1-1-2013

General Transformations for GPU Execution of Tree Traversals

Michael Goldfarb
Purdue University - Main Campus, mgoldfar@purdue.edu

Youngjoon Jo
Purdue University, yjo@purdue.edu

Milind Kulkarni
Purdue University - Main Campus, milind@purdue.edu

General Transformations for GPU Execution Of Tree Traversals

Michael Goldfarb

Youngjoon Jo

Milind Kulkarni

TR-ECE-13-09

August 09, 2013

Purdue University

School of Electrical and Computer Engineering

465 Northwestern Avenue

West Lafayette, IN 47907-1285

General Transformations for GPU Execution of Tree Traversals

Michael Goldfarb, Youngjoon Jo and Milind Kulkarni
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN
{mgoldfar, yjo, milind}@purdue.edu

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.*, [12, 13]).

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, 5, 16, 17, 20]. 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, 10]. 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, 4–6, 20]. 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 [5, 20]. 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 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 octtree 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 locations of objects in a scene, and then rays traverse the tree to determine which object(s) they intersect. In data mining, kd-trees 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 foreach 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.

```
1  TreeNode root = ...
2  foreach (Point p : points)
3   recurse(p, root, ...);

5  void recurse(Point p, TreeNode n, ...) {
6   if (truncate?(p, n, ...)) return;

8   //update point based on current node
9   update(p, n, ...);

11  //continue traversal
12  foreach (TreeNode child : n.children()) {
13   recurse(p, child, ...);
14  }
15 }
```

Figure 1. Abstract pseudocode for tree-traversal algorithms

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 simultaneously, reducing parallelism. The SMs are connected to a large high-latency, high-throughput *global* DRAM memory with a hardware-managed level 2 cache. Global memory is capable of achieving very high throughput as long as threads of a warp access elements from the same 128-byte segment. If memory accesses are *coalesced* then each request will be merged into a single global memory transaction; otherwise the hardware will group accesses into as few transactions as possible to satisfy the request.

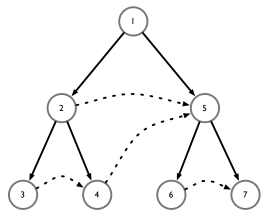


Figure 2. Tree with ropes installed. Solid lines are children pointers, dashed lines are rope pointers.

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, resulting in numerous extra loads.

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 unnecessary loads during vector operations [14]. 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 loads 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 [5, 20]. 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, ⑤. By using ropes, a recursive traversal of the tree is effectively turned into an iterative traversal of the ropes.

A major drawback of using ropes is that the tree must be preprocessed to install the ropes. This requires an additional pass prior to performing the actual traversals. In addition, rope installation is non-trivial for more complex traversals where there is no single traversal order—multiple ropes may need to be installed, to account for different possible orders. As a result, all prior attempts to use ropes to accelerate tree traversals have relied on application-specific transformations that leverage the semantics of the algorithm to efficiently place ropes [5, 20].

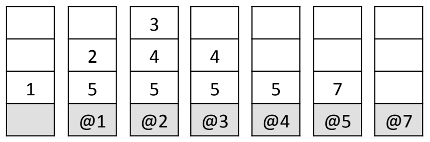


Figure 3. Saving ropes on the stack

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 loads of 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 much 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 ③ 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 algorithm. By using an explicit stack of rope pointers to drive the traversal, we avoid having to visit tree nodes more than once, without preprocessing or knowing algorithm semantics.

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

```
1  void recurse(node root, point pt) {
2   if(!can_correlate(root, pt))
3    return;
4   if(is_leaf(root)) {
5    update_correlation(root, pt);
6    return;
7   } else {
8    recurse(root.left, pt);
9    recurse(root.right, pt);
10  }
11 }
```

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   if(is_leaf(root)) {
5    update_closest(root, pt);
6    return;
7   }
8   if(closer_to_left(root, pt)) {
9    arg=arg+c+1;
10   recurse(root.left, pt, arg, c);
11   recurse(root.right, pt, arg, c);
12  } else {
13   recurse(root.right, pt, arg, c);
14   recurse(root.left, pt, arg, c);
15  }
16 }
```

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

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 ; we omit a discussion of this transformation for lack of space.

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[1], 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 fact 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 whether 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[2]. 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.

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 dynamically at runtime, obviating the need for additional preprocessing steps or semantic knowledge about the traversal. Transforming a recursive traversal to perform a rope based traversal is straightforward for pseudo-tail-recursive functions.

Figures 6 shows the result of applying the autoropes transformation to the unguided pseudo-tail-recursive traversal functions of Figure 4. In essence the autoropes transformation simply replaces the recursive call-sites with 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

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

[2]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.

