an oct-tree over a set of bodies, and each body traverses the tree to compute the forces acting upon it. In graphics, various structures such as kd-trees and bounding volume hierarchies are used to capture the relationships of data points, and these trees are repeatedly traversed to find nearest neighbors or correlation information. The common theme unifying all of these algorithms is that a tree is built over some data set, and then that tree is repeatedly traversed by a series of points. Notably, in a given algorithm, each point’s traversal may be different, based on properties of both the point and the tree.

At a high level, all these recursive tree algorithms can be abstracted using the pseudocode in Figure 1. Each point begins a recursive depth-first traversal starting at the tree root. As the point traverses the tree, portions of the tree are skipped by the traversal due to truncation conditions.

In some algorithms, such as Barnes-Hut, every point visits children in the same order; the only distinction between points’ traversals is whether a sub-tree is skipped due to truncation. In such algorithms, there is a single, canonical traversal order of the tree, and each point’s traversal order is consistent with the canonical order, with some portions of the sequence removed. In other algorithms, such as nearest neighbor, the order of the traversal might differ: the order in which a point visits children of a node is not fixed (the for- each child loop in line 12 may iterate over the children in a different order). In these algorithms, there is no canonical order, and each point’s traversal may differ significantly. We discuss some of the implications of these different types of algorithms when it comes to GPU implementations in Section 4.

2.2 GPU architecture

Modern graphics processors implement a highly parallel architecture well-suited for performing the same operation across thousands of individual data elements. Until recently these architectures lacked the general-purpose programming capability necessary to tackle all but a small subset of graphics processing operations. While there exist many different GPU architectures we focus on nVidia GPUs and the CUDA programming environment.

Unlike traditional CPU architectures where each thread may execute its own set of instructions, GPUs implement the single instruction multiple thread (SIMT) model, where a large collection of threads execute the same instruction simultaneously. Instead of focusing on optimizing instruction and memory access latency, GPUs aim to execute hundreds of the same instruction to achieve high data throughput by exploiting data parallelism. Using this approach, GPUs implement many simple cores capable of operating on individual data elements quickly and efficiently and do not contain complex instruction scheduling or prediction hardware.

nVidia GPUs are organized into a collection of cores called stream multiprocessors or SMs that are capable of executing hundred of threads in parallel. Each SM contains a set of simple cores called streaming processors, or SPs, which are responsible for executing an instruction for a single thread. Threads running in an SM are scheduled in groups of 32, called a warp, and execute in SIMD fashion on a set of SPs. Warps are further grouped into large blocks of threads for scheduling on an SM. To help mitigate latency, warps are scheduled in a multi-threaded fashion, where warps performing long running operations such as global memory accesses yield to allow other warps to proceed with execution.

A shared block of fast memory called shared memory serves as a software-controlled cache for the threads running in the SM. While shared memory can provide performance gains, if too much is used per thread, fewer thread blocks can occupy an SM simul-

```plaintext
1 void recurse (Point p, TreeNode n, ...) {
2   if (truncates?(p, n, ...)) return;
3   // update point based on current node
4   update(p, n, ...);
5   // continue traversal
6   foreach (TreeNode child : n.children()) {
7     recurse(p, child, ...);
8   }
9 }
10Figure 1. Abstract pseudocode for tree-traversal algorithms
```
3 Autoropes

This section discusses a novel transformation called autoropes that avoids unnecessary loads of tree data during traversal. A major source of overhead in GPU implementations of tree traversal codes is the data movement required as a thread moves up and down the tree during traversal. In particular, after traversing one path in the tree, a thread must return to higher levels of the tree to traverse down other children (viz. line 12 in Figure 1). As a result, interior nodes of the tree are repeatedly visited during traversals, requiring substantial extra work.

We note that the additional visits to interior nodes are not strictly necessary. If the tree were linearized according to traversal order, a traversal algorithm could be rewritten to simply iterate over the linear traversal order. This is complicated by the fact that each point’s traversal is different, so there is no single linear order of traversal. However, the overhead of superfluous loads is significant, so several application-specific approaches have been proposed to tackle the problem. In algorithms like Barnes-Hut, where a point’s traversal can be computed without executing the entire algorithm, a preprocessing pass can determine each point’s linear traversal, avoiding repeatedly visiting interior nodes during vector operations [15]. However, this preprocessing step can be expensive, and it cannot be done for algorithms such as nearest neighbor, where a point’s full traversal is only determined as the traversal progresses.

A more general approach to avoiding unnecessary visits is to preprocess the input data, but rather than preprocessing the traversals, preprocess the tree itself. Various GPU implementations of tree traversal algorithms have proposed installing ropes, extra pointers that connect a node in the tree not to its children, but instead to the next new node that a point would visit if its children are not visited [6, 21]. Figure 2 illustrates a simple binary tree with additional pointers that indicate the next element of a traversal (shown in dashed lines). Note that every node, including interior nodes, possesses such pointers.

With ropes “installed” into a tree, a point never needs to return to interior nodes to perform its traversals. In lieu of returning from a recursive call during a traversal, the point can follow the rope pointer to continue its traversal. Note that because ropes are installed on interior nodes as well, truncation of traversals is gracefully handled. For example, in Figure 2, if a point’s traversal is truncated at node 2, following the rope will correctly lead the point to the next node to visit, 6. By using ropes, a recursive traversal of the tree is effectively turned into an iterative traversal of the ropes.

