Hybrid CPU-GPU scheduling and execution of tree traversals

Jianqiao Liu, Nikhil Hegde and Milind Kulkarni
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN, USA
{liu1274, hegden, milind}@purdue.edu

ABSTRACT

GPUs offer the promise of massive, power-efficient parallelism. However, exploiting this parallelism requires code to be carefully structured to deal with the limitations of the SIMT execution model. In recent years, there has been much interest in mapping *irregular* applications to GPUs: applications with unpredictable, data-dependent behaviors. While most of the work in this space has focused on *ad hoc* implementations of specific algorithms, recent work has looked at generic techniques for mapping a large class of *tree traversal* algorithms to GPUs, through careful restructuring of the tree traversal algorithms to make them behave more regularly. Unfortunately, even this general approach for GPU execution of tree traversal algorithms is reliant on *ad hoc*, hand-written, algorithm-specific scheduling (*i.e.*, assignment of threads to warps) to achieve high performance.

The key challenge of scheduling is that it is a highly irregular process, that requires the inspection of thread behavior and then careful sorting of those threads into warps. In this paper, we present a novel scheduling and execution technique for tree traversal algorithms that is both general and automatic. The key novelty is a hybrid, *inspector-executor* approach: the GPU partially executes tasks to inspect thread behavior and transmits information back to the CPU, which uses that information to perform the scheduling itself, before executing the remaining, carefully scheduled, portion of the traversals on the GPU. We applied this framework to six tree traversal algorithms, achieving significant speedups over optimized GPU code that does not perform application-specific scheduling. Further, we show that in many cases, our hybrid approach is able to deliver better performance even than GPU code that uses hand-tuned, application-specific scheduling.

CCS Concepts

•Theory of computation → Scheduling algorithms; •Computing methodologies → Massively parallel algorithms; Parallel programming languages;

ICS '16, June 01-03, 2016, Istanbul, Turkey

ISBN 978-1-4503-4361-9/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2925426.2926261

Keywords

Heterogeneous architectures, GPU, Scheduling, Irregular applications, Tree traversal

1. INTRODUCTION

GPUs offer the promise of massive, energy-efficient parallelism, providing hardware that can execute hundreds of simultaneous parallel threads. As a result, the last decade has seen intense efforts towards mapping applications and kernels to GPUs to take advantage of that parallelism. Unfortunately, achieving such highly efficient parallelism requires compromises: GPUs present a somewhat limited SIMT (single-instruction, multiple-thread) execution model, where, to take full advantage of the parallel execution resources, threads that are executing simultaneously must perform the same computation (avoiding *control divergence*) and access memory in a predictable way (avoiding *memory divergence*). In other words, while GPUs appear to be well-suited to executing data-parallel algorithms, the full power of the GPU cannot be exploited unless the data-parallel tasks are similar to each other. As a result, most successful GPU algorithms are *regular*, with predictable control flow and memory access patterns.

There has been considerably less success tackling *irregular* algorithms. These algorithms, which feature data structures such as trees and graphs and data-dependent behavior, are much more difficult to map to GPUs, as the input-dependence precludes grouping together threads to minimize control divergence and the pointer-based data structures means that even if threads are performing similar operations, their memory accesses are likely not predictable, increasing memory divergence. As a result of these difficulties, most attempts to map irregular applications to GPUs have been one-off implementations: for each new algorithm, a new, *ad hoc* implementation for GPUs must be developed [5, 6, 8, 10, 11, 16, 17, 23].

Recently, Goldfarb *et al.* developed a general framework for mapping a class of kernels, recursive tree traversals, to GPUs [7]. These applications, which include classic algorithms such as Barnes-Hut [3] and kd-tree–based nearest neighbor searches [4], perform multiple recursive traversals of tree structures. The multiple traversals expose tremendous amounts of data parallelism. However, since the behavior of each traversal is input dependent, the data parallel tasks are not identical, making GPU execution challenging. If the tasks are merely mapped to SIMT threads, each thread ultimately experiences substantial memory and control divergence. To overcome this, Goldfarb *et al.* developed two transformations, *autoroping* and *lockstepping*, that restructure these tree traversal algorithms so that they execute effectively on GPUs (see Section 2.2 for more details).

Unfortunately, to achieve maximum performance using their frame-

work, Goldfarb *et al.* used application-specific "sorting" optimizations. To minimize control divergence while avoiding memory divergence, Goldfarb *et al.* reorganized the set of traversals to be performed so that the traversals that were grouped into SIMT thread groups were likely to touch similar portions of the tree. Figuring out a fast, effective way to perform this sorting requires careful understanding of an algorithm's behavior. In other words, this prior work relies on application-specific *sorting* to achieve high performance. What is missing is an approach to tackling these problems that does not rely on application-specific knowledge to be effective.

In this paper, we take advantage of a key insight about the behavior of tree traversal algorithms that allows us to perform effective scheduling without performing application-specific sorting: even though traversals are data-dependent and hence inherently unpredictable, if the behavior of a single traversal is examined part of the way through its execution, its future behavior is highly correlated with its past behavior. In other words, two traversals that have behaved similarly for the first half of their execution are likely to behave similarly for the second half, as well.

The insight that past traversal behavior is correlated with future behavior has been exploited before, in narrower, or application-specific contexts by Zhang *et al.* [31] and Pingali *et al.* [22], and in a more general tree-traversal context by Jo and Kulkarni [13, 12]. Fundamentally, these approaches all interleave the scheduling component (tracking past behavior and reorganizing computations based on that behavior) with the execution (perform the next phase of computations). This tight coupling has an advantage: by interleaving tracking and scheduling with execution, the schedule can be continuously adapted in response to the profiling information. However, these approaches also have a serious disadvantage: because the tracking and scheduling occur continuously, and are highly irregular processes, this tightly-coupled approach is ill-suited to execution on a GPU.

In this work, we introduce a completely different way of automatically scheduling tree traversals. We base our approach on an extension of the prior insight about traversal behavior. The depth-first nature of recursive traversals means that the behavior of a traversal as it explores the "lower half" of the tree (*i.e.*, nodes at depth more than half the tree height) is largely determined by its behavior in the "upper half" of the tree (*i.e.*, which nodes in the upper half of the tree the traversal visits, and in which order). In other words, two traversals that have similar behaviors in the upper half of the tree will behave similarly in the lower half of the tree, as well. However, the vast majority of the *work* performed by the traversal occurs in the lower half of the tree. As a result, it is possible to examine the behavior of traversals as they visit the upper half of the tree, and use *just that information* to reschedule the traversals as they execute the lower half of the tree.

Crucially, since this upper-half execution is a small fraction of the overall execution time, it is not burdensome to perform that execution more than once. This fact suggests a *automatic*, *hybrid*, *insepctor-executor* approach that can be readily mapped to a GPU. We perform the "top half" of all of the traversals on the GPU once, to collect profiling data about the behavior of each traversal. This *inspector* stage is highly data-parallel, with a small memory footprint, and is well-suited to the GPU. The GPU then transmits that profiling data back to the CPU, which performs the highly irregular scheduling process to determine a *new* schedule of execution. This new schedule of execution is then used to *execute* the original tree traversal algorithm on the GPU, using strategies such as Goldfarb *et al.*'s. Even though the inspector and scheduler phases add additional overhead to the application, the gains from the optimized schedule during the execution phase win out.

Contributions

This paper makes several contributions.

1. We formulate the general scheduling problem, and show that it is NP-hard, necessitating scheduling heuristics.
2. We introduce a hybrid, inspector executor–based, dynamic scheduling algorithm that performs partial traversals on the GPU, then reschedules the traversals on the CPU, before completing the work on the GPU.
3. We develop optimized versions of this scheduling algorithm that exploit structural properties of traversal algorithms to further improve our dynamic scheduling.
4. We develop a new skeleton for writing the GPU kernel portion of tree traversals that minimizes unnecessary memory accesses.
5. We implement a framework that performs this dynamic scheduling in a general, application-agnostic manner, allowing programmers to produce hybrid CPU-GPU implementations of tree traversal algorithms.

