REGTT: Accelerating Tree Traversals on GPUs by Exploiting Regularities

Feng Zhang[1], Peng Di[1], Hao Zhou[1], Xiangke Liao[2], and Jingling Xue[1]

[1]School of Computer Science and Engineering, UNSW, Australia
[2]School of Computer Science and Technology, NUDT, China

***Abstract*—Tree traversals are widely used irregular applications. Given a tree traversal algorithm, where a single tree is traversed by multiple queries (with truncation), its efficient parallelization on GPUs is hindered by branch divergence, load imbalance and memory-access irregularity, as the nodes and their visitation orders differ greatly under different queries.**

We leverage a key insight made on several truncation-induced tree traversal regularities to enable as many threads in the same warp as possible to visit the same node simultaneously, thereby enhancing both GPU resource utilization and memory coalescing at the same time. We introduce a new parallelization approach, REGTT, to orchestrate an efficient execution of a tree traversal algorithm on GPUs by starting with BFT (Breadth-First Traversal), then reordering the queries being processed (based on their truncation histories), and finally, switching to DFT (Depth-First Traversal). REGTT is general (without relying on domain-specific knowledge) and automatic (as a source-code transformation). For a set of five representative benchmarks used, REGTT outperforms the state-of-the-art by 1.66x on average.

I. INTRODUCTION

Irregular applications are widely used in the real world. In addition to graph-based applications [1], [2], another important subclass includes tree traversal algorithms. Operating on a tree, tree traversal is the process of visiting every tree node exactly once (unless truncated otherwise) in a particular order. In this paper, we focus on tree traversal algorithms, where a single tree is traversed multiple times under multiple input queries (i.e., by multiple points). Such algorithms are ubiquitous in the real world. Some representative examples are the Barnes-Hut tree traversal in physics and astronomy (e.g., in N-body simulations) [3], the k-dimensional tree traversal in data mining and graphics (e.g., in Point Correlation solvers) [4], and the vantage-point tree traversal in machine learning (e.g., in Nearest Neighbor classification) [5].

Compared with regular applications [6], [7], [8], [9], [10] that exhibit predictable memory accesses, such repeated tree traversals (with truncation) are challenging to parallelize on GPUs. As the nodes and their visitation orders are highly input-dependent and thus vary greatly across the queries, a simple-minded solution would result in low GPU resource utilization caused by branch divergence and load imbalance as well as long memory latency caused by memory-access irregularity. On the other hand, as different queries traverse a tree independently, repeated tree traversals are well-suited to SIMT parallelization on massive multi-threading GPUs due to their high-performance and power efficiency.

AUTOROPES [11] represents the only solution for parallelizing general-purpose tree traversal algorithms on GPUs. AUTOROPES parallelizes an algorithm by applying DFT (Depth-First Traversal) in either *lockstep* or *non-lockstep* mode. In *non-lockstep*, AUTOROPES aims to maximize GPU resource utilization by allowing different threads that handle different queries in the same warp to visit different tree nodes at the same time at the expense of branch divergence and memory-access irregularity. In *lockstep*, AUTOROPES makes the opposite tradeoff by ensuring that all the threads in the same warp visit the same tree node at the same time, even though many threads are idle traversing a truncated node or no node at all.

In this paper, we leverage a key insight made on tree traversal regularities induced by tree truncations (i.e., by skipping a subtree rooted at a certain node) across the queries. These regularities are concerned with truncation condition checks at sibling tree nodes, truncation possibilities at different tree levels, and future traversal behaviors of different queries with similar truncation histories. By exploiting these regularities, we introduce a new parallelization approach, called REGTT, to orchestrate an efficient execution of a tree traversal algorithm on GPUs by starting with BFT (Breadth-First Traversal), then reordering the queries being processed (based on their truncation histories), and finally, switching to DFT. By switching from BFT (encountering some but not excessively many tree truncations) to DFT (operating on reordered queries so that queries with similar truncation histories are regrouped into the same warp) at a suitable tree depth, we can potentially enable as many threads in the same warp as possible to visit the same tree node simultaneously. As a result, we can improve potentially both GPU resource utilization (by reducing branch divergence and load imbalance) and memory coalescing (by mitigating memory-access irregularity). To the best of our knowledge, REGTT is the first to sequence BFT and DFT this way, with a runtime query reordering in between.

REGTT is general (without relying on domain-specific knowledge) and automatic (as a source-code transformation). This paper makes the following contributions:

- We describe some important observations about regularities in repeated tree traversals, which enable tree traversal algorithms to be parallelized effectively on GPUs.
- We introduce an effective parallelization approach, REGTT, for tree traversals on GPUs, that can enhance both resource utilization and memory coalescing simul-

Algorithm 1: A Generic Tree Traversal Algorithm

```
Input: Tree $T$, Set $Q$
Procedure TreeTraversal()
1    Let $n$ be the root node of $T$;
2    foreach query $q \in Q$ do
3        Traversal($n$, $q$);

Procedure Traversal(Node $n$, Query $q$)
4    visit($n$); // Performing a node-specific task
5    if !Truncation($n$, $q$) then
6        foreach child node $n'$ of $n$ do
7            Traversal ($n'$, $q$);
```

taneously as well as improve load balance in a warp.

- We have compared REGTT with AUTOROPES [11], the state-of-the-art, on parallelizing a set of five representative benchmarks on GPUs. REGTT outperforms AUTOROPES by 1.66x on average.

The rest of this paper is organized as follows. In Section II, we introduce a generic tree traversal algorithm and discuss some challenges faced for its parallelization on GPUs. In Section III, we describe our key observations about tree traversal regularities, review the state-of-the-art, and finally, discuss the design and implementation of REGTT. In Section IV, we evaluate REGTT by comparing with the state-of-the-art. In Section V, we describe related work. Finally, in Section VI, we conclude the paper and discuss some future work.