```
void recurse(node root, point pt) {
 stack stk = new stack();
 stk.push(root);
 while(!stk.is_empty()) {
  root = stk.pop();
  if(!can_correlate(root, pt))
   continue;
  if(is_leaf(root)) {
   update_correlation(root, pt);
  } else {
   stk.push(root.right);
   stk.push(root.left)
  }}}
```

Figure 6. Autoropes transformation applied to Figure 4

```
1  void recurse(node root, point pt, int arg, int c) {
2   stack stk = new stack();
3   stk.push(root, arg);
4   while(!stk.is_empty()) {
5    root = stk.peek(0);
6    arg = stk.peek(1);
7    stk.pop();
8    if(!can_correlate(root, pt))
9     continue;
10   if(is_leaf(root)) {
11    update_closest(root, pt);
12    continue;
13   }
14   if(closer_to_left(root, pt)) {
15    arg=arg+c+1;
16    stk.push(root.right, arg);
17    stk.push(root.left, arg);
18   } else {
19    stk.push(root.left, arg);
20    stk.push(root.right, arg);
21   }}}
```

Figure 7. Autoropes transformation applied to Figure 5

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 while preventing the remainder of the loop body from executing. We assume that our recursive traversals function do 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 nodes are 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 [7]. 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 tree node 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 [7].

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[3]. 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 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.

[3]In 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.

```
void recurse(node root, point pt) {
 uint mask;
 stack stk = new stack();
 stk.push(root, ~0); // all threads active
 while(!stk.is_empty()) {
  root = stk.peek(0);
  mask = stk.peek(1);
  stk.pop();
  if (bit_set(mask, threadId)) {
   // this thread in the warp is still active
   if(!can_correlate(root, pt))
    bit_clear(mask, threadId);

   if(is_leaf(root)) {
    update_correlation(root, pt);
    bit_clear(mask, threadId);
   }
  }
  // combine mask from all threads in warp
  mask = warp_and(mask);
  if(mask != 0) {
   // a thread is still active
   stk.push(root.right, mask);
   stk.push(root.left, mask)
  }}}
```

Figure 8. Lockstep traversal version of code in Figure 6

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 multiple *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 [18, 22]. 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 whether their traversals are similar [10]). 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 framework[4].

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 autoropes transformation discussed in Section 3.2.2. Autoropes 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

```
1 void recurse(oct_node p, oct_node root, float dsq) {
2  if (!far_enough(root, p) && root.type != LEAF) {
3   for (i = 0; i < 8; i++)
4    recruse(p, child, dsq * 0.25);
5  } else {
6   update(root, p);
7  }
8 }
```

(a) Barnes-Hut recursive call before transformation