We compare our framework for hybrid traversal algorithms to Goldfarb *et al.*'s approach (the best available general framework for traversal algorithms on GPUs). We demonstrate on six benchmarks that (a) our approach produces substantially faster implementations than Goldfarb *et al.*'s application-agnostic GPU implementations and (b) our approach yields performance that nears, or even substantially exceeds, Goldfarb *et al.*'s hand-tuned implementations, which use application-specific traversal schedules.

2. BACKGROUND

This section describes the necessary background for the remainder of the paper. We briefly cover the SIMT execution model of GPUs and Goldfarb *et al.*'s approach for mapping traversal algorithms to GPUs.

2.1 GPU execution model

GPUs use a SIMT (single-instruction, multiple-thread) execution model that allows multiple threads to execute efficiently in parallel. SIMT execution is, essentially, vector execution: multiple threads can execute in parallel provided that all the threads are performing *the same instruction*. In the simplest case, consider a group of threads that each perform exactly the same operation on different pieces of data in an array. In a SIMT execution, some number of threads will be combined into a single group (called a "warp" in NVIDIA parlance, and a "wavefront" by AMD; for brevity, we will use the term "warp" hereafter). These threads will execute in lockstep, each executing the same instruction simultaneously. As long as all the threads perform the same instruction, and all memory accesses performed by the threads are well-structured (*e.g.*, adjacent locations in an array), the GPU will deliver large amounts of efficient parallelism.

The key to the SIMT execution model, which both lends it its ease of use and hides a series of performance pitfalls, is how it deals with situations when threads *do not* perform exactly the same instruction (*control divergence*), or do not access memory in well-structured ways (*memory divergence*). In the presence of divergence, GPU utilization can drop precipitously, at which point the parallelism advantages of a GPU are moot: execution on a CPU can often be faster. Unfortunately, irregular applications often incur both types of divergence. Because of data-dependent behavior,

threads often suffer from control divergence. Because of the dynamic memory allocation inherent in pointer-based data structures, threads often access unpredictable memory locations on loads, leading to memory divergence. As a result, most attempts to map irregular applications to GPUs have required very careful, application-specific tricks and techniques to achieve good performance.

2.2 Autorope and lockstep traversal

Goldfarb *et al.* described an approach for mapping a general class of irregular applications—those that perform repeated tree traversals—to GPUs [7]. These applications are characterized by the following structure: a set of "points" each traverse a single tree in a recursive, depth-first manner. However, GPU implementations of recursive tree traversals suffer from a specific performance pitfall. After a thread finishes traversing along a particular path in the tree, it must return to upper nodes in the tree to reach other branches. Thus the interior nodes are repeatedly traversed, an overhead that is compounded by the expense of numerous recursive calls on a GPU. To mitigate this overhead, Goldfarb *et al.* proposed a transformation called *autoropes* [7].

Ropes are a common technique for mapping traversal algorithms to GPUs. Rather than letting threads discover which nodes to visit through a series of recursive calls, ropes are additional pointers installed in the tree that directly point to the next node to be traversed (*e.g.*, to a sibling node in the tree), avoiding the expense of revisiting interior nodes. Ropes provide a linearization of the tree. Unfortunately, the particular targets of rope pointers are application specific, and are complicated when multiple traversal orders are possible. Autoropes is an application-agnostic transformation that uses a stack of dynamically-instantiated rope pointers to linearize trees. When visiting a node using autoropes, the thread pushes pointers to the children nodes onto a *rope stack* in the reverse order they will be traversed. Then, instead of making recursive calls, the autorope traversal just iterates over the rope stack, eliding the overhead of recursion, and ensuring that each node is visited just once.

Autoropes replaces the recursive call stack with a simple iteration over the rope stack. As a result, threads experience significantly less control divergence (because they are all simply iterating over a stack). Unfortunately, this means threads in a warp can diverge in the tree, with different threads touching very different portions of the tree, resulting in unnecessary memory traffic. *Lockstepping* mitigates this problem by introducing additional control flow that keeps threads in sync in the tree during traversal. When a thread is truncated at certain node 𝔫, it doesn't move to the next node through autorope stack directly. Instead, if other threads in the warp want to continue the traversal to the subtree rooted at node 𝔫, the thread will be carried along by others, masked out from any computation. A warp only truncates its traversal when all its threads in the warp have given up the traversal. To ensure that lockstepping does not result in many threads being dragged along through the tree doing no useful work, it is important to carefully schedule the computation so that threads with similar traversals get grouped together into a warp. Together with autoropes, lockstep traversal delivers high performance for well-scheduled inputs [7].

3. TRAVERSAL SCHEDULING

As explained by the previous section, efficiently mapping tree traversals on GPUs requires carefully *scheduling* those traversals so that traversals that are grouped together into the same warp are as similar as possible. This ensures that *lockstep* traversal is able to exploit substantial commonality in the memory accesses performed by traversals while not overly expanding the amount of work done by a warp. This section shows that the general scheduling problem, Sched is NP-hard, necessitating the use of heuristics. It then summarizes prior scheduling and sorting heuristics for tree traversal algorithms.

3.1 Sched is NP-hard

The general scheduling problem for tree traversals, which we call Sched, is simple to define. Given a point p that represents a traversal, define $t(p)$ as the set of nodes visited during p's traversal of the tree. For two points, p_i and p_j, define the *difference* between the traversals, $\delta(p_i, p_j)$ as $t(p_i) \cup t(p_j) - (t(p_i) \cap t(p_j))$—in other words, the nodes that exist in one traversal but not in the other. Note that these are the nodes that result in non-convergent computation, as only one point needs to visit them.

Sched is the following problem. Given a set of points, $\{p_1, ..., p_n\}$, produce a sequence s of those points that minimizes:

$$\Delta = \sum_{i=1}^{n-1} |\delta(s_i, s_{i+1})|$$

In other words, Sched minimizes the total differences between consecutive points in the sequence—it produces a *sorted* sequence.

Theorem 1. Sched *is NP-hard.*

Proof. To show that Sched is NP-hard, we reduce from Hamiltonian Path: given an undirected graph $G = (V, E)$, find a path that visits each vertex once. We show how to design a tree traversal problem based on G where solving Sched for that problem solves the Hamiltonian Path problem.

First, build a tree with $|E| + 2$ leaves. Let the first leaf in the tree be x and the last leaf in the tree be y. Label each of the other $|E|$ leaves in the tree according to the edges in E; call these *edge-based* leaf nodes. Then, attach subtrees with $4|E|$ nodes to x and y. For each vertex $v_i \in V$, we specify a point p_i with a traversal defined as follows: p_i visits all of the nodes in the tree *except* the subtrees rooted at x and y and any edge-based leaf node corresponding to an edge *not* incident on v. The only leaf nodes visited by p_i correspond to the edges incident on v_i, $E(v_i)$. Then define two additional points p_x and p_y, which truncate at x and y, respectively, and otherwise visit all of the other nodes in the tree except the edge-based leaf nodes.

Note that for any two vertices v_i and v_j, $|\delta(p_i, p_j)| = |E(v_i)| + |E(v_j)| - 2|E(v_i) \cap E(v_j)|$. Note that the last term is zero unless v_i and v_j share an edge. Also, for all vertices v_i, $|\delta(p_x, p_i)| = |\delta(p_y, p_i)| = 4|E| + |E(v_i)|$.