II. PROBLEM STATEMENT

A. A Generic Tree Traversal Algorithm

In Algorithm 1, we give a generic tree traversal algorithm performed in the DFT manner. Given a set of queries Q, each query q is answered when it traverses a tree T from its root node. For each node n, $visit(n)$ represents an application-specific task to be performed at n when n is said to be *visited*. $Truncation(n, q)$ is an application-specific condition to be tested to see if further visits of n's child nodes are necessary or not. If not, the subtree rooted at n will be truncated. As a result, the nodes and their visitation orders are highly input-dependent and thus vary greatly across the queries.

Let us consider a Point Correlation solver [4], which collects all the objects close to a given query object with their distances being no more than $\mathcal{R}$ in a k-dimensional space. This is illustrated in Figure 1 (with $k = 2$). Figure 1(a) depicts a 2-dimensional space containing 10 objects, denoted by the 10 nodes in solid circles. Figure 1(b) shows a binary tree built based on the spatial relations among the 10 objects. This is done by dividing recursively the given 2-dimensional space into two halves such that their node counts differ by at most one. Eventually, a leaf node (depicted by a circle) represents an object and a non-leaf node (depicted by a gray circle) represents the geometric center of its descendant leaf objects. The location of a node is represented by its bounding box.

Let us apply Algorithm 1 to process a set $Q = \{A, B, C, D\}$ of four queries depicted in Figure 1(a) with respect to the search tree in Figure 1(b). Their DFT sequences are given in Figure 1(c). Due to tree truncations, some nodes are skipped. Take query A for example. When visiting node ⑤, the distance between A and ⑤ (i.e., the distance between the location of A and the bounding box of the ⑤) is larger than $\mathcal{R}$ as shown, the subtree rooted at ⑤ is truncated. As a result, its two children, ⑥ and ⑦, are not visited.

B. Tree Traversals on GPUs

Repeated tree traversals for multiple queries can be solved in parallel on GPUs in the SIMT fashion as all the queries are handled independently. A Nvidia GPU consists of a number of *streaming multiprocessors (SMs)*, where each *SM* contains a number of cores called *streaming processors (SPs)*. A tree traversal algorithm can be executed by a grid of threads over these cores. Threads are grouped into several *thread blocks*, and assigned to *SMs*. Threads in one block are partitioned into 32-thread *warps*, which are the units of the thread execution.

There are three challenges faced in parallelizing tree traversals on GPUs due to query-dependent tree truncations:

- **Memory-Access Irregularity** The threads that handle different queries in the same warp may visit different nodes at the same time, making it hard to take advantage of *memory coalescing*, a well-known optimization for coalescing a number of simultaneous global accesses from the threads in a warp into a single *memory transaction* to reduce GPU's global memory access overhead.
- **Load Imbalance** Different threads in the same warp may have different workloads due to query-dependent tree truncations, resulting in poor load balance.
- **Branch or Warp Divergence** The threads that handle different queries in a warp may suffer from *branch, i.e., warp divergence* at the truncation test at Line 5 of Algorithm 1, resulting in poor resource utilization.

Therefore, these three challenges must be addressed in order for repeated tree traversals to run efficiently on GPUs.

III. EFFICIENT TREE TRAVERSALS ON GPUS

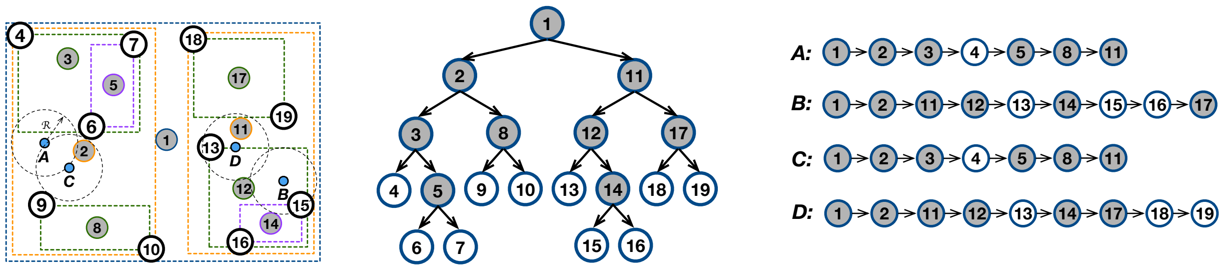
Exploring regularities in repeated tree traversals is critical to ensure that such traversals can be parallelized efficiently on GPUs. We first summarize three types of regularities observed (Section III-A). We then discuss why AUTOROPES [11] fails (Section III-B) but our REGTT approach succeeds (Section III-C) in exploiting these regularities. Finally, we describe our implementation of REGTT (Section III-D).

A. Discovering Tree Traversal Regularities

We describe below three types of regularities regarding (1) truncation condition checks at sibling tree nodes, (2) truncation possibilities at different tree levels, and (3) future traversal behaviors of different queries with similar truncation histories.

Observation 1. *If a tree node is visited (at Line 4 of Algorithm 1), then all its sibling nodes are also visited.*

This observation is validated by the presence of Line 5 of Algorithm 1. If one node is visited, then all its sibling nodes will also be visited. Thus, for the purposes of performing truncation checks at a set of sibling nodes, the relevant data at these sibling nodes will have to be retrieved, together


A: 1 → 2 → 3 → 4 → 5 → 8 → 11
B: 1 → 2 → 11 → 12 → 13 → 14 → 15 → 16 → 17
C: 1 → 2 → 3 → 4 → 5 → 8 → 11
D: 1 → 2 → 11 → 12 → 13 → 14 → 17 → 18 → 19

(a) 2-Dimensional Space (b) Constructed Tree (c) DFT Sequences for Queries $A - D$

Fig. 1: Tree traversals for a point correlation solver.

with some other data, if necessary. This is an obvious fact, but seemingly little noticed or exploited in parallelizing tree traversal algorithms on GPUs by the prior work.