Autoropes avoids these issues. Our transformation (a) can be performed without semantics-based knowledge; (b) applies to any traversal algorithm, even those with complex traversal patterns; and (c) does not require any preprocessing of the tree. Intuitively, autoropes transforms the recursive traversal of the tree into a depth-first search of the tree through a transformation akin to tail-call elimination, allowing an efficient implementation that avoids all superfluous visits to interior nodes. Autoropes ensures that each node of the tree is visited no more than once during any traversal. The following sections provide a high level overview of the autoropes technique (Section 3.1); identify the circumstances under which autoropes can be applied and discuss how to transform code to implement autoropes (Section 3.2); and argue for the correctness of our transformation (Section 3.3).

3.1 Overview of autoropes

Prior work that used ropes to speed up traversals relied on preprocessing the tree to install ropes into the nodes of the data structure. This results in good performance (traversals require following pointers in the already-loaded tree nodes), but does not work in the general case: there are too many ropes that could be followed from a node, and algorithm-specific knowledge is required to reduce the number of ropes. Because autoropes are intended to work for any tree traversal, we must sacrifice efficiency for generality.

Rather than storing ropes in the nodes of the data structure directly, we save ropes to a stack in the same way the next instruction addressed is saved to the function call stack. Figure 3 shows how an example recursive traversal is implemented with ropes stored onto a stack. At each point of the traversal, child nodes are pushed onto the rope stack in the order that the children will be traversed. To start the traversal, node 1 is popped from the top of the rope stack. At node 1, either node 2 or 5 may be the next child visited by the traversal. Since the ordering of child nodes can be determined at runtime, child nodes are pushed onto the rope stack in the appropriate order at run time—for example, first 5, then 2. The traversal will proceed by popping 2 from the top of the stack and then again determine the order that the children of 2 must visit. At node 2 we see the benefit of ropes, as we can jump directly to node 4 by popping the rope from the top of the stack without backtracking up to node 2. The traversal will continue in this fashion until the rope stack is emptied.

One way to view this approach is that we represent the traversal as an iterative, pre-ordered, depth-first search. Essentially, rather than pre-computing rope pointers and storing them in the tree, we compute them during the traversal and store them on the stack. This results in slightly more overhead than the hand-coded version (due to stack manipulation), but allows autoropes to work for any tree traversal.
1. void recurse(node root, point pt) {
2. if (!can_correlate(root, pt)) {
3. return;
4. }
5. if (is_leaf(root)) {
6. update_correlation(root, pt);
7. return;
8. } else {
9. recurse(root.left, pt);
10. recurse(root.right, pt);
11. }
12. }

Figure 4. Pseudo-tail-recursive function

1. void recurse(node root, point pt, int arg, int c) {
2. if (!can_correlate(root, pt)) {
3. return;
4. }
5. if (is_leaf(root)) {
6. update_closest(root, pt);
7. return;
8. }
9. if (closer_to_left(root, pt)) {
10. arg = arg + c + 1;
11. recurse(root.left, pt, arg, c);
12. recurse(root.right, pt, arg, c);
13. } else {
14. recurse(root.right, pt, arg, c);
15. recurse(root.left, pt, arg, c);
16. }

Figure 5. Pseudo-tail-recursive function with multiple paths

3.2 Applying autoropes

This section describes how the autoropes transformation is applied. Not every recursive traversal can be transformed directly to use autoropes; only functions that are pseudo-tail-recursive can be readily transformed in this manner. A pseudo-tail-recursive function is a function where all recursive function calls are the immediate predecessors either of an exit node of the function’s control flow graph, or of another recursive function call. That is, along every path from a recursive function call to an exit of the control flow graph, there are only recursive function calls. Pseudo-tail-recursion is a generalization of tail-recursive functions, which have the same property, but with only a single recursive function call.

While many traversal functions are expressed in pseudo-tail-recursive form, any function with arbitrary recursive calls and control flow can be systematically transformed to meet the criteria. At a high level, the transformation proceeds by turning intervening code between a pair of recursive calls into code that executes at the beginning of the latter call’s execution. In essence, computation intended to be performed at a particular node is “pushed” down to one of its children. By passing arguments identifying the call set and current child to the recursive method, a check at the beginning of the method can determine whether any computation needs to be performed on behalf of a node’s parent. Full details of this transformation can be found in the accompanying tech report [4].

Figures 4 and 5 give examples of pseudo-tail-recursive functions. Note that the former has just one path through the CFG that can execute recursive calls (lines 8–9 in Figure 4), while the latter has two paths through the CFG that execute recursive calls in different orders (lines 10–11 and lines 13–14 in Figure 5). The first step in transforming a pseudo-tail-recursive function to use autoropes is identifying the paths through the code that call recursive functions, a process we call static call set analysis.

3.2.1 Static call set analysis

A static call-set (we omit the “static” modifier hereafter for brevity) is the set of recursive calls executed along one path through a function. Because of control flow, a recursive function may contain only one call set, or it may contain several. Figure 4 contains just one call set; there is only one path through the code where recursive calls are made. In contrast, Figure 5 has two.

Our call set analysis proceeds through a control flow analysis. Note that identifying call sets is not quite the same as identifying all paths through a function: not all control flow that produces distinct paths will produce different call sets. We instead analyze a reduced CFG, which contains all recursive calls and any control flow that determines which recursive calls are made. We assume that all loops containing recursive calls can be fully unrolled, meaning that this reduced CFG is acyclic. Call sets can then be identified by computing all possible paths through the reduced CFG that contain at least one recursive call.

In general, a single recursive call may participate in more than one call set. In pseudo-tail-recursive functions, each recursive call exists in only one call set. This means that prior to executing any recursive call, the complete set of calls that will execute can be determined. This property of pseudo-tail-recursive functions simplifies the autoropes transformation.