```
1  void recurse(gpu_params params) {
2   for(pid = blockIdx.x*blockDim.x + threadIdx.x;
3    pid < params.n; pid += gridDim.x*blockDim.x) {
4    p = params.bodies[pid];
5    STACK_INIT();
6    STACK_PUSH(params.root, params.dsq);
7    while(sp >= 0) {
8     STACK_POP(root, dsq);
9     node0 = params.nodes0[root];
10    if (!far_enough(node0, p) && node0.type != LEAF) {
11     node1 = params.nodes1[root];
12     for (i = 7; i >= 0; i--)
13      STACK_PUSH(node1.children[i], dsq * 0.25);
14    } else {
15     update(node0, node1, p);
16    }
17   }}}
```

(b) Barnes-Hut recursive call after transformation

Figure 9. Barnes-Hut recursive call transformation

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 made to contiguous location in memory, providing the best opportunity 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 performed 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 memory 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 coalescing, because adjacent threads in a warp will access fields of adjacent nodes. However, because we are accessing nodes of a tree,

[4]`http://rosecompiler.org`

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 structure 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, incorrect 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 conservative, field based approach [8]. 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 recursive call site into a prologue and epilogue. Any variables that are read after the recursive function call are saved to intermediate storage 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 engineering traversal algorithms by comparing the overall performance of multi-threaded CPU codes against GPU implementations derived directly from our techniques.

6.1 Evaluation Methodology

To demonstrate the general applicability of our techniques, we transformed 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 implementations. First, we compare against a naïve GPU implementation that uses CUDA compute capability 2.0's support for recursion to directly map the recursive algorithm to the GPU. We use a masking technique similar to that described in Section 4 to implement non-lockstep and lockstep variants of the recursive implementation[5]. 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.

[5] Although lockstep traversal should have no effect on recursive implementations, 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.

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 5.0[6].

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 memory.

- The **CPU system** contains four AMD Opteron 6176 processors 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 multiple inputs, for a total of 18 benchmark/input pairs. For each input, we evaluated both sorted and unsorted versions of the input.

Barnes-Hut (BH) is a fast O(nlgn) n-body simulation [1] that performs and efficient gravitational simulation between objects in an environment represented by an oct-tree. It is an unguided algorithm. We derived our implementation from the Lonestar benchmark suite [11], 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 radius [19]. 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**, consisting of 200,000 7-dimensional points with random coordinate 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 [26]. 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] https://developer.nvidia.com/cuda-toolkit

			Sorted					Unsorted				
Benchmark	Input	Type	Traversal Time (ms)	Avg. # Nodes	Speedup		Improv. vs Recurse	Traversal Time (ms)	Avg. # Nodes	Speedup		Improv. vs Recurse
					vs 1	vs 32				vs 1	vs 32	
Barnes Hut	Plummer	L	669.07	3345	150.07	7.18	1409%	4580.48	22107	32.55	1.85	1364%
		N	8206.30	2551	12.24	0.59	-26%	13938.18	2551	10.70	0.61	210%
	Random	L	213.71	1068	211.16	12.77	1400%	2467.92	11909	34.85	2.75	1348%
		N	2391.84	671	18.87	1.14	-19%	4517.50	671	19.04	1.50	416%
Point Correlation	Covtype	L	5738.00	76160	123.08	15.48	199%	18533.40	257771	45.31	4.60	202%
		N	48582.40	28057	14.54	1.83	-2%	37871.60	28057	22.17	2.25	345%
	Mnist	L	2070.60	26188	48.93	4.68	173%	7204.40	97653	24.24	1.94	188%