Observation 2. *Tree truncation (at Line 5 of Algorithm 1) is less likely to happen higher in the tree being traversed.*

This follows from how a search tree is constructed for a tree traversal algorithm and how a truncation condition is formulated. In general, a truncation happens at a node only when all its descendant leaf nodes can be neglected without affecting the overall correctness and/or precision of the algorithm. Therefore, the higher a node is in the tree, the less likely the subtree rooted at the node will be truncated.

Observation 3. *Queries with similar truncation histories tend to exhibit similar future traversal sequences.*

Truncation similarity provides a strong hint for spatial similarity. Queries that have similar truncation histories (with sufficiently many truncations) tend to be close together, suggesting that their future traversal behaviors will also be similar.

B. Examining the State-of-the-Art

In addition to domain-specific tree traversal algorithms on GPUs [12], [13], [14], AUTOROPES [11] represents the only approach for automatically parallelizing general-purpose tree traversal algorithms (abstracted in Algorithm 1) on GPUs. Two parallelization strategies, *lockstep* and *non-lockstep*, are considered, with different tradeoffs. In Figure 2, we show how AUTOROPES works for our example in Figure 1 and discuss its limitations. We assume that the warp size is 2.

1) lockstep: As shown in Figure 2(a), AUTOROPES forces all the threads in a warp to visit the same tree node at every traversal step. As a result, there does not exist any non-coalesced memory access (represented by ●) across the nodes. In addition, all memory accesses at the same node are trivially coalesceable (marked by ●). Unfortunately, a thread that handles a query is effectively *idle* (marked by ⬭) when it is forced to traverse a node that is truncated, i.e., not supposed to be visited at Line 4 of Algorithm 1 by the query. By forcing all the threads to run in lockstep, AUTOROPES avoids warp divergence altogether (with no divergent branches represented by ▬). However, the GPU resources can be severely underutilized due to poor load balance, caused by the presence of potentially many idle threads in a warp.

2) non-lockstep: As shown in Figure 2(b), AUTOROPES makes the opposite tradeoff as above. In this case, different threads in a warp can visit their respective nodes as intended (at their own discretion). Thus, AUTOROPES improves GPU resource utilization but at the expense of warp divergence and memory-access irregularity. For example, the warp that handles queries A and B suffers from branch divergence when accessing two nodes, $\textcircled{4}$ and $\textcircled{12}$, at the same time (highlighted by ▬), as well as long memory latency since the memory accesses at the two nodes are unlikely coalesceable (highlighted by ●). While GPU resource utilization has improved (compared to *lockstep*), idle threads still exist at the end of a warp execution due to load imbalance (marked by ⬭).

3) Discussion: AUTOROPES makes opposite tradeoffs in its *lockstep* and *non-lockstep* modes. Neither is superior over the other. A better strategy should tackle the three challenges discussed in Section II-B by improving both GPU resource utilization and memory coalescing simultaneously. This is possible if we can ensure that the threads in a warp can visit simultaneously the same tree node as often as possible.

In Section III-A, we have examined some regularities in repeated tree traversals. By **Observations 1** and **Observations 2**, the threads in a warp should visit all the sibling tree nodes first before their child nodes to reduce potentially warp divergence. By **Observation 3**, a warp should be responsible for handling queries with similar truncation histories.

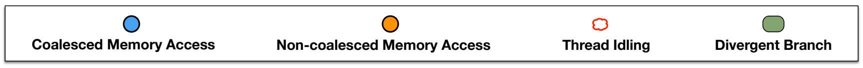
However, AUTOROPES fails to exploit these regularities. In addition, AUTOROPES also suffers from some limitations by performing DFT alone. As the deeper nodes are reached, warp divergence will likely become more frequent. For *lockstep*, GPU resource utilization will be worse due to poor load balance (with more idle threads). For *non-lockstep*, warp divergence will be more severe and global memory access overhead will also increase (due to non-coalesced accesses).

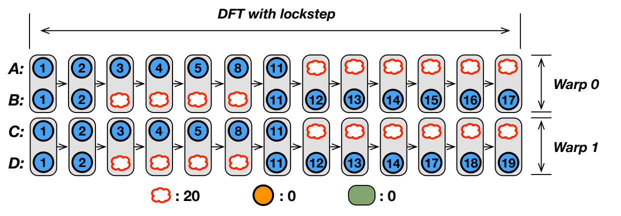
Below we demonstrate how our REGTT approach can overcome these limitations by exploiting tree traversal regularities.

C. Designing Our REGTT *Approach*

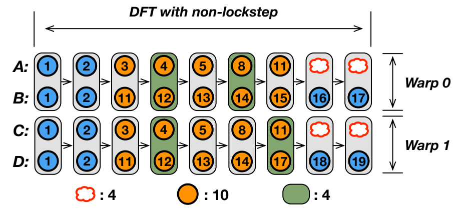
We first present our REGTT approach (Section III-C1). We then examine its three major phases in more detail (Sections III-C2 – III-C4). Finally, we illustrate our approach in Figure 3 by revisiting our motivating example (Section III-C5).