Guided vs. unguided traversals Static call set analysis can be used for more than determining how to apply autoropes. An interesting property that can be identified using call-sets is the whether a recursive traversal is unguided or guided. An unguided traversal is a recursive traversal where all points will follow a common traversal order through the tree. Each traversal linearizes the tree in the same order, with the only differences being which portions of the tree are skipped due to truncation. For example in the algorithm shown in Figure 4, points visit the left child of a node before the right child, implying a global linearization order for the tree.

The alternative to an unguided traversal is a guided one. An example of guided traversal is an efficient recursive nearest neighbor search of a kd-tree, shown in Figure 5. At each node the recursive traversal must decide which child node to visit, left first then right or right first then left. Thus, different points may linearize the tree in different ways. Even two points that visit exactly the same nodes in the tree may visit them in different orders.

Static call-set analysis allows us to conservatively determine if a traversal is guided or unguided. An unguided traversal requires that if a point visits a particular tree node, it will choose the same call set as all other points that visit that tree node. If there is more than one static call set in an algorithm, we conservatively assume that points may take different paths at a particular tree node, depending on which call set is chosen; having a single call-set is a necessary condition for our analysis to classify a traversal as unguided. If a traversal algorithm has a single call set, then as long as the node arguments of the recursive calls are not dependent on any properties of the points, the traversal is unguided. Section 4 explains how we can leverage the guided versus unguided distinction to further optimize traversal codes on GPUs.

As recursive calls in tree traversals are used to visit children, we are essentially assuming that tree nodes have a maximum out-degree.

A more sophisticated analysis might be able to prove that all points will choose the same call set for a given tree node, for example, if the choice of call sets was independent of any point-specific information.
3.2.2 Transforming recursive traversals

As noted earlier, autoropes ensures that each node of the tree is visited exactly once during any traversal by saving a rope that points to to-be-visited nodes onto a stack. Autoropes saves these ropes to the stack push operations that save pointers to each child traversed. Traversal is facilitated by a loop that repeatedly pops the address of the next node in the traversal from the top of the stack until the stack is empty, indicating there are no more nodes to visit. At the beginning of each iteration of the loop the next node is popped from the stack. After popping the node from the stack the body of the function is executed on that node, possibly pushing more nodes onto the stack or returning due to a truncation.

An important detail to note is that the recursive calls are replaced with stack push operations, but the order in which nodes are pushed is the reverse of the original order of recursive calls, ensuring the order that nodes are visited remains unchanged. Also, function returns are replaced with a continue statement to prevent the traversal loop from prematurely exiting and preventing the remainder of the loop body from executing. We assume that the recursive traversal function does not return any values. However any function that returns data via the function call stack can be transformed to a function that returns no values by using a separate stack to maintain function return values.

We note that autoropes removes calls and returns from the traversal function; the implicit function stack is replaced with an explicit rope stack. This plays a crucial role in efficiency. Because autoropes applies to pseudo-tail-recursive functions, there is no need to save local variables on the rope stack; control never returns to a parent node. Similarly, because the original traversal function is pseudo-tail-recursive, the rope-stack preserves the ordering information of recursive calls, and there is no need to save additional information such as the return address that would have been stored on a function stack. We note the similarities between this transformation and the elimination of tail-call recursion, where the function stack, with storage for local variables, is eliminated.

Figure 7 shows the result of the autoropes transformation on our guided traversal example. Conveniently, autoropes does not need to distinguish between guided and unguided traversal, and can use the same transformation process. Again we note that this traversal function is pseudo-tail-recursive so we can directly apply autoropes to the function to produce an iterative traversal. We note in this example how autoropes handles function arguments (in this case, arg and c). If a function argument is not traversal-invariant (some invocations pass a different value for the argument), the argument must be stored on the rope stack along with the next node to be visited (line 16). In contrast, if an argument is traversal invariant (e.g., c), we can simply maintain its value outside the traversal loop, and need not store it on the rope stack.

3.3 Correctness

The autoropes transformation preserves all dependences that exist in the original recursive traversal because the order that the tree is traversed is unchanged. We refer to the guided traversal in Figure 5 to demonstrate the preservation of this order. If we consider a particular call-set in the traversal, for example when the left_is_near condition is satisfied, the recursive traversal will first visit the left child and its children, then, after unwinding the recursion, the right child and its children. Similarly for the other call-set, the right child is visited first and then the left. This ordering must be enforced at every node to produce an equivalent, rope-based traversal and is guaranteed by the rope stack that enforces last-in, first-out order of child ropes pushed at each recursive call site. By pushing the child nodes in reverse, nodes will be popped from the traversal stack in the original traversal order expressed by the recursive calls.

4 Lockstep traversals

This section discusses lockstep traversal, an approach to improving throughput of traversal algorithms on GPUs.

4.1 Memory coalescing and thread divergence

Effective execution on GPUs typically requires carefully managing two aspects of an application’s behavior, memory coalescing and thread divergence.

Memory coalescing issues arise as a consequence of the GPU’s memory architecture. Unlike CPUs, GPUs do not rely on caches to hide memory access latency. Instead they rely on high throughput memory to service an entire warp of threads. To achieve this throughput, the GPU memory controller requires threads within the same warp to access contiguous regions of memory so that a single wide block of memory can be transferred in a single transaction. When threads access non-contiguous elements, accesses must be serialized into multiple transactions, decreasing throughput. To help mitigate the performance impact of not adhering to such access patterns, modern GPUs will examine the accesses made by each thread and group them into as few memory transactions as
possible. For example, if all threads in a warp issue a load to the same memory address, only a single transaction is sent to the memory controller.