		N	9707.00	6138	10.44	1.00	71%	8689.40	6138	20.10	1.61	618%
	Random	L	3125.40	37618	52.20	6.04	186%	11586.60	156353	23.00	2.52	202%
		N	17017.40	10161	9.59	1.11	42%	16978.00	10161	15.70	1.72	504%
k-Nearest Neighbor	Covtype	L	2907.00	25277	4.72	0.28	332%	16049.00	197160	1.57	0.12	57%
		N	1816.40	1982	7.56	0.45	180%	2408.50	1982	10.48	0.77	269%
	Mnist	L	6396.00	60172	4.54	0.26	181%	16153.00	199840	3.28	0.24	64%
		N	3827.30	4150	7.59	0.44	161%	5359.30	4150	9.89	0.74	234%
	Random	L	2008.00	16695	9.63	0.43	599%	16234.00	200000	2.30	0.17	59%
		N	2448.00	2937	7.90	0.35	84%	3692.90	2937	10.11	0.73	244%
Nearest Neighbor	Covtype	L	12350.20	53948	27.09	3.17	124%	58470.80	259132	7.48	0.70	131%
		N	38116.10	16669	8.78	1.03	348%	34814.90	16669	12.57	1.18	925%
	Mnist	L	14673.60	65812	25.64	3.19	119%	60540.20	267645	8.26	0.87	124%
		N	43886.00	19020	8.57	1.07	427%	46764.00	19020	10.70	1.13	769%
	Random	L	1869.70	8808	15.32	0.75	110%	15666.10	73011	2.53	0.19	107%
		N	2559.00	1838	11.19	0.55	427%	3846.00	1838	10.30	0.77	866%
Vantage Point	Covtype	L	1787.00	11814	6.13	0.48	18%	10235.4	109719	2.25	0.14	65%
		N	1623.40	686	6.75	0.52	295%	1704.60	686	13.50	0.81	365%
	Mnist	L	4034.20	36347	11.46	0.87	43%	13835.00	150992	6.61	0.39	66%
		N	5114.00	2763	9.04	0.68	412%	5599.80	2763	16.33	0.96	451%
	Random	L	4541.00	41054	11.13	1.00	45%	13130.60	143189	7.14	0.43	67%
		N	5074.60	2659	9.96	0.90	401%	5355.00	2659	17.50	1.05	453%

Table 1. Performance summary of transformed traversals

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).

Though direct performance comparisons to hand-written implementations are difficult due to lack of source, we note that our general speedups over CPU implementations are comparable to those reported for hand-tuned versions of BH [2] and NN [20], despite our application-agnostic approach.

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.

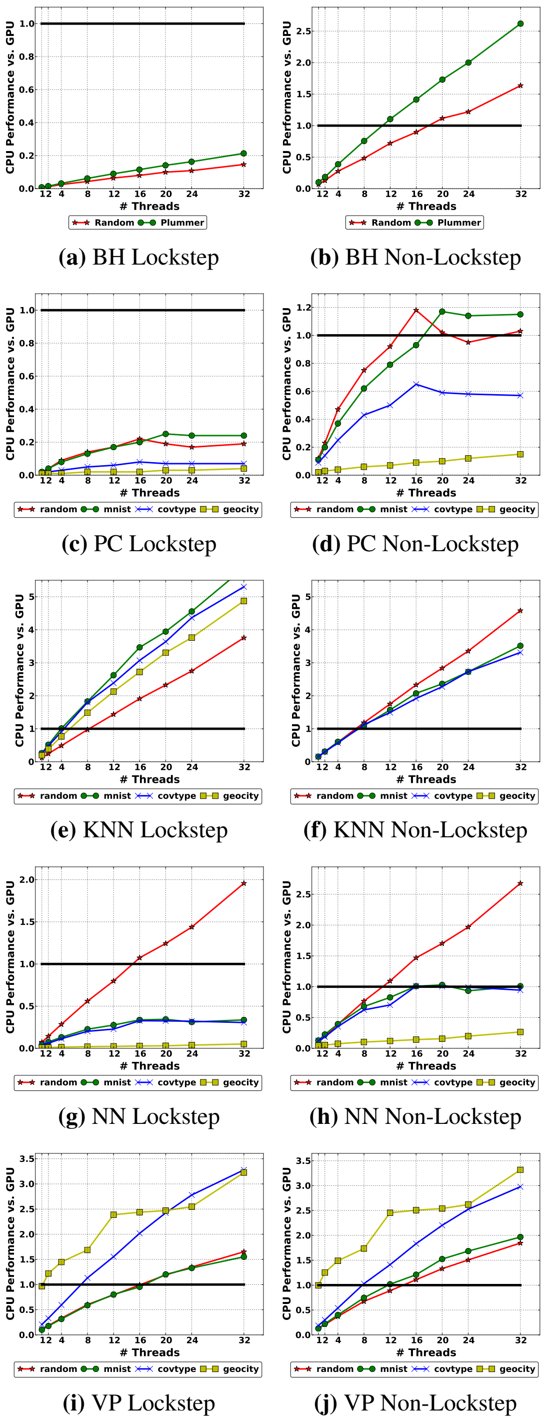
Overall, we find that the best variant (lockstep vs. non-lockstep) for each benchmark/input pair far outperforms the single-threaded CPU version (except kNN for the **geocity** input, discussed below). Furthermore, as shown in Figures 10 and 11, except for a handful of inputs, the best GPU variants of 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.

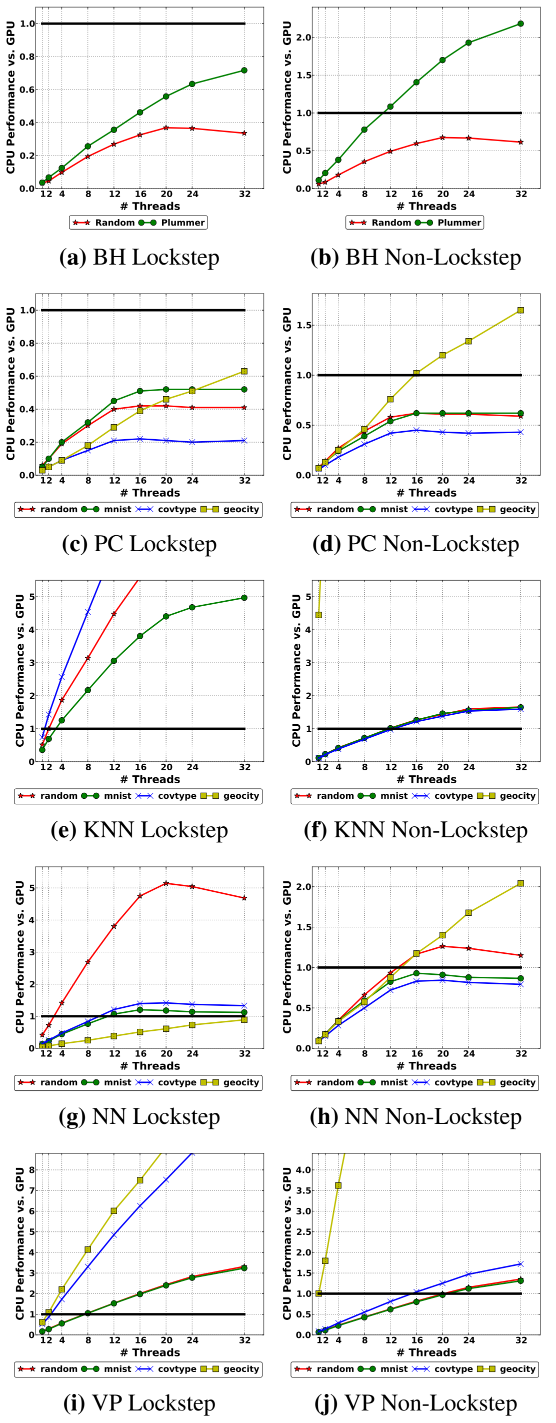
There is one consistent outlier to these broad trends: the **geocity** input performs especially well on the CPU for both kNN and VP, and as a result, the GPU versions are considerably *slower* than the CPU version. This is primarily because **geocity** is a low-dimension input, and as a result, traversals are very short, promoting good locality and performance on the CPU [9]. Furthermore, the input 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*.



(a) BH Lockstep (b) BH Non-Lockstep (c) PC Lockstep (d) PC Non-Lockstep (e) KNN Lockstep (f) KNN Non-Lockstep (g) NN Lockstep (h) NN Non-Lockstep (i) VP Lockstep (j) VP Non-Lockstep

Figure 10. Speedup of GPU traversal versus CPU (sorted)



(a) BH Lockstep (b) BH Non-Lockstep (c) PC Lockstep (d) PC Non-Lockstep (e) KNN Lockstep (f) KNN Non-Lockstep (g) NN Lockstep (h) NN Non-Lockstep (i) VP Lockstep (j) VP Non-Lockstep

Figure 11. Speedup of GPU traversal versus CPU (unsorted)

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

Benchmark	Input	Sorted		Unsorted	
Barnes Hut	Plummer	1.33	(1.35)	8.97	(9.40)
	Random	1.51	(1.53)	17.35	(17.78)
Point Correlation	Covtype	4.16	(6.25)	20.71	(40.11)
	Mnist	6.20	(6.20)	27.49	(8.24)
	Random	4.35	(4.88)	20.00	(23.21)
	Geocity	101.08	(207.30)	1.46	(1.47)
k-Nearest Neighbor	Covtype	19.59	(30.21)	187.54	(285.08)
	Mnist	17.03	(19.58)	60.86	(70.12)
	Random	6.87	(8.62)	89.29	(102.89)
	Geocity	4.03	(8.99)	1479.11	(1591.59)
Nearest Neighbor	Covtype	5.20	(8.37)	35.85	(67.86)
	Mnist	4.46	(5.66)	20.68	(27.99)
	Random	5.64	(6.29)	50.60	(58.31)
	Geocity	4.62	(31.69)	618.00	(885.71)
Vantage Point	Covtype	4.70	(5.24)	39.34	(41.87)
	Mnist	5.58	(5.87)	22.05	(22.47)
	Random	6.62	(7.01)	20.73	(21.26)
	Geocity	3.68	(4.74)	57.76	(91.04)

Table 2. Average work expansion per warp of lockstep traversals (standard deviation in parenthesis)