1) Overview: REGTT is designed to exploit **Observations 1 – 3** to accelerate tree traversals on GPUs. As il-



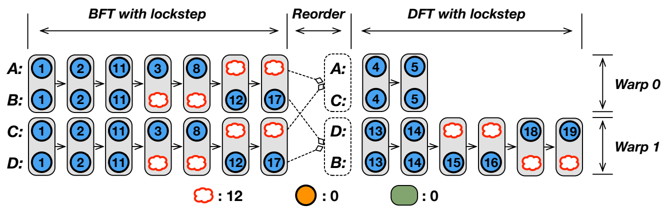


(a) *lockstep*

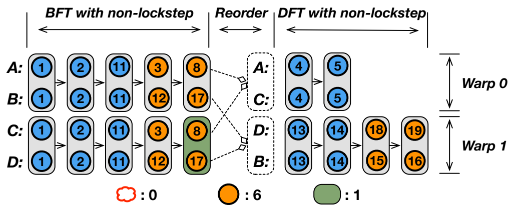


(b) *non-lockstep*

Fig. 2: Tree traversals with AUTOROPES.



(a) *lockstep*



(b) *non-lockstep*

Fig. 3: Tree traversals with REGTT ($d = 3$).

lustrated in Algorithm 2, REGTT answers a set of queries Q on a tree T, by employing a procedure Tree_Traversal running at the host CPU and three GPU kernels, Tree_BFT(), stable_sort_by_key() (from Thrust), and Tree_DFT().

Initially, BFT is performed on the tree nodes at depths less than an *reorder depth* d (starting from 0), based on **Observations 1** and **2**. Then all the queries are reordered at depth d according to their truncation histories, based on **Observation 3**. Finally, DFT is applied to the tree nodes at depth d or deeper. How to select d is discussed in Section III-D2. W, V and K are thread-local. W keeps track of the work (i.e., a set of nodes) to be visited by each thread. V records the truncation information for each query. Finally, K contains an integral representation of the truncation history for each query.

Following AUTOROPES [11], REGTT performs a tree traversal in either *lockstep* or *non-lockstep* mode. This is indicated by $Mode$, which is initialized to 1 if *lockstep* is used and 0 if *non-lockstep* is used. For each thread t that handles a particular query, V_t maintains its truncation information. As an invariant in both modes, $V_t[n] = False$ (i.e., 0) if and only if n's child nodes are not visited (at Line 4 of Algorithm 1). This means that the subtree rooted at n has been truncated.

These two execution modes are handled differently at Lines 21 and 33. In *lockstep*, we make use of a CUDA function $\_\_any(V[n])$ that returns $true$ if there exists at least one thread t in the warp such that $V_t[n] = true$ for node n. This forces all the threads in the same warp to visit the child nodes of n in lockstep. To ensure correctness, thread t is effectively idle at n if it is truncated for its corresponding query, which is indicated when $V_t[n] = false$ at Lines 19 and 31, causing $visit(n)$ to be skipped correctly. In *non-lockstep*, every thread t decides whether to visit the child nodes of a node n by considering its own $V_t[n]$ only (as in Algorithm 1).

2) Breadth-First Traversal: If no tree truncation happens at a parent node, all its child nodes will be visited according to **Observation 1**. To ensure that all such sibling nodes can be visited before their child nodes, BFT rather than DFT (as adopted originally in Algorithm 1) is performed.

BFT exploits better tree traversal regularities at sibling nodes than DFT. However, applying BFT exclusively to an entire search tree is ineffective for two reasons. First, more branch divergence and higher memory access overhead are expected to happen at the nodes deeper in the tree according to **Observation 2**, resulting in poor GPU resource utilization. Second, the memory usage incurred by BFT grows exponentially as we increase the depth of the tree to be explored.

As a compromise, we apply BFT only to the nodes lower than a given reorder depth d, by calling the GPU kernel Tree_BFT(). For these nodes, tree truncations will happen, but moderately rather than excessively (**Observation 2**). The truncation histories for all queries are recorded in V. For the remaining nodes at depth d or deeper, we switch to DFT (but on reordered queries based on their truncation histories).

3) Query Reordering: Query-dependent tree truncations can cause warp divergence and memory-access irregularity, resulting in poor performance. However, queries with similar truncation histories tend to exhibit similar future traversal behaviors according to **Observation 3**. Thus, we are motivated to rearrange the queries being processed based on their truncation histories. The threads with similar truncation histories will be regrouped together to execute in the same warp. As a result, these threads will be more likely to visit the same tree node simultaneously, resulting in improved memory coalescing and load balance as well as reduced warp divergence.

Query reordering introduces pure overhead. All the threads need to synchronize with each other before the reordering phase takes place. The performance penalty caused by frequent reorderings can outweigh the performance benefit achieved. In our REGTT approach, we apply query reordering only once.

In theory, query reordering can be done at any time during the tree traversal. However, we choose to do it between BFT