Thread divergence occurs when different threads in the same warp execute different instructions. On GPU architectures, all threads in a warp must be executing the same instruction; when different threads must execute different instructions, some of the threads in the warp are “disabled” and sit idle while the other threads perform useful work.

Because of the way that GPUs manage thread divergence, naïve recursive traversal algorithms can suffer from extreme thread divergence. If one thread in a warp makes a method call, all other threads will wait until the call returns before proceeding; as recursive calls can lead to long call chains, divergence can substantially decrease warp-level parallelism [8]. In contrast, autorope-enabled traversal algorithms do not suffer significant divergence: because the recursive method is translated into a loop over a stack, control immediately re-converges at the top of the loop, even as the threads diverge in the tree.

Interestingly, by reducing thread divergence, autorope implementations inhibit memory coalescing: as soon as threads’ traversals differ, they begin accessing different parts of the tree at the same time, and the threads are unlikely to return to the same part of the tree in the future. We find that the penalty of losing memory coherence far outweighs the benefit of decreased thread divergence. To address this, we introduce lockstep traversal, a transformation for unguided traversals (i.e., traversals with a single call set; see Section 3.2.1) which deliberately forces threads to diverge in order to keep them in sync in the tree, promoting memory coalescing.

4.2 Overview of lockstep traversal

In a lockstep traversal, rather than considering the traversals performed by individual points, the algorithm is recast in terms of the traversal performed by an entire warp. When a point is truncated at an interior node, ⊥, the point does not immediately move to the next node in its traversal. Instead, if other points in the warp wants to continue traversing the subtree rooted at ⊥, the truncated point is carried along with the other points in the warp, but masked out so that it does not perform computation. A warp only truncates its traversal when all the points in the warp have been truncated. Then, when the warp’s traversal returns to the node tree which the truncated point would have visited next, it is unmasked, and resumes its computation. Essentially, lockstep traversal forces autorope implementations to implement the same thread divergence behavior the GPU naturally provides for recursive implementations [8].

The masking and unmasking can be efficiently realized by pushing a mask bit-vector onto the rope stack marking whether a point should visit a child or not. When a warp visits a node, the mask bit-vector determines whether a given thread performs any computation or not. If a thread would truncate at a node (i.e., it would return from the recursive call), the mask bit for that thread is cleared. When deciding whether to continue its traversal, the warp constructs a new mask using a special warp voting/bit masking function. If all bits in the mask are cleared, the warp stops its traversal. If not, the warp continues its traversal, and propagates the mask using the rope stack. Figure 8 shows how the autorope version of a simple traversal code implements lockstep traversal.

There are multiple consequences of lockstep traversal. First, if multiple points in a warp have traversals that visit the same tree

```c
1 void recurse(node root, point pt) {
2 uint mask;
3 stack stk = new stack();
4 stk.push(root, ~0); // all threads active
5 while(!stk.is_empty()) {
6 root = stk.peek(0);
7 mask = stk.peek(1);
8 stk.pop();
9 if (bit_set(mask, threadId)) {
10 // this thread in the warp is still active
11 if (!can_correlate(root, pt))
12 bit_clear(mask, threadId);
13 if (!is_leaf(root)) {
14 update_correlation(root, pt);
15 bit_clear(mask, threadId);
16 }
17 }
18 }
19 // combine mask from all threads in warp
20 mask = warp_and(mask);
21 if (mask != 0) {
22 // a thread is still active
23 stk.push(root.right, mask);
24 stk.push(root.left, mask)
25 }
```

Figure 8. Lockstep traversal version of code in Figure 6

node, lockstep traversal ensures that all the points visit the node at the same time. Hence, all threads in the warp will be loading from the same memory location.

Second, as mentioned above, lockstep traversal only applies to unguided, single-call-set traversals. In traversals with multiple call sets, traversals are more likely to diverge. Further, because different points have different traversal orders, it is simply infeasible to keep the points in sync with each other. Section 4.3 discusses circumstances under which a multi-call-set algorithm can be transformed to a single-call-set version amenable to lockstep traversal.

Finally, a warp will visit all the tree nodes in the union of its constituent points’ traversals. In contrast, in a non-lockstep implementation, a warp will take time proportional to the longest traversal in the warp. This means that overall traversal time for a lockstep implementation can be longer than if the points were allowed to freely perform their traversals. Thus, if the threads in a warp have significantly divergent traversals, lockstep traversal may perform worse than the non-lockstep version. Section 4.4 discusses how to promote the similarity of traversals in a warp.

4.3 Lockstep for multi-call-set algorithms

In many guided traversal (multi-call-set) algorithms, the multiple call-sets are purely a performance optimization: by visiting children nodes in a different order, a thread’s traversal can terminate earlier, but regardless of the order of traversal, the result of the computation will be the same. For example, in the nearest-neighbor code of Figure 5, points prioritize which part of the tree to look for their nearest neighbor, resulting in two call sets. While this prioritization is an important performance optimization, it is not a correctness issue: even if a point chose the “wrong” call set, it would still find its nearest neighbor.

If a programmer can indicate (through annotation) that the multiple call sets are semantically equivalent (i.e., that they only offer different performance), we can automatically transform an algorithm to force all points in a warp to use a single call set at each step. We perform a simple majority vote between the threads in a warp, and then execute only the most popular call-set. This effectively turns a multi-call-set algorithm into a (dynamically) single-call-set algorithm. Note that even though all the threads in a particular warp will adopt the same traversal order, a different warp may make a different set of call-set choices: there are still multi-

3In our example a special function performs a bitwise and of each mask to produce a new mask that is given to each thread of the warp. On nVidia GPUs the ballot thread voting instruction can be used to implement an equivalent operation.
ple static call sets, but the transformation guarantees that there will only be one dynamic call set per warp. Hence, this approach is more efficient than statically choosing a single call-set for the entire traversal. While this transformation, unlike the other optimizations we discuss in this paper, requires some semantic knowledge, it requires only a simple annotation from the programmer. In the absence of this information, we do not perform the transformation: guided traversals will always perform non-lockstep traversals.