Now we solve Sched across all the points—those corresponding to the vertices of G as well as p_x and p_y. Note, first, that minimizing Δ requires that p_x and p_y be scheduled first or last—otherwise their $4|E|$ difference penalty is accounted for twice. Each other point appears in two pairs in the scheduled sequence. Every edge therefore is accounted for four times—twice for each vertex it is incident on—unless it is incident on both vertices of a pair. We thus have that Δ is:

$$8|E| + 4|E| - 2Q$$

Where Q is the number of point pairs in the sequence produced by Sched that share a leaf node—in other words, the number of vertex pairs that share an edge. Δ is minimized when Q is maximized. In other words, Δ is minimized when all vertex pairs in the sequence share an edge—a Hamiltonian Path. Hence, if a Hamiltonian Path exists, Sched will find it. □

Note that Sched merely refers to an arbitrary set of tree traversals. However, we are not interested in arbitrary tree traversals—

```
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    else {
7      recurse(root.left, pt);
8      recurse(root.right, pt);
9    }
10 }
```

Figure 1: Point correlation

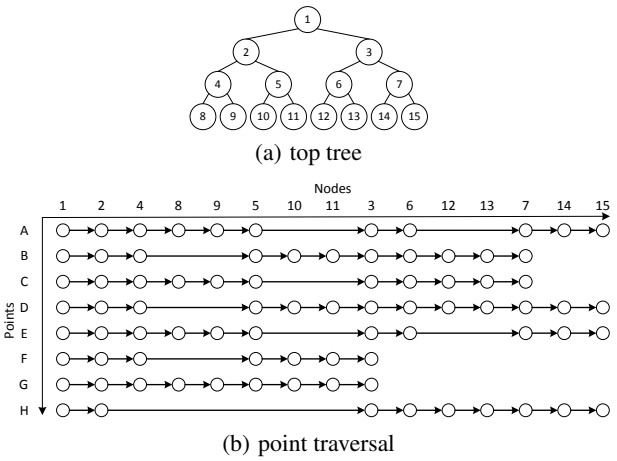
we are interested in tree traversals that are generated from recursive tree traversal algorithms, as Point Correlation algorithm in Figure 1. It is straightforward to construct such an algorithm for a given graph. When building the tree for the graph, color each of the nodes that should be visited by *all* of the points white, all of the nodes visited only by p_x red, all of the nodes visited only by p_y green, and all of the other nodes blue. It is clear that the `can_correlate` predicate in Figure 1 can be modified to ensure that the points visit exactly the nodes they are supposed to: if a node is white, then `can_correlate` is always true; if a node is red or green, `can_correlate` is true only if the point is p_x or p_y, respectively; if a node is blue, `can_correlate` looks up whether the graph vertex associated with the point is incident on the edge associated with the node. This modified recursive algorithm, when presented with a set of points derived from the vertices of the graph in question, and a tree built as specified above, produces exactly the set of traversals needed for SCHED to find a Hamiltonian path if one exists.

3.2 Prior sorting heuristics

Because SCHED is NP-hard, we must instead turn to heuristics to schedule traversals. The typical approach for tree-traversal applications is to use *ad hoc*, application-specific sorting heuristics, based on a programmer's understanding of the behavior of the tree-traversal algorithm. As a result, there have been several strategies proposed for specific traversal algorithms. For Barnes-Hut alone, researchers have proposed sorting using space-filling curves [2], Z-curves [9], orthogonal bisection [26], or the structure of the Barnes-Hut tree itself [28]. For ray tracers, researchers have suggested various ray-reorganization techniques [21, 20, 19, 1, 18]

Rather than devising new sorting strategies for each new traversal algorithm, several researchers have looked at using the past behavior of computations to predict their future tree accesses, and hence dynamically schedule them with minimal application-specific knowledge [31, 22, 13, 12]. Most directly relevant, as they target the same types of algorithms as this paper, is Jo and Kulkarni's *traversal splicing* work [13, 12]. Traversal splicing operates by tracking each traversal's behavior during execution. Each traversal is partially executed until it either truncates its execution at a node in the tree, or reaches some pre-specified maximum depth in the tree. Traversals that are truncated at the same node in the tree (including those that make it to the pre-specified maximum depth) are considered similar, and a new execution order is constructed based on this information. Then all of the traversals are again partially executed until they truncate again or reach another pre-determined stopping point, and the process repeats.

Traversal splicing is based on the insight that traversals that truncate at the same part of the tree are behaving similarly, and hence are likely to behave similarly in the future. Unfortunately, traversal splicing requires very careful bookkeeping, monitoring of traversals, sorting, and interleaving the execution of the traversals with



(a) top tree

(b) point traversal

Figure 2: Tree traversal algorithm

the highly irregular scheduling process. As a result, traversal splicing incurs noticeable runtime overhead [13][1], and is very *ill-suited* to execution on GPUs.

Hence, we are left with a dilemma: known scheduling approaches are either application-specific, or require highly-irregular computation that is poorly matched to GPUs' SIMT execution model. In the next section, we present a novel hybrid scheduling approach that splits the tasks of scheduling and execution, and hence is substantially simpler than prior dynamic scheduling approaches, incurs less runtime overhead, and is well-suited to mapping to GPUs.

4. DESIGN

This section describes our hybrid CPU-GPU scheduling strategy, a novel scheduling and execution technique for tree traversal algorithms that is both general and automatic. We do not rely on any application-specific or semantic knowledge. Instead, our technique uses two GPU kernels: one that runs a portion of the traversal code on the GPU while inspecting the behavior of individual traversals. The CPU then uses this information to dynamically reorder the traversals so that when the second kernel is called, threads grouped into warps perform similar work, improving SIMT efficiency. This strategy is, essentially, an instance of the inspector-executor model [27], where the initial GPU kernel acts as the inspector, the CPU is used to perform the re-scheduling, and the second GPU kernel acts as the executor. The key phases of the technique are:

1. *Profiling.* A carefully constructed GPU kernel runs a small portion of every traversal in the algorithm to collect behavioral information that is used during scheduling. (Section 4.1).

2. *Scheduling.* The CPU analyzes the profiling information and groups threads into different buckets. Threads in one bucket are more likely to access the same branches of the tree and hence exhibit better locality. (Section 4.2).

3. *Execution.* This schedule is then used to execute a second GPU pass that performs the rest of the traversal, using an optimized kernel (Section 4.3).

In Section 4.4, we argue that this scheduling strategy is sound.

[1]Though this overhead is often mitigated by gains in locality.

```
1  void profiling(node root, point pt) {
2    stack stk = new stack();
3    stk.push(root);
4    while (!stk.is_empty()) {
5      root = stk.pop();
6      if (!can_correlate(root, pt))
7        continue;
8      // update information here
9      if (is_leaf(root)) {
10       matrix[pt.id][root.id] = root.id;
11     } else {
12       stk.push(root.right);
13       stk.push(root.left);
14     }
15   }
16 }
```

Figure 3: Profiling for point correlation

	1	2	4	8	9	5	10	11	3	6	12	13	7	14	15
A	1	2	4	8	9				3	6			7	14	15
B	1	2	4			5	10	11	3	6	12	13	7		
C	1	2	4	8	9				3	6	12	13	7		
D	1	2	4			5	10	11	3	6	12	13	7	14	15
E	1	2	4	8	9				3	6			7	14	15
F	1	2	4			5	10	11	3						
G	1	2	4	8	9	5	10	11	3						
H	1	2							3	6	12	13	7	14	15
Result				ACEG			BDFG				BCDH			ADEH	

Figure 4: Scheduling matrix for point correlation

4.1 Profiling

In the profiling stage, we run a GPU kernel that performs each traversal *only in the top half of the tree*. Because the top portion of the tree is small relative to the rest of the tree, this profiling step accounts for a very small proportion of the overall computation, and hence even if the points are poorly scheduled, the overall impact on performance is small.