Algorithm 2: REGTT (*lockstep* and *non-lockstep*)

```
Input: Tree $T$, Set $Q$, $Mode = 1$ (lockstep) or 0 (non-lockstep)
Procedure Tree_Traversal()
1     Compute a reorder depth $d$ by profiling and regression analysis;
2     Let $ThreadCnt$ be the total number of GPU threads available;
3     for $i = 0$ to $Q.size() - 1$ step $ThreadCnt$ do
4         Let $Q_i$ be the subset of $Q$ processed at the $i$-th iteration;
5         $W[0 : ThreadCnt - 1]$ is an array of work lists ($\emptyset$ initially);
6         $K[0 : ThreadCnt - 1]$ is an array of keys (uninitialized);
7         $V \leftarrow \{V_t \mid 0 \leq t < ThreadCount\}$, where $V_t$ is a bit vector
          for thread $t$ (initially 0, i.e., False for all its bits);
8         Tree_BFT ($Q_i, W, K, V, d$);
9         stable_sort_by_key ($K, W$);
10        Tree_DFT ($Q_i, W, V$);

GPU_Kernel Tree_BFT (Set $Q$, Array $W$, Array $K$, Set $V$, Depth $d$)
11    foreach GPU thread $t$ do
12        Let $q \in Q$ be the query handled by $t$;
13        Add $T$'s root node $n_r$ at the end of $W[t]$;
14        $V_t[n_r] \leftarrow$ True;
15        while front node of $W[t]$ is at a depth less than $d$ do
16            $n \leftarrow$ FRONT node of $W[t]$ removed and assigned;
17            if $n$ has a parent node $n_p$ then
18                $V_t[n] \leftarrow V_t[n_p]$;
19            if $V_t[n]$ then
20                $visit(n)$; // Line 4 in Algorithm 1
21            $V_t[n] \leftarrow V_t[n]$ && !Truncation(n, q);
22            if $Mode$ ? __any($V[n]$) : $V_t[n]$ then
23                Add child nodes of $n$ at the end of $W[t]$;
24        Compute $K[t]$ from the truncation history in $V_t$;

GPU_Kernel Tree_DFT (Set $Q$, Array $W$, Set $V$)
25    foreach GPU thread $t$ do
26        Let $q \in Q$ be the query handled by $t$;
27        while $W[t] \neq \emptyset$ do
28            $n \leftarrow$ BACK node of $W[t]$ removed and assigned;
29            if $n$ has a parent node $n_p$ then
30                $V_t[n] \leftarrow V_t[n_p]$;
31            if $V_t[n]$ then
32                $visit(n)$; // Line 4 in Algorithm 1
33            $V_t[n] \leftarrow V_t[n]$ && !Truncation(n, q);
34            if $Mode$ ? __any($V[n]$) : $V_t[n]$ then
35                Add child nodes of $n$ at the end of $W[t]$;
```

and DFT by calling Thrust::stable_sort_by_key (K, W), a C++ template library for CUDA from Thrust, at Line 9. Here, W is an array of worklists to be sorted according to its parallel array K of sorting keys. At Line 24, $K[t]$ is obtained as an integral representation of the truncation history for the query handled by thread t. Let $n_0, n_1, \dots, n_{s-1}$ be the BFT sequence produced for all the s nodes at depths less than d in the search tree T considered. $K[t]$ is simply the integer derived from its binary representation $V_t[n_0]V_t[n_1] \cdots V_t[n_{s-1}]$.

4) Depth-First Traversal: After having reordered the queries, the GPU kernel Tree_DFT() is called to perform DFT afterwards. The threads that exhibit similar future traversal behaviors are now regrouped into the same warp, resulting in improved memory coalescing and GPU resource utilization.

5) Example: Let us explain REGTT in Figure 3 by using our motivating example in Figure 1. We will continue to

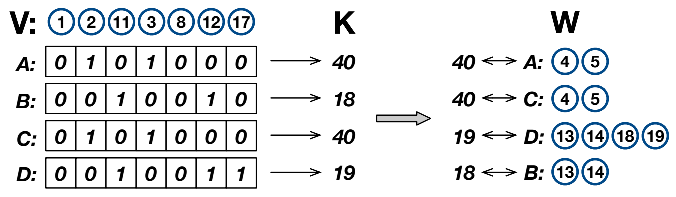


Fig. 4: Query reordering for Figure 3. The nodes shown for each query represent the work left by BFT in its worklist and handled over to DFT.

assume that the warp size is 2. In addition, we assume that $d = 3$. Figure 3(a) illustrates one case when both BFT and DFT are performed in *lockstep*. Figure 3(b) illustrates another case when both BFT and DFT are performed in *non-lockstep*.

Figure 4 shows how query reordering is performed. At the BFT phase, the truncation histories for all the four queries A – D are recorded in V and transformed into the four sorting keys stored in K. Note that W contains the work handled over from BFT to DFT. Specifically, $W[t]$ contains the root nodes of a forest of subtrees to be visited in the DFT manner for its corresponding query. For BFT, the four queries are processed in the order of A, B, C and D (arbitrarily), so that A and B are executed in one warp and C and D are executed in another warp. For DFT, its performance-enhancing order is found to be A, C, D and B. As a result, A and C will be regrouped into one warp and D and B will be regrouped into another.

Compared with AUTOROPES as illustrated in Figure 2, REGTT can make significant performance improvements, by reducing branch divergence, improving memory coalescing, and enhancing load balance (i.e., GPU resource utilization).

D. Implementing Our REGTT *Approach*

BFT and DFT are well-known techniques. In this section, we describe our implementation of REGTT, by focusing on how to reduce memory consumed by DFT (Section III-D1), how to select reorder depths (Section III-D2), how to deal with query-dependent traversal orders for sibling nodes (Section III-D3), and how to reduce global memory access overhead with simple data layout transformations (Section III-D4).

1) Reducing Memory Usage by Bit Vectors: In REGTT (Algorithm 2), V_t keeps track of the truncation history for its corresponding query by both BFT and DFT. Due to the nature of DFT, the space consumed by V_t at the DFT phase can be reduced significantly. V_t can be limited to a bit vector of size equal to the maximum depth of the search tree, by re-cycling its bits in the DFT manner. How to limit the space consumed by V_t at the BFT phase is discussed below.

2) Selecting an Reorder Depth: We use a simple technique to predict an reorder depth d to be used for a tree traversal algorithm by combining profiling and regression analysis. In Section IV, we show that this simple model is reasonably precise for a set of five representative benchmarks evaluated.

The total execution time T elapsed on executing a tree traversal algorithm under REGTT can be estimated as:

$$T = \sum_{i=0}^{I-1} T_i \qquad I = \lceil \frac{Q.size()}{ThreadCnt} \rceil \tag{1}$$

where T_i is the time consumed by each iteration of the `for` loop at Line 3 of Algorithm 2. I gives the total number of iterations executed when all the queries in Q are block-cyclically distributed to all the available threads totaling $ThreadCnt$.