4.4 Point sorting
Sorting the traversals is a well-known technique for improving locality in traversal codes [19, 23]. By arranging the points carefully, similar traversals will execute closer together, increasing the likelihood that nodes will be found in cache. However determining an appropriate order for points is application-specific and often requires semantic knowledge to implement. Finding a good a priori order is especially difficult for algorithms like nearest-neighbor search, where the order of traversal is determined dynamically.

Point sorting can be of great benefit to lockstep traversal. Sorting ensures that nearby points—and hence the points in a given warp—have similar traversals. As discussed above, this ensures that the union of the warp’s traversals is not much larger than any individual traversal, minimizing the load balance penalty incurred by lockstep traversal. In contrast, unsorted points mean that a warp is likely to have highly divergent traversals, and the penalty for lockstep traversal will outweigh the load-balancing benefits.

While point sorting is algorithm-specific, and hence cannot be automated, Jo and Kulkarni’s run-time profiling method can be adopted to determine whether points are sorted (by drawing several samples of neighboring points from the set of points and seeing if their traversals are similar [11]). If the points are sorted, we use the lockstep implementation; otherwise we use the non-lockstep version. Section 6 looks at the performance of both lockstep and non-lockstep implementations of traversal algorithms on both sorted and unsorted inputs.

5 Implementation
In this section we discuss our automatic approach to replacing the CPU based recursive traversal with a fast GPU kernel. We also discuss several important decisions that must be made with respect to memory layout and storage, GPU tree representation, etc. These transformations are implemented in a C++ source-to-source compiler built on top of the ROSE compiler framework4.

5.1 Identifying the algorithmic structure
The first step in translating traversal algorithms to GPUs is identifying the key components of traversal algorithms: the recursive tree structure itself, the point structures that store point-specific information for each traversal, the recursive method that performs the recursive traversal, and the loop that invokes the repeated traversals. Jo and Kulkarni discussed approaches to automatically identifying these components, based on type information (the recursive structure contains a recursive field), structural analysis (the recursive method is recursive with a recursive structure argument), simple annotations (the loop is annotated to assert there are no inter-point dependencies) and heuristics (the point consists of any loop-variant arguments to the recursive function).

5.2 Transforming CPU traversals for the GPU
After identifying a repeated recursive traversal that can be parallelized onto the GPU we replace the original CPU implementation with a GPU kernel call. We separate our discussion of the transformation into two steps:

1. Transforming the recursive function call into an iterative GPU traversal kernel and,
2. Replacing the point loop and recursive function call with GPU kernel invocation.

Transforming the recursive traversal The first step to mapping the traversal to the GPU is preparing the recursive function for the autotopes transformation discussed in Section 3.2.2. Autotopes only requires that a function be expressed in pseudo-tail-recursive form to be correctly applied. While our current benchmarks are all pseudo-tail-recursive, we can apply a systematic transformation to restructure arbitrary recursive functions into pseudo-tail-recursive form, which we do not describe due to space limitations.

Figure 9a shows the original Barnes-Hut recursive traversal and the resulting GPU version is given in Figure 9b. First, all of the function arguments are replaced by a special structure that contains references to the original arguments passed into the recursive function. The GPU maintains a separate address space for all data, thus we must not only provide the original function arguments but also pointers to the GPU-resident copies of data structures.

The loop that repeatedly calls the traversal function will be parallelized by the CUDA call (as discussed below), but it can only execute a finite number of iterations. Hence, we strip mine the loop and move the inner loop into the recursive function, updating the initialization and increment statements so that each thread only processes one point per thread grid (lines 2–3 in Figure 9b). Finally, the traversal loop is introduced and the recursive call sites are replaced with stack pushes as described in Section 3.

Layout of rope stack and tree nodes An important consideration is how to lay out the rope stack and the nodes of the tree. The most general approach for laying out the stacks is to allocate global GPU memory for each threads’ stack where items are arranged such that if two adjacent threads are at the same stack level their accesses are

---

4 http://rosecompiler.org
made to contiguous location in memory, providing the best opportu-

nity for memory coalescing. In other words, the threads’ stacks
are interleaved in memory, rather than having each thread’s stack
contiguous in memory.

We can further optimize the stack storage if traversals are per-
formed in lockstep: all threads in a warp will perform the same
traversal, allowing any data which is not dependent on a particular
point to be saved per warp rather than per thread. Furthermore,
if the depth of the tree is reasonably small then the fast shared mem-
ory can be used to store all or part of the stack, reducing the amount
global memory access. For example in the Barnes-Hut traversal we
can apply lockstep traversal and use shared memory to maintain the
rope stack once per warp.

We must also consider how to represent the nodes of the tree in
memory. Typically for GPU applications, an array of structures is
transformed into a structure of arrays to facilitate memory coalesc-
ing, because adjacent threads in a warp will access fields of adja-
cent nodes. However, because we are accessing nodes of a tree,
there is limited opportunity to achieve a coalesced access pattern
that would exploit the structure-of-arrays layout. We have found
that the optimal way to organize nodes is to split the original struc-
ture into sets of fields based on usage patterns in the traversal. For
example, in our transformed Barnes-Hut kernel we load a partial
node that only contains the position vector of the current node and
its type (line 9). If the termination condition is not met then we
continue with the traversal and load another partial node (line 11)
that contains the indices of the nodes’ children.