Figure 2(a) shows the top portion for a binary tree, with nodes indexed in heap order. In the following sections, we call this top portion the *top-tree*. Figure 2(b) shows the traversals of eight points using the algorithm shown in Figure 1. The vertical axis shows different input points that would traverse the tree, while the horizontal axis records which nodes a point may visit. Each circle in the diagram represents a computation step during execution. Note that each point does not visit all the nodes.

During profiling, the traversal of each thread is traced and recorded into a *scheduling matrix*. The scheduling matrix has one row for each point, and one column for each node in the top tree. When a point visits a node, the appropriate cell in the table is marked. Figure 3 shows how the point correlation code is augmented with this profiling data. Note that the stack manipulation in lieu of recursive calls to visit different portions of the tree is due to the autoropes transformation (Section 2.2). Figure 4 shows the resulting scheduling matrix for the set of traversals in Figure 2(b). Note, we only need the leaf node columns, which are marked in gray, for scheduling. This matrix is transferred back to the CPU for use during scheduling, as described in the next section. The profiling overhead is discussed in Section 6.

Guided traversals In some algorithms, such as nearest neighbor, the particular order a point visits nodes is governed by point-specific data. For example, based on characteristics of a point, one point might visit the tree root's left child before its right, while another might visit the right child before the left. Following Goldfarb *et al.*'s terminology, we call algorithms that have this property *guided* traversals, in contrast to algorithms like point correlation that are *unguided* [7]. Note that whether a traversal is guided or not can be determined by a simple static analysis that determines whether the order of recursive calls is control dependent on any point-specific data.

```
1  void profiling(node root, point pt) {
2    stack stk = new stack();
3    stk.push(root);
4    int index = 0;
5    while (!stk.is_empty()) {
6      root = stk.pop();
7      if (!can_correlate(root, pt))
8        continue;
9      // update information here
10     if (is_leaf(root)) {
11       matrix[pt.id][index++] = root.id;
12     } else {
13       if (closer_to_left(root, pt)) {
14         stk.push(root.right);
15         stk.push(root.left);
16       } else {
17         stk.push(root.left);
18         stk.push(root.right);
19       }
20     }
21   }
22 }
```

Figure 5: Profiling for nearest neighbor

Because the traversal order of each point in a guided traversal is different than in an unguided traversal, our profiling code must also encode that traversal order. Figure 5 shows that code. Note, first, that the particular order of tree traversal is determined by the predicate `closer_to_left` (line 13), making the traversal guided. Second, the structure of the scheduling matrix is now different. Rather than each column representing a particular node in the tree, column *i* represents the *i*th node visited by a particular point. Line 11 shows how the particular traversal order of a given point is encoded into the matrix.

4.2 Scheduling

4.2.1 Scheduling unguided traversals

The profiling matrices we generate in the profiling step provide information that lets us reason about the behavior of the points. In Table 4, we see that points **ACEG** all visit the leaf nodes ⑧ and ⑨ of the top tree. Since all of these points reached the same leaf nodes of the top tree, it is more likely that they will behave similarly as they traverse the rest of the tree. Similarly points **BDFG** all visit nodes ⑩ and ⑪, so we expect them to behave similarly in the rest of the tree. (Note that point **G** shows up in both groups; we conclude that it behaves somewhat similarly to both groups of points).

We can scan the *columns* of the scheduling matrix to construct scheduling buckets. The last row of Table 4 shows the resulting buckets. Note that even though the top tree has eight leaf nodes, there are only four buckets. This is because sibling leaf nodes have the same information in our example.

These buckets represent points that have some similarity of behavior. Scheduling according to this information can greatly improve locality. However, points in the same bucket are still unoptimized. Since each bucket may contains millions of points, the divergence in a single bucket can still be considerable. Indeed, intra-bucket scheduling can be even more important than inter-bucket grouping. We hence perform intra-bucket scheduling while building each bucket.

At a high level, the idea behind intra-bucket scheduling is simple. In the result row of Table 4, **A** and **E** appear in two buckets together, while they only appear in one bucket with **D** and **H**. Hence, in the

bucket where all four points appear, ⑭, we would like to execute **A** and **E** consecutively, and then **D** and **H**. In other words, nodes that appear in several buckets together should be considered more similar, and hence scheduled together. Unfortunately, building the buckets and then searching for such similarity is very expensive. We thus use an intra-bucket scheduling algorithm that orders the points on the fly.

Intra-bucket scheduling Rather than treating the construction of each bucket as a separate process, we consider these steps a continuous process: we use the outcome of building one scheduling bucket as a guide to the construction of the next. The procedure is illustrated in Figure 6. The vector *sandpile* records scheduling result of each scheduling step. Before first step, it is initialized with original input points's index, from **A** to **H**.

In the scheduling process executed on node ⑧, *sandpile* is filtered into two subsets: *taken* and *untaken*. The *taken* subset contains points that would traverse node ⑧ while *untaken* collects the rest. Then we join *taken* and *untaken* together and take the new formed *sandpile* as the input for node ⑩. Notice that the new *sandpile* still has the same elements as the old one, but already contains scheduling information.

In node ⑩, we check whether points visit this node in the order they lay in the *sandpile* which is transferred from node ⑧. Point **A**, **C** and **E** show no record in matrix, so we push them into *untaken* subset. Point **G**, **B**, **D** and **F** visited node ⑩ and should be pushed into *taken*. After we insert point **H** into *untaken*, we join the two subsets again and create another new *sandpile*. Figure 7 shows the pseudo-code of the partition-join process. At the end of the scheduling, the *sandpile* is re-arranged as **DHAEBCGF**. Points **D** and **H** are put together because both of them visit node ⑭, as are **AE** (node ⑧), **BC**(node ⑫) and **GF**(node ⑩).

By continuously refining the schedule as we build the scheduling buckets, the *sandpile* eventually yields a final schedule that captures the similarity of points across multiple scheduling buckets. This process is quite efficient, as building a schedule for one million points across 256 scheduling buckets takes just a few tenths of a second.

4.2.2 Scheduling guided traversals

The scheduling matrix for guided traversals keeps both the node ID and traversal order for each point. Points' traversal are represented by a sequence of numbers, like the gene. What we would like to do is group together points with similar sequences together, as they will perform similar work. To do this, we iterate through each traversal sequence, partitioning points based on the order in which they visit nodes.

In Figure 8, the input points are filtered into four buckets based on the first leaf node they access: points **ABFH** would traverse node ⑧ first, while **CE** prefer node ⑩, **D** goes to node ⑫ and **G** visits node ⑮. Node ⑧'s bucket contains so many elements that we need to schedule them in finer granularity. In the second step[2], both point **A** and **F** traverse node ⑪ first then node ⑩ while **B** and **H** visit node ⑩ then ⑪. Point **A** and **F** present more similarity and thus should be arranged closer than others, so as point **B** and **H**. We could implement the same scheduling iteration multiple times until the end of the row, but we need to consider the cost. If there are only a few points left in a bucket, the divergence would be tolerable. We cut off our recursive scheduling process when the number of points is below some limits. A common example of such limit is thirty-two, the number of threads in a warp (line 17 in Figure 9).

Figure 9 shows the pseudo-code for guided traversal optimization. Assume that the number of input points is **npoints**, and the *top tree* has **nnodes** leaf nodes. We initialize a vector with **npoints** well-aligned elements that presents the original order of input points, and an empty sequence as the output. According to the first traversed node, we distribute **npoints** points into **nnodes** buckets. For the points in a bucket, we repeat the schedule until the number of points falls below a threshold. After all the points in the one bucket is well sorted, the program moves to the next bucket.

Since the number of nodes that a point would traverse varies, the length of a row may also change. In above pseudo-code, each node has an ID from 1 to **nnodes**, while the cells in matrix are initialized as zero. When the entry in a row changes from non-zero to zero, that means the traversal of a thread finishes. The thread that ends up earlier than others should be removed from further scheduling. We insert it in the bottom of the output buffer (line 12).