low work expansion in the *unsorted* case compared to the other benchmarks, which is why the lockstep variant still performs well with unsorted points. Interestingly BH achieves better convergence of its traversals than PC even though there are more potential paths in the oct-tree traversal. We attribute the higher work expansion of PC to the size of the adjustable correlation radius that determines when a traversal truncates; by decreasing this radius traversals will truncate more quickly leading to better load balance.

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 [6]. 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 [20]. 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 [5]. 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 [4]. 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 [23]. Merrill *et al.* discuss parallelization strategies and performance characterization of GPU graph traversals using various algorithms based on data parallel primitives [17]. Wei *et al.* map linked-list prefix computation to GPUs by using a splitting technique and other semantic knowledge to partition the computation [24]. Similar work on list ranking discusses the need to ensure load balancing to achieve good performance for irregular algorithms [21]. 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 [16]. Huo *et al.* examined efficient scheduling of recursive control flow on GPUs, and present results which improve upon traditional post-dominator based reconvergence mechanisms designed to handle thread divergence due to control flow within a procedure [7].

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 [27]. 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 [25].

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 [22] and ray tracing [15, 18]. Jo and Kulkarni develop automatic compiler transformations to enhance temporal locality for tree traversals, analogous to loop tiling in regular programs [9, 10]. Jo *et al.* also develop transformations to facilitate vectorization of tree traversals, targeting SIMD instructions on commodity CPUs [8].

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 Sriram Krishnamoorthy and Michael Garland for discussions regarding the effects of the lockstep transformation.

References

[1] J. Barnes and P. Hut. A hierarchical o(n log n) force-calculation algorithm. *nature*, 324:4, 1986.

[2] M. Burtscher and K. Pingali. An efficient CUDA implementation of the tree-based barnes hut n-body algorithm. In *GPU Computing Gems Emerald Edition*, pages 75–92. 2011.

[3] T. Foley and J. Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '05, pages 15–22, 2005.

[4] J. Gunther, S. Popov, H.-P. Seidel, and P. Slusallek. Real-time ray tracing on gpu with bvh-based packet traversal. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, RT '07, pages 113–118, 2007.

[5] M. Hapala, T. Davidovic, I. Wald, V. Havran, and P. Slusallek. Efficient Stack-less BVH Traversal for Ray Tracing. In *Proceedings 27th Spring Conference of Computer Graphics (SCCG) 2011*, pages 29–34, 2011.

[6] D. M. Hughes and I. S. Lim. Kd-jump: a path-preserving stackless traversal for faster isosurface raytracing on gpus. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1555–1562, Nov. 2009.

[7] X. Huo, S. Krishnamoorthy, and G. Agrawal. Efficient scheduling of recursive control flow on gpus. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ICS '13, pages 409–420, New York, NY, USA, 2013. ACM.

[8] Y. Jo, M. Goldfarb, and M. Kulkarni. Automatic vectorization of tree traversals. In *PACT '13: Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, 2013.

[9] Y. Jo and M. Kulkarni. Enhancing locality for recursive traversals of recursive structures. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 463–482, 2011.

[10] Y. Jo and M. Kulkarni. Automatically enhancing locality for tree traversals with traversal splicing. In *Proceedings of the 2012 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, 2012.

[11] M. Kulkarni, M. Burtscher, K. Pingali, and C. Cascaval. Lonestar: A suite of parallel irregular programs. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 65–76, April 2009.