Because the point data is copied to new storage during the traversal,
correct values may be computed if there are alternate access paths
to access the point data that are not transformed to use the copied
data. Rather than performing a complex alias analysis to
identify and transform all such access paths, we adopt a conserva-
tive, field based approach [9]. The copy-in/copy-out approach is
safe as long as (i) point fields read during traversal are not written
via other access paths and (ii) point fields written during traversal
are not read via other access paths.

Replacing the point loop with GPU kernel Our transformation
only targets the repeated recursive traversal of the program thus
care must be taken to preserve the original structure of the rest of
the code. As we discussed above, the point loop is strip-mined and
moved into the traversal function along with the other statements
of the point loop while any code that remains outside the traversal
is not modified. Since there may be arbitrary statements above and
below the recursive function call the point loop is split at the recur-
sive call site into a prologue and epilogue. Any variables that are
read after the recursive function call are saved to intermediate stor-
age and restored at the beginning of the epilogue. Because the GPU
memory resides in a separate address space we must also copy any
data to and from the GPU that is live-in and -out of the point loop,
and update CPU-resident data after the GPU kernel exits. Finally,
before the traversal kernel is invoked, an identical linearized copy
of the tree is constructed using a left-biased linearization, with the
nodes structured according to layout strategy mentioned above, and
copied to the GPU’s global memory.

6 Evaluation

We evaluate our techniques on four important scientific and engi-
neering traversal algorithms by comparing the overall performance
of multi-threaded CPU codes against GPU implementations de-
derived directly from our techniques.

6.1 Evaluation Methodology

To demonstrate the general applicability of our techniques, we trans-
formed several important tree traversal algorithms from different
application domains. For each benchmark, we evaluate non-lockstep
and lockstep versions of the algorithms (applying the call-set-reduction
optimization for guided traversals, as discussed in Section 4.3).

We compare these implementations against two alternative im-
plementations. First, we compare against a naïve GPU implement-
tion that uses CUDA compute capability 2.0’s support for recur-
sion to directly map the recursive algorithm to the GPU. We use a
masking technique similar to that described in Section 4 to imple-
ment non-lockstep and lockstep variants of the recursive implementa-
tion. In effect, the only difference between the naïve implementations
and ours is the use of autoropes. We also compare against parallel
CPU implementations of the same traversal algorithms.

CPU benchmarks were compiled using gcc 4.4.7 with optimization
level -O3 and GPU benchmarks were compiled using nvcc for
compute version 2.0 with CUDA toolkit version 5.06.

6.1.1 Platforms

We evaluate our benchmarks on two systems:

- The GPU system contains one nVidia Tesla C2070 GPU
  which contains 6GB of global memory, 14 SMs with 32 cores
  per SM. Each SM contains 64KB of configurable shared mem-
  ory.

- The CPU system contains four AMD Opteron 6176 proces-
sors that contain 12 cores running at 2300MHz. Each CPU
  has 64Kb L1 data cache per core, 512kB L2 cache per core,
  two 6MB shared L3 caches and 256GB of main memory.

Both Platforms run RHEL6.0 with Linux kernel v2.6.32.

6.1.2 Benchmarks

We evaluated our techniques on four benchmarks, each with multi-
ple inputs, for a total of 18 benchmark/input pairs. For each input,
we evaluated both sorted and unsorted versions of the input.

Barnes-Hut (RH) is a fast O(nlgm) n-body simulation [1] that per-
forms and efficient gravitational simulation between objects in an
environment represented by an oct-tree. It is an unguided algo-
rithm. We derived our implementation from the Lonestar bench-
mark suite [12], and ran our inputs for five timesteps. The inputs
are Plummer, the class C input from the Lonestar benchmark suite
that contains 1 million bodies of equal mass generated from the
Plummer model, and Random, a set of 1 million bodies of equal
mass, initialized with random position and velocity.

Point Correlation (PC) is a data mining algorithm that computes
the 2-point correlation statistic by traversing a kd-tree to find, for
each point in a data set, how many other points are in a given ra-
dius [20]. PC is an unguided algorithm. We evaluate four different
inputs: Covtype, an input derived from a 580,000 54-dimension
forest cover type dataset that has been reduced to 200,000 7-dimensional
points by random projection; Mnist, derived from a 8,100,000 784-
dimension dataset of handwritten digits that has been reduced to
200,000 7-dimensional points by random projection; Random, con-
sisting of 200,000 7-dimensional points with random coordinate

5 Although lockstep traversal should have no effect on recursive im-
plementations, we find that it improves performance. We speculate
that because the transformation explicitly forces thread divergence,
the compiler is able to generate more efficient code for the lockstep
variant using predication.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input Type</th>
<th>Sorted</th>
<th>Unsorted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes Hut</td>
<td>Plummer</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td></td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random</td>
<td>L</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Point Correlation</td>
<td>Covtype</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td></td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mnist</td>
<td>L</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Geocity</td>
<td>L</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>k-Nearest Neighbor</td>
<td>Covtype</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td></td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nearest Neighbor</td>
<td>Covtype</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td></td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vantage Point</td>
<td>L</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Performance summary of transformed traversals

values; and Geocity a 200,000 2-dimensional point city location dataset.

k-Nearest Neighbor (kNN) finds, for a given point in a data set, its k nearest neighbors. This algorithm operates by traversing a kd-tree to prune portions of the space that cannot contain nearby points. kNN is a guided algorithm with two call sets. We run the same input sets from PC for all platform and benchmark variants.