4.3 Execution

The final stage of our process is the execution phase. We simply run the original GPU kernel (with the addition of our optimized lockstep skeleton, described in the next section) for the traversal algorithm using the order of points determined during scheduling. Since this order of points is based on points that we expect to behave similarly, this has an analogous effect to when Goldfarb *et al.* ran their kernels on *sorted* input points.

Note that we re-run the *entire* traversal algorithm using the new schedule of points. In the profiling step, we execute the tree traversal algorithm for certain depth. Since the profiling work performs some of the work of the original algorithm, we could store the computation result of profiling, and restore from these break points in execution step. However, this requires communicating tremendous amounts of data back from the GPU after the profiling step. Instead, updates to points that occur during profiling on the GPU are *not* communicated back to the CPU. All we communicate back is the scheduling matrix.

While this strategy does result in redundant work, as the top tree is traversed a second time during the execution phase, the amount of time spent in the top tree overall is negligible, and hence this has very little impact on performance. Indeed, our experiments have shown that the expense of communicating point data back and forth and restore from the partial traversal results can actually *slow down* performance.

4.4 Correctness

Correctness is far more important than performance for any kind of code transformation and scheduling. We argue that our hybrid-scheduling is sound. Note that although scheduled code walks through the tree in a different order from original, for each point, scheduling does not change the nodes it visits, nor the order it visits them. For a given node, points that visit it are the same set, and the place where the value is updated is also preserved.

5. IMPLEMENTATION

This section describes an alternate lockstepping implementation that performs far fewer memory accesses than Goldfarb *et al.*'s original lockstep kernel, substantially improving performance.

Goldfarb *et al.*'s lockstep kernel operates as follows: each warp has a bit vector with one entry per thread called the *mask vector*. If a thread wants to truncate at a node, its bit in the mask vector is set. If any thread in the warp *does not* truncate (*i.e.*, not all bits in the mask vector are set), all threads continue recursion. The mask vector is used to supress the computation of threads that are

[2]We define the traversal of two sibling nodes with the same parent as a step. A step that visits node ⑧ then node ⑨ is recognized as different from a step traverses node ⑨ then node ⑧.

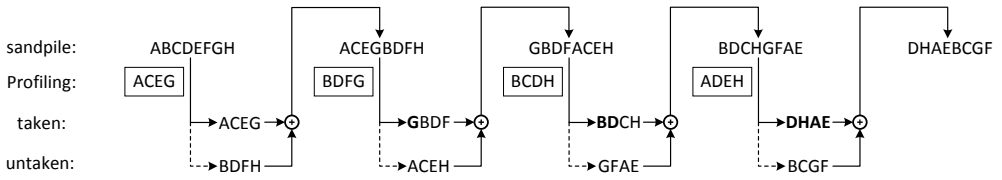


Figure 6: Intra-bucket scheduling example

```
1  void schedule() {
2    vector<int> *sandpile = {A, B, ..., H};
3    foreach(Node n in nodes) {
4      vector<int> *taken, *untaken;
5      for (i = 0; i < npoints; i ++) {
6        point_id = sandpile[i].id;
7        if(matrix[n->id][point_id])
8          taken->push_back(point_id);
9        else
10         untaken->push_back(point_id);
11     }
12     foreach (j in untaken)
13       taken->push_back(j);
14     delete sandpile, untaken;
15     sandpile = taken;
16   }
17 }
```

Figure 7: Intra-bucket scheduling code

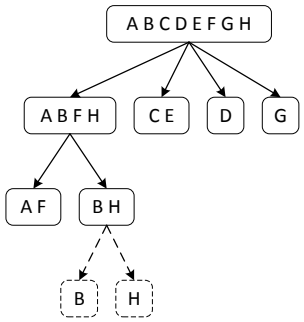


Figure 8: Guided optimization example

```
1  int *buffer = new int [npoints];
2  vector<int> sandpile = {A, B, ..., H};
3  schedule(sandpile, buffer, 0, 0);
4
5  void schedule (sandpile, buf, offset, index) {
6    clusters = new vector<int> [nnodes];
7    for (i  = 0; i < sandpile.size(); i ++) {
8      pos = sandpile[i];
9      temp = matrix[index][pos];
10     if(!temp)
11       cluster[temp].push_back(sandpile[i]);
12     else
13       buffer[offset++] = sandpile[i];
14   }
15   for(j = 0 ; i < nnodes; j ++) {
16     if(clusters[j].size() == 0) return;
17     if(index >= nnodes || clusters[j].size() <= 32)
18       for(k = 0; k < clusters[j].size(); k ++)
19         buffer[offset ++] = clusters[j][k];
20     else
21       schedule(cluster[j], buffer, offset, index + 2);
22   }
23 }
```

Figure 9: Guided optimization code

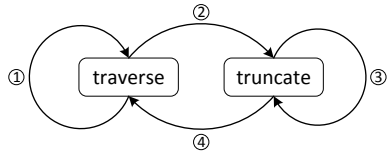


Figure 10: FSM state transfer

"carried along" with the warp even though they wanted to truncate higher in the tree.

While this approach successfully implements lockstepping, it has a performance penalty. The mask vector needs to be preserved throughout a warp's traversal. As a result, the mask vector is treated as an argument to the recursive method, and hence is pushed and popped as part of the function call stack, or, in the case of autoropes, the rope stack. Because this stack can get quite large, it is stored in slower memory. Hence, the repeated pushes and pops of the mask vector introduces substantial overhead.

We propose a new lockstep kernel that, rather than using a mask vector that must be preserved on the (function or autoropes) stack can instead be maintained in a register, dramatically reducing memory accesses. While the warp traverses the tree, each thread conceptually runs a finite state machine (FSM) with two states: *truncated* and *traversing*. A thread only performs computation while it is in traversing state. Figure 10 shows this state diagram and Figure 11 shows the pseudocode for the lockstep kernel, indicating when state transitions are taken.

All the threads in a warp are initially in the traversing state. When a thread reaches a condition that leads it to truncate its traversal (line 15), it transitions to the truncate state. The question is, when does the thread transition back to the traversing state? To keep track of this, the thread stores the current depth of the stack, `sp` in a local variable `critical`. When the stack depth returns to `critical`, the thread transitions back to traversing state (line 7).

The FSM lockstep kernel has two key features. First, as long as a thread is truncated, it performs no computation (line 30). Second, unlike Goldfarb *et al.*'s lockstep kernel, we need not track a mask vector. Instead, each thread simply tracks whether it is truncated. Nevertheless, all the threads are still carried along through the tree, preserving the memory coalescing benefits of lockstep traversal.

We note one other feature of the lockstep kernel. If some threads in the warp want to take one path through the traversal, while other threads want to take a different path, lines 20–26 determine which direction the majority of the non-truncated threads want to go, and send all the threads in that direction. This behavior ensures that all threads in the warp *dynamically* select a single path through the tree.

Example In previous unguided traversal example, the warp diverges at node ④ when the depth of stack is 2. Point **B** is truncated (*critical = 2*) and wants to visit node ⑤, but the warp wants to traverse the subtree rooted at node ④. The warp pushes node ⑧ and ⑨ into the *stack* and increases the *sp* to 4. The *stack* keeps

```
__global__ void gpu_kernel() {
  flag = 1; critical = INT_MAX; cond = 0;
  sp = 1; //current depth of the warp-shared ropes stack
  for(pidx = blockIdx.x * blockDim.x + threadIdx.x;
    pidx < npoints; pidx += blockDim.x * gridDim.x) {
    while (sp >= 1) {
      if (sp <= critical)
        flag = 1;                    // transition 4
      sp --;
      if (flag) {
        //compute condition
        cond = ...;
        if (!__any(cond))
          continue;
        if (!cond) {                 // transition 2
          flag = 0;
          critical = sp;
        } else {                     // transition 1
          // compute branch condition
          cond_left = ...;
          cond_right = ...;
          vote_left = __ballot(cond_left);
          vote_right = __ballot(cond_right);
          num_left = __popc(vote_left);
          num_right = __popc(vote_right);
          if (num_left > num_right)
            // stack operation
          else
            // stack operation
        }
      } else {}                      // transition 3
    }
  }
}
```

Figure 11: FSM lockstep kernel

growing when node ⑧ or node ⑨ also has children, but the node ⑤ cannot be visited unless all these nodes above it have been popped. In other words, the traversal of point **B** would never resume before *sp* drops below *critical* value.