The per-iteration execution time T_i can be broken into:

$$T_i = T_i^B + T_i^R + T_i^D \tag{2}$$

which is the sum of the times spent at the three major phases of REGTT: BFT, query reordering and DFT.

Let a search tree be k-ary with N nodes in total. In the worst case, k^j represents the maximum number of nodes at depth j (starting from 0). According to **Observation 2**, the number of nodes visited at BFT is estimated roughly by:

$$N^B = \sum_{j=0}^{d-1} k^j = \frac{k^d - 1}{k - 1} \tag{3}$$

Thus, the number of nodes visited by DFT is estimated as:

$$N^D = N - N^B = N - \frac{k^d - 1}{k - 1} \tag{4}$$

For our parallelization approach, we consider three performance metrics under a traversal strategy $s \in \mathcal{S}$, where $\mathcal{S} = \{B, D\}$, with B standing for BFT and D for DFT:

- $\mathcal{M}_{\text{TI}}^s$: percentage of the number of times that an idle thread visits a truncated node or no node at all (marked by ○ in Figures 2 and 3) over the number of times that any thread visits any node or no node at all
- $\mathcal{M}_{\text{MEM}}^s$: percentage of the number of global load transactions over the number of global load requests
- $\mathcal{M}_{\text{BD}}^s$: percentage of divergent branches over the total number of branches executed (at Line 5 of Algorithm 1)

Let $\mathcal{P} = \{\text{TI}, \text{MEM}, \text{BD}\}$. We model each of these three performance metrics as a quadratic function of d:

$$\mathcal{M}_p^s = a_p^s \times d^2 + b_p^s \times d + c_p^s \tag{5}$$

where $p \in \mathcal{P}$. Here, a_p^s, b_p^s and c_p^s are coefficients to be found.

Recall that REGTT runs in two modes: *lockstep* and *non-lockstep*. As illustrated in Figures 2 and 3, $\mathcal{M}_{\text{TI}}^s$ is relevant to *lockstep* while $\mathcal{M}_{\text{MEM}}^s$ and $\mathcal{M}_{\text{BD}}^s$ are relevant to *non-lockstep*. Thus, T_i^B and T_i^D can be estimated as follows:

$$T_i^B = \begin{cases} \rho^B \times \theta^B \times N^B \times \mathcal{M}_{\text{TI}}^B & (\textit{lockstep}) \\ \rho^B \times \theta^B \times N^B \times \mathcal{M}_{MEM}^B \times \mathcal{M}_{BD}^B & (\textit{non-lockstep}) \end{cases} \tag{6}$$

$$T_i^D = \begin{cases} \rho^D \times \theta^D \times N^D \times \mathcal{M}_{\text{TI}}^D & (\textit{lockstep}) \\ \rho^D \times \theta^D \times N^D \times \mathcal{M}_{MEM}^D \times \mathcal{M}_{BD}^D & (\textit{non-lockstep}) \end{cases} \tag{7}$$

where ρ^s denotes the average processing time at a node for each query and θ^s denotes the percentage of nodes visited (over N^s) under a particular traversal strategy $s \in \mathcal{S}$.

In (2), the query reordering time T_i^R can be estimated as:

$$T_i^R = \phi \times N^B \tag{8}$$

where ϕ represents the average sorting time per node, subject to the following constraint on global memory usage:

$$G \geqslant \frac{1}{8} \times N^B \times ThreadCnt \tag{9}$$

where G is the global memory size in bytes.

Combining (2) – (8) and then substituting into (1), we obtain a function $T = f(d)$ that relates the execution time T to d. To discover all the unknown coefficients in T for a benchmark under a particular traversal strategy, we execute it under REGTT on GPUs for m times under different input queries Q with m different values of d subject to (9), $d_0, d_1, \cdots d_{m-1}$, and record the corresponding executions times as $T_0, T_1, \cdots T_{m-1}$. Given the data set $\{(d_i, T_i) \mid 0 \leq i < m\}$, we find all the unknown coefficients in T for the benchmark by performing a non-linear regression analysis. Finally, the optimal reorder depth d for the benchmark can be found by solving the non-linear function $T = f(d)$ subject to (9) by using a math tool, e.g., Mathematica assisted with its advanced visualization package.

3) Regulating Traversal Orders on Siblings: In Algorithm 1, the order adopted for traversing the child nodes of a parent node is left unspecified at Line 6. When there is no preferable traversal order, such sibling nodes can always be traversed from left to right. However, in some algorithms such as Nearest Neighbor classification [5], the traversal order for the child nodes at a particular parent node is decided at run time. This is an optimization technique as visiting some nodes earlier enables refining the truncation condition used sooner, thereby allowing more nodes to be truncated in future traversals. For such algorithms, *non-lockstep* works in the usual way as different threads in the same warp can work on different nodes at the same time. For *lockstep*, all the threads in the same warp will select the traversal order for the child nodes at a parent node by majority voting, by examining only the first child node traversed by each thread, as in AUTOROPES [11].

4) Optimizing Memory Access: We improve memory access efficiency by enhancing memory coalescing as follows:

- When constructing a search tree, we store its nodes consecutively in the global memory according to the order in which they are likely to be accessed. The nodes at depths less than d are stored in the BFT order. The remaining ones are stored in the DFT order.
- In tree traversals, the same field of different struct objects is often requested simultaneously by multiple threads, which may cause great memory access overhead if the requests cannot be coalesced. In order to avoid this, we apply an AoS (Array of Structures) to SoA (Structure of Arrays) transformation when the tree data are copied from the host main memory to the device global memory.