Nearest Neighbor (NN) is a variation of nearest neighbor search with a different implementation of the kd-tree structure. NN is a guided algorithm with two call sets. We ran the same input sets from PC for all platform and benchmark variants.

Vantage Point Tree (VP) is also a variation of nearest neighbor search using a vantage point tree [27]. Like NN, VP is also a guided two call set algorithm. We ran the same input sets from PC for all platform and benchmark variants.

6.2 Results

Table 1 summarizes the results of our techniques. Columns 1–3 specify the name of the benchmark, the input set used and the type of traversal performed. Lockstep traversals are indicated by an L, non-lockstep traversals are indicated by an N. Lockstep variants are automatically produced for BH and PC, while kNN, NN and VP use the annotation described in Section 4.3 to enable the lockstep variant. To characterize the performance of our techniques we measure the total traversal time, average number of nodes accessed per point, the speedup of our GPU traversal versus a single-thread and 32-thread CPU implementation, and the improvement against a recursive GPU implementation. Columns 4–8 contain the results for sorted inputs and columns 9–13 are the results for unsorted inputs. Figures 10 and 11 compare our GPU implementations against CPU implementations as the number of CPU threads increases (all numbers normalized to GPU performance).

Due to the differences in behavior between irregular applications, as well as the input-dependent behavior within a given irregular application, there are no universal performance trends. However, we can see general patterns, despite the occasional outlier.

In general, we find that our GPU implementations are far faster than naïve recursive implementations on GPUs. Recall that lockstep traversal can also be applied to the recursive GPU implementations, so all comparisons are apples-to-apples. In almost all cases, our autoropes transformation, which turns the recursive traversal into an iterative one, is able to deliver significant improvements.

Overall, we find that the best variant (lockstep vs. non-lockstep) for each benchmark/input pair far outperforms the single-threaded CPU version (except kNN for the Geocity input, discussed below). Furthermore, as shown in Figures 10 and 11, except for a handful of inputs, the best GPU variants of our benchmarks outperform the CPU implementation up to at least 8 threads, and in most cases outperform the CPU implementation even at 32 threads.

In all cases, lockstep implementations traverse more nodes than their non-lockstep counterparts. Nevertheless, for sorted inputs, lockstep implementations outperform non-lockstep ones; the extra work performed by lockstep traversal is outweighed by lockstep's other performance benefits. For unsorted inputs, the story is
more muddled: single-call-set applications (BH and PC) still benefit from lockstep traversal, while multi-call-set applications do not. Section 6.3 discusses the reasons for this in more detail.

In the case of NN, kNN and VP, the lockstep variants required a small amount of input from the programmer. Looking just at the non-lockstep versions of these benchmarks, we see that for most inputs (sorted and unsorted) NN is faster even when the CPU version uses 32 threads, while kNN is faster to 8 threads for sorted inputs and 12 threads for unsorted inputs, and VP is faster to 12 threads for sorted inputs and 20 threads for unsorted inputs. Note that the reason the GPU versions appear to do better on unsorted inputs is because the CPU versions do worse.

There is one consistent outlier to these broad trends: the Geocity input performs especially well on the CPU for both kNN and VP, and as a result, the GPU versions are considerably slower than the CPU version. This is primarily because Geocity is a low-dimension input, and as a result, traversals are very short, promoting good locality and performance on the CPU [10]. Furthermore, the input...
Table 2. Average work expansion per warp of lockstep traversals (standard deviation in parenthesis)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input</th>
<th>Sorted</th>
<th>Unsorted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes Hut</td>
<td>Plummer</td>
<td>1.35 (1.35)</td>
<td>8.97 (9.40)</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>1.51 (1.53)</td>
<td>17.35 (17.70)</td>
</tr>
<tr>
<td>Point Correlation</td>
<td>Lovetype</td>
<td>1.49 (1.53)</td>
<td>2.94 (2.99)</td>
</tr>
<tr>
<td></td>
<td>Minist</td>
<td>2.15 (2.21)</td>
<td>5.61 (5.71)</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>2.01 (2.03)</td>
<td>6.89 (6.96)</td>
</tr>
<tr>
<td></td>
<td>Geocity</td>
<td>3.04 (3.31)</td>
<td>1.30 (1.31)</td>
</tr>
<tr>
<td>k-Nearest Neighbor</td>
<td>Lovetype</td>
<td>19.59 (30.21)</td>
<td>187.34 (285.08)</td>
</tr>
<tr>
<td></td>
<td>Minist</td>
<td>17.03 (19.58)</td>
<td>60.86 (70.12)</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>6.87 (8.62)</td>
<td>89.29 (102.89)</td>
</tr>
<tr>
<td></td>
<td>Geocity</td>
<td>4.03 (5.99)</td>
<td>1479.11 (1591.59)</td>
</tr>
<tr>
<td>Nearest Neighbor</td>
<td>Lovetype</td>
<td>5.20 (6.37)</td>
<td>35.85 (67.86)</td>
</tr>
<tr>
<td></td>
<td>Minist</td>
<td>4.46 (5.66)</td>
<td>20.68 (27.99)</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>5.64 (6.29)</td>
<td>50.50 (58.31)</td>
</tr>
<tr>
<td></td>
<td>Geocity</td>
<td>4.62 (5.31)</td>
<td>618.00 (885.71)</td>
</tr>
<tr>
<td>Vantage Point</td>
<td>Lovetype</td>
<td>4.70 (5.52)</td>
<td>39.34 (41.87)</td>
</tr>
<tr>
<td></td>
<td>Minist</td>
<td>5.58 (5.87)</td>
<td>22.05 (22.47)</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>6.62 (7.01)</td>
<td>20.73 (21.26)</td>
</tr>
<tr>
<td></td>
<td>Geocity</td>
<td>3.68 (4.74)</td>
<td>57.76 (91.04)</td>
</tr>
</tbody>
</table>

is highly clustered, leading to extremely variable behavior on the GPU: traversals in a warp may have very different lengths, leading to load imbalance and hence poor performance.