6. EVALUATION

6.1 Methodology

Platform We evaluate our benchmarks on a server with two AMD Opteron 6164 HE Processors, each of which contains 12 cores running at 1700MHz. The GPU is an nVidia Tesla K20C with 5120 MB GDDR5 memory and 2496 CUDA cores. The system has 32GB system memory and runs on Red Hat 6.6 with Linux kernel v2.6.32.

Benchmarks We evaluate our scheme on six benchmarks:

Point Correlation (PC) is an important algorithm in bioinformatics and data mining field. The two-point correlation can be computed, for each point in a data set, by traversing a kd-tree to count the number of other points that fall within a certain radius.

Nearest Neighbor (NN) finds the nearest neighbors of points in a metric space. NN builds a kd-tree over a set of input points. It then takes a set of *query* points, and for each query point traverses the kd-tree to find its nearest neighbor.

k-Nearest Neighbor (kNN) is a non parametric instance-based learning algorithm widely used for classification and regression. Unlike NN, which finds the nearest neighbor of a query point, kNN finds the *k* nearest neighbors [4].

Ball Tree (BT) is a variation of nearest neighbor that uses ball trees, where the multi-dimensional space is partitioned by hyperspheres.

Vantage Point (VP) is a variation of nearest neighbor search that uses vantage point trees rather than kd-trees: subspaces are split according to distance from a chosen vantage point.

Barnes-Hut (BH) performs an n-body simulation [3]. BH recursively divides the set of n bodies into groups by sorting them in an octree. Then each body traverses the tree to calculate the gravity acting upon it. We replace the force computation part of the original code with our GPU variants, but leave the remainder of the code the same. We time the force computation phase of a single iteration.

Inputs The first five benchmarks are evaluated with four inputs: Covtype, Mnist, and Rand_Dim7, each of which contains 400,000 7-dimensional points, and Geocity, which contains 400,000 2- dimensional points. For the four nearest-neighbor problems (NN, kNN, VP, BT)[3], we partition the 400,000 points into two subsets, S_1 and S_2. S_1 is used to build the tree, while the points in S_2 are the query points[4]. BH is evaluated with two 1-million bodies inputs: Plummer, which uses the Plummer model to generate bodies, and Rand_Dim3, with uniformly randomly generated bodies.

Evaluation methodology For each benchmark, we evaluate five variants: the original kernels (Goldfarb *et al.*'s original lockstep code) and our new FSM lockstep kernel on both unsorted and sorted inputs, and our hybrid scheduling approach on the unsorted input. The hybrid scheduling variant uses the FSM lockstep kernel. We choose the original kernel with unsorted input as the baseline, and take the application specific sorting cost as the baseline overhead.

File I/O, tree building, etc. are not targets for optimization, so we do not include those components in our timing. We measure the runtime spent in GPU execution of tree traversals, as well as time spent in profiling and scheduling.

6.2 Results

We begin by comparing the performance of the different implementations of tree traversal algorithms. Figure 12 shows the speedup of all of the implementations over the baseline. The columns in every benchmark are arranged in the following order: the original lockstep with unsorted input, the FSM lockstep with unsorted input, the original lockstep with hand-sorted input, the FSM lockstep with hand-sorted input and the FSM kernel with hybrid scheduling strategy. The percentage value presents the speedup of hybrid scheduling over the baseline.

The primary comparison is between Goldfarb *et al.*'s baseline (Original lockstep with unsorted) and our full system (Hybrid scheduling with unsorted), which implements both our hybrid scheduling algorithm and our optimized traversal kernel. We see that our hybrid scheduling algorithm yields, even in the worst case, a 1.76× speedup over the baseline. ***On average, our technique is 5.96× faster than the baseline.*** Hence, when presented with the original inputs, our new approach delivers enormous gains over the previous, best-known general implementations.

We also isolate the benefits of our optimized kernel by using the original schedule, but using the optimized kernel (FSM lockstep with unsorted). We see that across the board, the optimized kernel is faster than the original kernel, though some times the gains are minimal. Ultimately, we can conclude that most of the performance gains from our new techniques come from the optimized schedule.

Finally, to show the effectiveness of our automatic scheduling approach over a hand-tuned approach that uses application-specific sorting routines to derive schedules for each application, we compare our technique (Hybrid scheduling) to using the optimized kernel on *sorted* inputs (FSM lockstep with sorted). We see that, with

[3]The implementation of BT we use does not work with two-dimensional data, so we do not evaluate BT on the Geocity input.

[4]In the results we present, S_1 and S_2 each contain 200,000 points. We evaluated different random subsets of points, as well as different sizes for S_1 and S_2 and found qualitatively similar results.

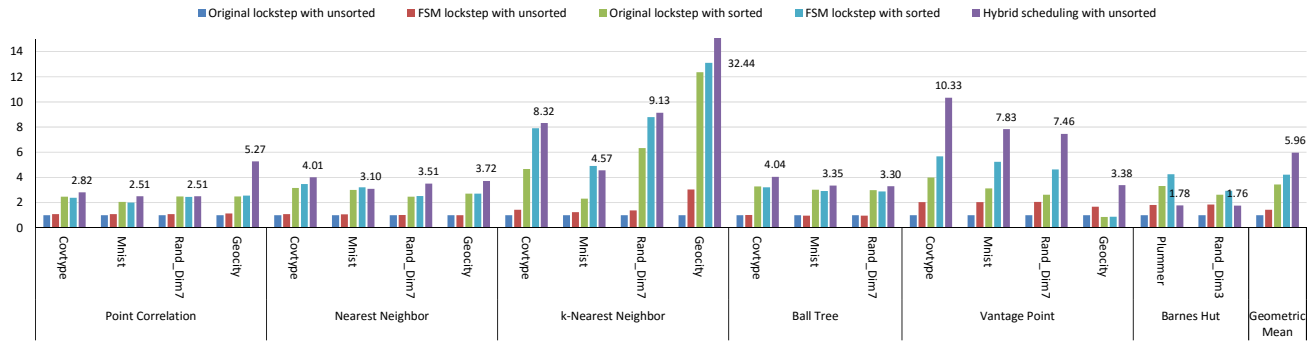


Figure 12: Speedup comparison

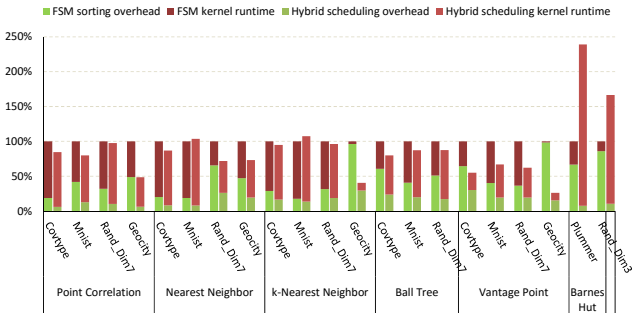


Figure 13: Overhead cost ratio comparison

the exception of Barnes-Hut, *our technique is always faster*. In other words, on average, **our automatic technique is 1.41× faster than hand-tuned implementations**!