IV. EVALUATION

We evaluate REGTT against the state-of-the-art, AUTOROPES [11], for parallelizing tree traversal algorithms on five representative benchmarks. REGTT outperforms AUTOROPES by 1.66x on average, by enhancing load balance, improving memory coalescing and reducing branch divergence.

A. Methodology

1) Experimental Setting: All experiments are conducted on a Nvidia Tesla K20c GPU, which has 13 SMs with each SM containing 192 SPs running at 706MHz. The GPU has 4GB of

TABLE I: Benchmark characteristics.

Bench-mark	Fields	Description	#Objects	#Queries	Tree Construction Algorithm	Object Representation	Tree Type	Traversal Order for Sibling Nodes at Line 6 of Algo. 1
BH	astronomy	n-body simulation	100,000	10,000,000	bh-tree	leaf nodes	oct-tree	fixed (left-to-right)
NN	data mining pattern recognition	finding nearest neighbors	100,000	100,000	kd-tree	leaf nodes	binary-tree	dynamic (app-specific)
kNN (k=8)	data mining pattern recognition	finding top k nearest neighbors	100,000	100,000	kd-tree	all tree nodes	binary-tree	dynamic (app-specific)
VP	graphics	finding nearest neighbors	100,000	100,000	vp-tree	all tree nodes	binary-tree	dynamic (app-specific)
PC	physics	finding number of neighbors falling within a radius	100,000	100,000	kd-tree	leaf nodes	binary-tree	fixed (left-to-right)

global memory and 48KB of shared memory per block. The version of CUDA is 7.5 with compute capability 3.5.

2) Benchmarks: Table I lists a set of five benchmarks from several real-world application areas as shown. To build the search tree for a benchmark, we give its dataset size, the algorithm used for its construction, the type of the tree constructed (bh-tree [3], kd-tree [15] and vp-tree [5]), and the number of queries issued. In the case of *BH*, the same 100,000 objects are issued in 100 time steps. In addition, we also indicate whether the traversal order for sibling nodes at Line 6 of Algorithm 1 is fixed (or static), e.g., from left to right or dynamic (to be determined at run time). Finally, *NN* and *kNN* are considered to be different (as shown in Column "Object Representation"). In *NN*, the information in the objects is stored in "leaf nodes" only. In *kNN*, such information is also stored in interior nodes. This affects only how *visit*() is executed at Line 4 of Algorithm 1.

3) Automatic Parallelization: Given a sequential tree traversal algorithm (in C), REGTT applies Algorithm 2 to automatically transform it into a CUDA program, which is then compiled by the **NVCC** compiler under **-O2**. To provide insights on our experimental results, the Nvidia profiler **nvprof** is used to measure some metrics introduced in Section III-D2.

B. Performance Improvements

Table II compares REGTT and AUTOROPES under their best threading configurations for the five benchmarks, with each configuration identified by the total number of threads created, i.e., $ThreadCnt$ in Algorithm 2. As in [11], these best configurations are found by experimentation. For REGTT, the predicted reorder depths for these configurations are used and will be discussed in Section IV-C1. The inputs used are taken from [11] and explained briefly in Section IV-B2.

REGTT outperforms AUTOROPES by 1.66x on average. The best speedup 8.24x is observed at *PC* with input `Geocity` in *non-lockstep*. REGTT is inferior slightly at *VP* with `Random` in *lockstep* (0.91x) and `Covtype` in *non-lockstep* (0.99x).

1) Benchmarks: The average speedups of REGTT over AUTOROPES are 1.18x for *BH*, 1.82x for *NN*, 1.38x for *kNN*, 1.10x for *VP*, and 2.58x for *PC* under all the configurations considered (with different inputs under the two execution modes). *PC* is the best performer due to its fixed left-to-right order for traversing all sibling nodes (Table I). While also adopting the same fixed order, *BH* does not benefit as much due to its relatively larger query reordering overhead incurred on an oct-tree used. The remaining three benchmarks, *VP*, *NN* and *kNN*, are all concerned with finding nearest neighbors, by dynamically deciding the visitation orders for sibling nodes (Table I). Among the three, *VP* is the worst performer while *NN* is the best. As shown in Table I, *NN* and *kNN*'s search trees are kd-trees while *VP*'s search tree is a vp-tree. In the sequential setting (Algorithm 1), vp-trees are introduced to solve nearest neighbor queries but may end up with too many branches for high-dimensional datasets [16]. For each node, *VP* visits its child node that most likely contains the nearest neighbor. This way, *VP* can update its truncation condition more effectively, thereby potentially skipping more nodes than *NN* and *kNN*. Thus, REGTT is more effective for *NN* and *kNN* than *VP*. Despite this, REGTT still outperforms AUTOROPES slightly for *VP*, due to improved memory coalescing and GPU resource utilization made at the DFT phase under most of the configurations evaluated. Note that different parallelization strategies may visit different numbers of nodes, affecting possibly their performance but not correctness.

2) Inputs: All the inputs are taken from [11]. In particular, `Random` is a dataset for objects with random information, `Plummer` is a dataset for bodies of equal mass, `Covtype` represents a forest cover dataset, `Mnist` is a dataset of handwritten digits, and `Geocity` represents a 2-dimensional point city location dataset. `Plummer`, `Covtype`, `Mnist` and `Geocity` are derived from real-world applications, in which objects tend to cluster together in space. Thus, REGTT always performs better under one of these real inputs than `Random`, as spatially close queries can be handled efficiently, initially under BFT and later under DFT on reordered queries.

3) lockstep vs. non-lockstep: REGTT outperforms AUTOROPES by 1.34x in *lockstep* and 1.98x in *non-lockstep*. In *lockstep*, REGTT improves performance by reducing the number of threads idling on visiting a truncated node or no node at all. In *non-lockstep*, REGTT improves performance by enhancing memory coalescing and load balance as well as reducing branch divergence. Thus, REGTT is more effective for *non-lockstep*, as analyzed further in Section IV-C3.