[12] S. Lee and R. Eigenmann. Openmpc: Extended openmp programming and tuning for gpus. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, 2010.

[13] C.-K. Luk, S. Hong, and H. Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, 2009.

[14] J. Makino. Vectorization of a treecode. *J. Comput. Phys.*, 87:148–160, March 1990.

[15] E. Mansson, J. Munkberg, and T. Akenine-Moller. Deep coherent ray tracing. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, RT '07, pages 79–85, 2007.

[16] M. Méndez-Lojo, M. Burtscher, and K. Pingali. A gpu implementation of inclusion-based points-to analysis. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 107–116. ACM, 2012.

[17] D. Merrill, M. Garland, and A. Grimshaw. Scalable gpu graph traversal. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 117–128, 2012.

[18] B. Moon, Y. Byun, T.-J. Kim, P. Claudio, H.-S. Kim, Y.-J. Ban, S. W. Nam, and S.-E. Yoon. Cache-oblivious ray reordering. *ACM Trans. Graph.*, 29(3):28:1–28:10, July 2010.

[19] A. Moore, A. Connolly, C. Genovese, A. Gray, L. Grone, N. Kanidoris II, R. Nichol, J. Schneider, A. Szalay, I. Szapudi, et al. Fast algorithms and efficient statistics: N-point correlation functions. *Mining the Sky*, pages 71–82, 2001.

[20] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum*, 26(3):415–424, Sept. 2007. (Proceedings of Eurographics).

[21] M. Rehman, K. Kothapalli, and P. Narayanan. Fast and scalable list ranking on the gpu. In *Proceedings of the 23rd international conference on supercomputing*, pages 235–243, 2009.

[22] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy. Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multipole, and radiosity. *J. Parallel Distrib. Comput.*, 27(2):118–141, June 1995.

[23] V. Vineet, P. Harish, S. Patidar, and P. Narayanan. Fast minimum spanning tree for large graphs on the gpu. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 167–171. ACM, 2009.

[24] Z. Wei and J. JaJa. Optimization of linked list prefix computations on multithreaded gpus using cuda. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010.

[25] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '13, pages 57–68, New York, NY, USA, 2013. ACM.

[26] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, SODA '93, pages 311–321, 1993.

[27] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for gpu computing. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS XVI, pages 369–380, New York, NY, USA, 2011. ACM.

APPENDIX

A From Recursive to Pseudo-Tail Recursive

The recursive traversal shown in Figure 12a is an example of a typical non pseudo tail recursive traversal function that contains multiple recursive call-sets surrounded by conditional statements and intermediate code that must be executed between the recursive function calls. In Figure 12b we show the result of applying the pseudo tail recursive transformation to the code from Figure 12a. Eventually we will replace the recursive calls with stack push operations to facilitate traversal on the GPU so we must consider any code that must be executed between any two recursive function calls and rearrange it so that it can be executed before any of the recursive calls are made. In our example this code is represented by the function *intermediate* and may arbitrary code but does not include any recursive calls. As we no longer can rely on the function call stack to return to a particular location in the function call we must another on another mechanism for determining what statements must be executed before we invoke any of the recursive function calls.

Our basic approach to transform a general recursive traversal to pseudo tail recursive form we consider is to "push" all of the intermediate statements that execute after one function into the function call of the next recursive call. Effectively any intermediate code becomes the first set of statements in the recursive function, but is conditionally executed depending on what recursive call is being made. What is immediately apparent is that knowing which intermediate code to execute corresponds directly to the call-sets the statements are a member of. For example the first recursive call marks the end of the first call-set and contains some set of state-