Barnes-Hut is the one outlier in our comparisons to the baseline (yielding the smallest improvement) and to hand-tuned schedules (where the hand-tuned schedules are faster). In other words, our hybrid scheduling is less effective for Barnes-Hut than for other benchmarks. In Barnes-Hut, traversals each visit a larger fraction of the tree than in other benchmarks. In other words, there is relatively *less* divergence between traversals in the tree. Because our scheduling algorithm relies on thread divergence as a signal for sorting, Barnes-Hut's lack of divergence limits the effectiveness of our scheduling. Sections 6.3.2 and 6.3.4 explore this result in more detail.

6.3 Performance breakdowns

The following subsections explore the performance results in more detail. First, we investigate the performance of our techniques relative to the hand-tuned techniques, studying both the cost of scheduling and the cost of execution. Second, we explore how our techniques' effectiveness varies with the depth of the profiling phase. Third, we isolate the performance of our optimized lockstep kernel with Goldfarb *et al.*'s original traversal kernels. Finally, we directly assess the effects of hybrid scheduling on divergence.

6.3.1 *Scheduling time ratio*

Figure 13 shows the ratio that the overhead of hybrid scheduling strategy and hand-tuned sorting over the whole runtime. In our hybrid scheduling, the extra work includes the profiling execution, matrix transmission from GPU to CPU, CPU scheduling, and final result transmission back to the GPU[5]. For hand-tuned sorting, the extra work is only the application-specific sorting procedure.

Our hybrid scheduling strategy shows consistent advantages over application-specific sorting in the majority of benchmarks. We note that in many cases, the advantage of our hybrid scheduling strategy comes not from better schedules, but from the fact that the overhead of sorting in the hybrid strategy is far smaller. This result justifies our decision to perform profiling on the GPU, where the parallelism advantages of GPU execution win out.

6.3.2 *Profiling depth sensitivity analysis*

Our hybrid scheduling tunes the execution order of the original input sequence according to the information collected during the execution of top tree. That means that the larger the top tree is, the more knowledge our scheduling may learn, and thus the better the resulting schedule, at the cost of additional inspection overhead. Balancing the benefits of more profiling information against profiling overhead is a classic problem in any profile-guided optimization.

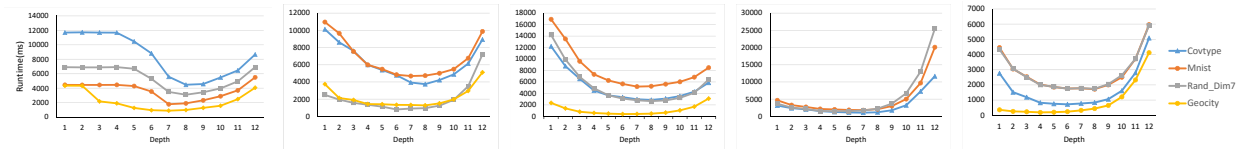
Figure 14 shows the sensitivity of our benchmarks' performance to how deep in the tree the inspection phase runs for the binary tree–based benchmarks (*i.e.*, all but BH). In all cases, we see the expected U-shaped curves: as depth increases, more profiling information leads to better schedules, until the overhead of profiling outweighs the scheduling benefits. Interestingly, we see that the optimal depth is roughly the same in all cases, regardless of benchmark or input. This suggests that for binary tree–based benchmarks, the profiling depth can be set in an application- and input-independent manner.

We further explore the overhead/effectiveness tradeoff in Figure 15, which separately measures the runtime of the inspection and execution phases. Figure 15(a) shows, for kNN, the expected behavior: as profiling depth increases, kernel execution time decreases, but overhead increases. Figure 15(b) shows the same breakdown for BH. Here, we see further evidence for why our hybrid scheduling underperforms for BH. While the trends match the other benchmarks, BH's octree means that overhead increases dramatically with even a small increase in profiling depth. Hence, it is impractical to try and capture more profiling information, resulting in a poorer-quality schedule.

6.3.3 *FSM kernel versus original kernel*

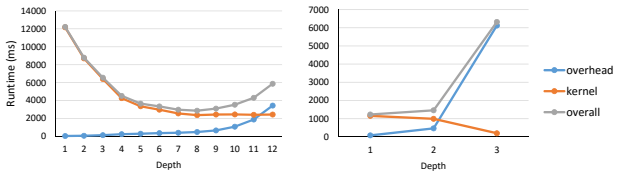
Figure 12 lets us isolate the performance of our optimized lockstep kernel. With unsorted input, the FSM lockstep kernel presents similar or slightly better performance over baseline. The average speedup for FSM lockstep is 1.22×. When the input point set is

[5]Note that although the profiling matrix is large in size, we may overlap its transmission with profiling kernel execution through CUDA streams.



(a) PC (b) NN (c) kNN (d) BT (e) VP

Figure 14: Depth sensitivity analysis



(a) kNN_Cov (b) BH_Rand_Dim3

Figure 15: Detailed depth sensitivity analysis

Benchmark	Input	Baseline (ms)		Average num of nodes visited by a warp		
		runtime	overhead	Unsorted	Sorted	Hybrid
k-Nearest Neighbor	Covtype	23904	861	147977	15785	31805
	Mnist	24048	861	173048	32002	53574
	Rand_Dim7	24261	870	158046	13321	27931
	Geocity	13526	994	57865	271	1240
Vantage Point	Covtype	8869	1016	105896	11758	10976
	Mnist	13694	1042	150307	36288	33919
	Rand_Dim7	13103	1043	142820	40924	34981
	Geocity	636	711	19178	337	507
Nearest Neighbor	Covtype	14996	850	256560	53813	132900
	Mnist	14681	848	265901	65607	136449
	Rand_Dim7	3335	864	70676	8783	17937
	Geocity	4655	808	131793	21779	103805
Ball Tree	Covtype	4305	806	74660	8263	24373
	Mnist	6081	847	112330	20478	42482
	Rand_Dim7	5004	882	92135	13611	28207
Point Correlation	Covtype	12752	994	256374	76047	80038
	Mnist	4835	987	95624	26021	31664
	Rand_Dim7	7811	1005	156050	37605	54155
	Geocity	4663	881	120388	22223	27312
Barnes Hut	Plummer	4395	680	22106	3346	19847
	Rand_Dim3	2163	630	10797	984	9291

Table 1: Scheduling effects on divergence

hand-sorted, the FSM lockstep kernel shows significant advantage over baseline, and even the original lockstep traversal with the same hand-sorted input. The average speedup for FSM lockstep with hand-sorted input is 4.22×, while the original lockstep with hand-sorted input only provides 3.45× speedup.

6.3.4 Hybrid scheduling effects on divergence

The lockstep kernels, which force traversals to "stay together" in the tree, converts memory divergence (accessing different parts of the tree) into branch divergence (forcing traversals to "pause" while other traversals access a portion of the tree). Thus our hybrid scheduling's effect on reducing branch divergence can essentially addresses both concerns.

In the lockstep kernel, all threads in a warp traverse the same portion of the tree. We can thus evaluate the effectiveness of scheduling by measuring the total number of nodes visited *by a warp*. If a warp contains highly-convergent threads, it will visit fewer nodes than if the threads diverge and want to touch more of the tree. Table 1 shows the results. We see that hybrid scheduling results in a large decrease in the number of nodes visited per warp compared to the baseline: on average, warps scheduled using the hybrid approach traverse *7.15*× fewer nodes than the baseline, a substantial decrease in divergence.

We note that this evaluation shows that application-specific sorting yields even better convergence (about 2× better, on average)—an unsurprising result. However, as discussed in Section 6.3.1, while hand sorting yields more convergence and hence faster kernels, our hybrid approach spends less time in scheduling, yielding better performance overall.

Note that these results further explain hybrid scheduling's performance on BH. Not only is our profile-guided sorting relatively ineffective at reducing divergence (indeed, our 1.76× speedup comes almost entirely from the optimized kernel), application specific sorting is *highly* effective.