On the whole, by choosing the right set of optimizations, we can automatically map traversal algorithms to GPUs and achieve results that are (a) substantially better than naïve GPU implementations; (b) much faster than single-threaded CPU implementations; and (c) competitive with even highly-threaded CPU implementations. Crucially, all of this can be achieved without taking advantage of application-specific knowledge.

6.3 Work expansion in lockstep traversals

As we discussed in Section 4.2, lockstep traversal can lead to significant gains when points of a warp perform similar traversals or potentially degraded performance when traversals diverge and threads in a warp sit idle while visiting unimportant nodes. We measure the cost of this divergence by comparing the number of nodes accessed by each warp in the lockstep traversal with the number of nodes in the longest traversal of each warp (which captures how long a warp would take to finish in the non-lockstep variant). This metric measures the amount of work expansion that occurs due to lockstep traversal; Table 2 shows the work expansion for each benchmark.

We see that sorting is a highly effective optimization when used in conjunction with lockstep traversals, especially for single call-set algorithms such as PC and BH. We also note that PC and BH have low work expansion in the unsorted case compared to the other benchmarks, which is why the lockstep variant still performs well with unsorted points.

While we expect and found sorting to benefit lockstep traversals of single call-set algorithms, multi-call-set algorithms are more susceptible to work expansion because the traversals take sub-optimal paths through the tree due to our dynamic single-call-set optimization (discussed in Section 4.3), causing traversals that take the “wrong” path to run longer. This tradeoff is clearly shown by the extreme level of work expansion in the kNN benchmark, where the non-lockstep traversals performed best even for sorted inputs.

7 Related work

Much of the work on running tree traversal algorithms on GPUs has focus on the graphics domain, where hierarchical structures such as kd-trees are repeatedly traversed to compute ray-object intersections efficiently. Foley et al. develop two stackless approaches for kd-tree traversals that use per-ray bounding boxes to prune nodes that have already been searched [3]. Further work by Hughes et al. use an implicit kd-tree structure to compute the position of the next node, avoiding excessive backtracking [7]. Popov et al. expand on the stackless kd-tree traversal algorithm by adding ropes to the leaf nodes of the tree, as discussed in Section 3 [21]. Similar work on bounding volume hierarchy (BVH) traversals incorporates knowledge of the traversal structure to build a simple state machine to determine the next node to visit [6]. Another implementation of BVH traversals in GPUs groups rays into packets and then traverses the tree in lock step, sharing a per-packet stack to avoid traversal divergence between rays in the same packet [5]. All of these techniques take advantage of application-specific knowledge.

Researchers also focus on efficiently implementing other, non-tree traversal, irregular codes on GPUs. Vineet et al. develop a GPU implementation of Boruvka’s minimum spanning tree algorithm using data parallel primitives such as sort, scan and reduce [24]. Merrill et al. discuss parallelization strategies and performance characterization of GPU graph traversals using various algorithms based on data parallel primitives [18]. Wei et al. map linked-list prefix computation to GPUs by using a splitting technique and other semantic knowledge to partition the computation [25]. Similar work on list ranking discusses the need to ensure load balancing to achieve good performance for irregular algorithms [22]. Méndez-Lojo et al. present a GPU implementation of inclusion-based points-to analysis that performs graph rewrites in terms of matrix-matrix multiplication by leveraging clever encodings of a compressed sparse row representation [17]. Huo et al. examined efficient scheduling of recursive control flow on GPUs, and present results which improve upon traditional post-dominator based re-convergence mechanisms designed to handle thread divergence due to control flow within a procedure [8].

GPU performance for irregular programs suffers from irregular memory references. Zhang et al. develop G-Streamline, a software framework which removes dynamic irregularities from GPU applications through data reordering and job swapping [28]. Wu et al. show that finding an optimal solution to irregular memory references is NP-complete, and illuminate the space, time and complexity tradeoffs of algorithm designs for data reorganization [26].

Prior work in enhancing temporal locality for tree traversals on CPUs can also benefit GPUs. Sorting approaches (Section 4.4) use application semantics to schedule similar traversals consecutively, for Barnes-Hut [23] and ray tracing [16, 19]. Jo and Kulkarni develop automatic compiler transformations to enhance temporal locality for tree traversals, analogous to loop tiling in regular programs [10, 11]. Jo et al. also develop transformations to facilitate vectorization of tree traversals, targeting SIMD instructions on commodity CPUs [9].

8 Conclusions

We described a series of transformations that enable the efficient execution of tree traversal algorithms on GPUs. These techniques, unlike in most prior work on GPU implementations of irregular algorithms, do not rely on application-specific semantic knowledge, instead leveraging only structural properties of traversal algorithms. We show that our transformations produce GPU implementations that are superior to naïve GPU implementations and competitive with large-scale multithreaded CPU implementations.

Acknowledgments

This work was supported in party by an NSF CAREER award (CCF-1150013) and a DOE Early Career award (DE-SC0010295). The Tesla GPUs used in the evaluation were graciously donated by NVIDIA corporation. The authors would like to thank Sriрам Krishnamoorthy and Michael Garland for discussions regarding the effects of the lockstep transformation.
References