```
void recurse(node root, point pt) {
 update_closest(root, pt);
 if(!is_leaf(root)) {
  left_near = left_is_near(root, pt);
  if(left_near) {
   recurse(root.left, pt);
  } else {
   recurse(root.right, pt);
  }
  intermediate();
  dist = split_dist(root, pt);
  if(pt.cdist > dist) {
   if(left_near) {
    recurse(root.right, pt);
   } else {
    recurse(root.left, pt);
   }
  }}}
```

(a) A non pseudo tail recursive function

```
void recurse(node root,point pt,int call,float dist)
{
 if(call == 1 || call == 3) {
  intermediate();
  if(!(pt.cdist > dist))
   return;
 }
 update_closest(root, pt);
 if(!is_leaf(root)) {
  left_near = left_is_near(root, pt);
  if(left_near) {
   recurse(root.left, pt, 0, 0.0);
   dist = split_dist(root, pt);
   recurse(root.right, pt, 1, dist);
  } else {
   recurse(root.right, pt, 2, 0.0);
   dist = split_dist(root, pt);
   recurse(root.left, pt, 3, dist);
  }}}
```

(b) A pseudo tail recursive function after transformation

Figure 12. Transformation to pseudo tail recursive form

ments denoted by the *prologue* function. Likewise *intermediate1* and *intermediate2* are each part of the next two call-sets ended by the other recursive calls respectively. To rearrange this code we can simply move all statements from call-sets 2 and 3 to the beginning of the recursive function and add an additional argument to recurse that denotes the call-set number for that particular call. Note that statements from call-set 1 do not need to be rearranged because they will always be executed before any of the recursive calls.