7. RELATED WORK

Most of the research work about tree traversal focuses on GPU-only implementation, and is targeted at specific tree traversal algorithms. Foley and Sugerman propose two variations of kd-tree traversal that take advantage of bounding box to eliminate per-ray stack, and rely on kernel masking and scheduling to reduce overhead [6]. Horn *et al.* extends Foley *et al.*'s work by the usage of packet, restarting from lower subtree instead of the root and stack-based restart check [11]. Popov *et al.* develop another stackless kd-tree traversal approaches for GPU ray tracing by adding ropes to leaf nodes and using packets of rays [23].

Hybrid scheduling is mostly used in other irregular applications, such as clustering and map-reduce. Ren *et al.* use a hybrid approach for k-means computing with very large data sets [29]. They execute the map on the GPU, transfer the result back to the CPU, and execute the reduction on the CPU. Rather than assign map and reduce separately, Ravi *et al.* parallelize the computation by splitting the input data set into sections and distribute every section to either CPU or GPU [25]. Ravi and Agrawal also describe a scheduling framework for data parallel loop by configuring application with various behavior patterns for heterogeneous architecture through a *cost* model [24].

Zhang *et al.* propose a generic framework for handling irregular GPU computation called *G-Streamline* [30]. G-Streamline adopts a similar inspector-executor style approach as our hybrid scheduling: the CPU determines the set of data accesses that a set of GPU threads will perform and then remaps data or reorders GPU threads to improve convergence. G-Streamline primarily targets accesses through indirection arrays and relies on knowing (or approximating) which data a GPU thread will access prior to execution. In addition, G-Streamline's rescheduling heuristic relies on each thread's accessing a small amount of data. The tree applications our approach targets do not have these characteristics: we must generate a GPU profiling pass to predict the similarity of threads (unlike G-streamline, inspection is performed on the GPU to avoid high profiling overheads), and then use a more sophisticated scheduling algorithm to effectively handle tree applications.

Most other work on heterogeneous execution has focused on taking a set of tasks and dividing them between the CPU and GPU. Qilin is an API and runtime programming system that automatically maps computation to CPU and GPU cores [15]. It adopts offline profiling to analyze programs and adaptively maps them to

an analytical performance model to determine actual job distribution. Kaleem *et al.* also propose two profiling based scheduling algorithms that automatically partition workload between CPU and GPU [14]. They showed that profile-based scheduling for integrated GPU has very little overhead and presents comparable efficiency with offline model.

8. CONCLUSIONS

We describe a general, application-agnostic scheduling framework for tree traversals that automatically uses partial execution to inspect GPU threads' behaviors, and then uses that information to reorder execution on the CPU prior to re-execution on the GPU. We show that the scheduling problem is NP-hard, and develop optimized version of scheduling according to structural properties of traversal algorithms. We also introduce a new skeleton for GPU kernels that uses a register-level finite state machine to minimize memory access. Our experiments show that our work significantly outperforms the baseline, and can even outperform hand-tuned, application-specific scheduling.

Acknowledgments

The authors would like to thank the anonymous referees for their suggestions and comments. This work was supported in part by an NSF CAREER award (CCF-1150013) and a DOE Early Career award (DE-SC0010295).

9. REFERENCES

[1] Timo Aila and Tero Karras. Architecture considerations for tracing incoherent rays. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 113–122, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[2] Margarita Amor, Francisco Argüello, Juan López, Oscar G. Plata, and Emilio L. Zapata. A data parallel formulation of the barnes-hut method for n -body simulations. In *Proceedings of the 5th International Workshop on Applied Parallel Computing, New Paradigms for HPC in Industry and Academia*, pages 342–349, 2001.

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

[4] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18:509–517, September 1975.

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

[6] Tim Foley and Jeremy 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.

[7] Michael Goldfarb, Youngjoon Jo, and Milind Kulkarni. General transformations for gpu execution of tree traversals. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 10:1–10:12, New York, NY, USA, 2013. ACM.

[8] Johannes Gunther, Stefan Popov, Hans-Peter Seidel, and Philipp Slusallek. Realtime 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.

[9] Hwansoo Han and Chau-Wen Tseng. Exploiting locality for irregular scientific codes. *IEEE Trans. Parallel Distrib. Syst.*, 17:606–618, July 2006.

[10] Michael Hapala, Tomas Davidovic, Ingo Wald, Vlastimil Havran, and Philipp Slusallek. Efficient Stack-less BVH Traversal for Ray Tracing. In *Proceedings 27th Spring Conference of Computer Graphics (SCCG) 2011*, pages 29–34, 2011.

[11] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree gpu raytracing. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, I3D '07, pages 167–174, 2007.

[12] Youngjoon Jo, Michael Goldfarb, and Milind Kulkarni. Automatic vectorization of tree traversals. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, PACT'13, pages 363–374. IEEE, 2013.

[13] Youngjoon Jo and Milind Kulkarni. Automatically enhancing locality for tree traversals with traversal splicing. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 355–374, New York, NY, USA, 2012. ACM.

[14] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Brian T. Lewis, Chunling Hu, and Keshav Pingali. Adaptive heterogeneous scheduling for integrated gpus. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 151–162. ACM New York, NY, USA, August 2014.

[15] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 45–55. ACM, December 2009.

[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] Bochang Moon, Yongyoung Byun, Tae-Joon Kim, Pio Claudio, Hye-Sun Kim, Yun-Ji Ban, Seung Woo Nam, and Sung-Eui Yoon. Cache-oblivious ray reordering. *ACM Trans. Graph.*, 29(3):28:1–28:10, July 2010.

[19] Paul Arthur Navratil. *Memory-efficient, scalable ray tracing*. PhD thesis, 2010.

[20] Paul Arthur Navratil, Donald S. Fussell, Calvin Lin, and William R. Mark. Dynamic ray scheduling to improve ray coherence and bandwidth utilization. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, RT '07, pages 95–104, Washington, DC, USA, 2007. IEEE Computer Society.

[21] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 101–108, 1997.

[22] Venkata K. Pingali, Sally A. McKee, Wilson C. Hseih, and John B. Carter. Computation regrouping: Restructuring

programs for temporal data cache locality. In *Proceedings of the 16th International Conference on Supercomputing*, ICS '02, pages 252–261, New York, NY, USA, 2002. ACM.

[23] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum*, 26(3):415–424, September 2007. (Proceedings of Eurographics).

[24] Vignesh T. Ravi and Gagan Agrawal. A dynamic scheduling framework for emerging heterogeneous systems. In *Proceedings of the 2011 18th International Conference on High Performance Computing*, pages 1–10. IEEE Computer Society Washington, DC, USA, 2011.

[25] Vignesh T. Ravi, Wenjing Ma, , and Gagan Agrawal. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In *Proceeding of the 24th ACM International Conference on Supercomputing*, pages 137–146. ACM, June 2010.

[26] John K. Salmon. *Parallel hierarchical N-body methods*. PhD thesis, California Institute of Technology, 1991.

[27] Joel H. Saltz and Ravi Mirchandaney. Run-time parallelization and scheduling of loops. In *IEEE Transactions on Computers*, volume 40, pages 603–612. IEEE, May 1991.

[28] Jaswinder Pal Singh, Chris Holt, Takashi Totsuka, Anoop Gupta, and John 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, 1995.

[29] Ren Wu, Bin Zhang, and Meichun Hsu. Clustering billions of data points using gpus. In *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*, pages 1–6. ACM, May 2009.

[30] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng 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*, pages 369–380. ACM New York, NY, USA ©2011, March 2011.

[31] Xingbin Zhang and Andrew A. Chien. Dynamic pointer alignment: Tiling and communication optimizations for parallel pointer-based computations. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '97, pages 37–47, New York, NY, USA, 1997. ACM.