```
prologue();
recurse();
intermediate1();
recurse();
intermediate2();
recurse();
```

```
if(call==2)
 intermediate1();
if(call==3)
 intermediate2();
prologue();
recurse(1);
recurse(2);
recurse(3);
```

The above example demonstrates our basic strategy for transforming recursive traversals to pseudo tail recursive form: identify the call-sets and hoist the intermediate statements to the top of the function. With this basic strategy in mind we will consider the case when a recursive function is executed under arbitrary control flow.

```
prologue();
if(cond1())
 recurse(left);
 eplilogue1();
 if(cond2())
  intermediate1();
  recurse(right);
 else
  recurse(center)
else
 recurse(right);
 epilogue2();
 if(cond3())
  intermediate2();
  recurse(right);
 else
  recurse(center)
```

```
if(call==2)
 epilogue1();
 if(!cond2()) return;
 intermediate1();
if(call==3)
 if(cond2()) return;
if(call==4)
 epilogue2();
 if(!cond3()) return;
 intermediate2();
if(call==5)
 if(cond3()) return;
prologue();
if(cond1())
 recurse(left, 1);
 recurse(right, 2);
 recurse(center, 3)
else
 recurse(right, 4);
 recurse(left, 5);
 recurse(center, 6);
```

Control flow requires particularly careful handling because evaluation of conditionals may depend on or more recursive calls proceeding the test execute after the recursive call completes. Consequently some call-sets may never be executed but because we are not able to resolve the condition early the recursion, which will eventually be replaced by a stack push, must be executed. Then at a later point in time when the conditional can be correctly resolved the call-sets that would not be reached will be invalidated by returning before executing any other statements of the recursive call.

The above example demonstrates how these arbitrary conditions, denoted by *condN* functions, can be handled when rearranging the statements of each call-set. Here call-set 2 contains the statements *epilogue1*, *cond2*, *intermediate2* and call-set 3 contains the statements *epilogue1*, and *cond2*. The first observation we make is that call-set 2 corresponds to *cond2* evaluating to true, thus if we are executing statements of call-set 2 and find that *cond2* is not true then the call-set must be invalid and we need to stop executing any statements from call-set two. The next important observation is that any statements up to and including *cond2* are shared by both call-sets 2 and 3. If we were to naively execute these statement in both call-set 2 and 3 then we would possibly produce incorrect results because there may be side-effecting statements. However, we know that in order to reach call-set 3 statements from call-set 2 up to the conditional test must have already been run, even if the call-set was invalid. Therefore those statements in common need not be executed when call-set 3 is reached because they are guaranteed to execute when call-set 2 is reached. Lastly, like the check in call-set 2, the hoisted code from call-set 3 must ensure that it is supposed to execute by evaluating *cond2* to ensure the branch leading to call-set 3 was taken.

Another problem induced by control flow is that there may be ambiguity in which statements must be executed in a call-set. An example is when control flow diverges along two separate paths but then merges back at a later point as shown below. On the left call-set 3 contains the statements *epilogue1*, *epilogue2* and *intermediate* but *epilogue1*, *epilogue2* are dependent on the result of *cond1*. Given this ambiguity there are two possible ways to resolve it: (1) Peel the epilogues of the if-statements into a another if-then-else block or (2) Fuse the statements that occur beyond the if-then-else block into each branch. We choose method (2) when handling such cases.

```
prologue();
if(cond1())
  recurse();
  epiloge1();
else
  recurse();
  epilogue2();

intermediate();
recurse();
```

```
prologue();
if(cond1())
  recurse();
  epiloge1();
  intermediate();
  recurse();
else
  recurse();
  epilogu2();
  intermediate();
  recurse();
```

The last problem we must consider is how to handle epilogue statements that follow the last recursive call before the end of the function. Unlike the statements that precede a recursive call there is no mechanism by which the statements of that call-set can be executed because we rely on the action of performing the recursive call to know that we must execute statements of its call-set. Thus in these instances we introduce another recursive call that does not visit a node but facilitates the execution of the statements in the call-set. One particular problem we have not addressed here is how to handle loops and other arbitrary control flow within the recursive call. Loops that do not invoke any recursive functions are trivially handled by the steps outlined above. While we can replace any arbitrary loop in the recursive function with additional recursive calls the number of nodes pushed onto the node stack will be potentially unbounded. Thus we assume all loops that call recursive functions have a constant number of iterations can can be removed with unrolling.

```
prologue();
recurse(left);
recurse(right);
epilogue();
recurse(NULL);
```

```
if(call=3)
  epilogue();  return;
prologue();
recurse(left,1);
recurse(right,2);
recurse(NULL,3);
```