C. Performance Analysis

We analyze our results by examining the effects of reorder depths on performance (Section IV-C1), the effects of query reordering on performance (Section IV-C2), and the correlations of the performance improvements with the three performance metrics introduced in Section III-D2 (Section IV-C3).

TABLE II: Performance results of comparing REGTT and AUTOROPES.

Bench-mark	Input	Lockstep					Speed-up (x)	Non-lockstep					Speed-up (x)
		Autoropes		RegTT				Autoropes		RegTT			
		ThreadCnt	Time (ms)	*ThreadCnt*	*d*	Time (ms)		*ThreadCnt*	Time (ms)	*ThreadCnt*	*d*	Time (ms)	
BH	Plummer	174720	29775	199680	4	25271	1.18	199680	58087	291200	3	44237	1.31
	Random	149760	31141	199680	3	28325	1.10	199680	39249	66560	4	34711	1.13
NN	Covtype	66560	5091	174720	10	3255	1.56	58240	5006	33280	13	3572	1.40
	Mnist	66560	5088	174720	10	3231	1.57	66560	5415	33280	13	4298	1.26
	Random	133120	1524	174720	9	930	1.64	58240	660	33280	11	552	1.20
	Geocity	66560	1552	133120	10	898	1.73	66560	2492	16640	11	593	4.20
kNN	Covtype	66560	5291	66560	5	3812	1.39	58240	725	266240	8	601	1.21
	Mnist	58240	5321	58240	5	3988	1.33	58240	1276	266240	8	1053	1.21
	Random	66560	5367	58240	6	3810	1.41	58240	977	16640	10	739	1.32
	Geocity	66560	3171	124800	4	1759	1.80	66560	14	33280	8	10	1.40
VP	Covtype	66560	2483	349440	2	2241	1.11	133120	408	199680	2	412	0.99
	Mnist	58240	3590	149760	2	3560	1.01	66560	1361	199680	2	1359	1.00
	Random	66560	3520	266240	2	3852	0.91	66560	1329	199680	2	1326	1.00
	Geocity	66560	190	399360	2	134	1.42	133120	25	49920	6	19	1.32
PC	Covtype	16640	2918	116480	6	2309	1.26	83200	6505	16640	14	1892	3.44
	Mnist	133120	1162	133120	6	946	1.23	66560	1502	66560	12	805	1.87
	Random	66560	1938	99840	6	1597	1.21	66560	2785	16640	13	1271	2.19
	Geocity	66560	1120	99840	8	909	1.23	116480	1573	133120	10	191	8.24
Average		80888.89	6124.56	165475.56	5.56	5045.94	1.34	90133.33	7188.28	107697.78	8.33	5424.50	1.98

1) Reorder Depths: Different benchmarks tend to settle with different reorder depths at their best configurations, as shown in Table II. When interpreting the reorder depths for *BH*, we need to recall the fact that it uses an oct-tree (Table I). As for *VP*, its reorder depths are small (i.e., 2 except for one input) due to the nature of its vp-tree used, as discussed earlier in Section IV-B1. We will analyze the speedups achieved by REGTT with such small reorder depths in Section IV-C3.

In general, *lockstep* prefers reorder depths that are no larger than *non-lockstep* (except for *BH* with `Plummer`). In *lockstep*, REGTT aims to improve only GPU resource utilization, which is translated indirectly into performance. In *non-lockstep*, REGTT reduces branch divergence and improves memory coalescing, which are both related directly to performance. As a result, with relatively larger reorder depths, more precise truncation histories can be discovered, giving rise to greater overall benefits to *non-lockstep* than *lockstep*.

Figure 5 shows the performance gaps of REGTT with the predicted reorder depths used (in Table II) and their corresponding optimal ones (found by exhaustive experimentation). Our simple prediction model (Section III-D2) appears to be reasonably accurate. The average gap is 1.06%, with the largest being 8.07%, observed at *PC* with *Mnist* in *non-lockstep*. In this case, the predicted reorder depth is 12 while the optimal one is 11. With the suboptimal choice, REGTT still delivers a speedup of 1.87x over AUTOROPES (Table II).

2) Query Reordering: To understand the effects of query reordering on performance, Table III gives the speedups over AUTOROPES with query reordering turned off (using the same baseline as in Table II). These results demonstrate clearly its significantly positive impact on performance. Without reordering, REGTT is not attractive, with 1.02x for *lockstep* and 1.01x for *non-lockstep*, on average. However, this does not undermine the significant role that BFT plays when query reordering is turned on. With BFT, we are able to keep track of queries' truncation histories effectively at the top of a tree without introducing much bookkeeping overhead, enabling REGTT to achieve significant speedups over AUTOROPES.

TABLE III: Speedups of REGTT over AUTOROPES without query reordering.

Benchmark	Input	Speedups (x)	
		Lockstep	Non-Lockstep
BH	Plummer	0.99	0.98
	Random	1.00	0.96
NN	Covtype	0.99	0.94
	Mnist	0.98	0.92
	Random	0.99	0.95
	Geocity	0.99	0.99
kNN	Covtype	1.02	1.07
	Mnist	1.01	1.08
	Random	1.01	1.02
	Geocity	1.04	1.27
VP	Covtype	0.99	0.96
	Mnist	0.99	0.96
	Random	0.99	0.96
	Geocity	0.98	0.96
PC	Covtype	1.15	0.99
	Mnist	1.09	1.01
	Random	1.03	0.94
	Geocity	1.07	1.23
Average		1.02	1.01

For *NN* or *PC* with large reorder depths, REGTT is still slightly superior over AUTOROPES without query reordering due to improved memory coalescing and GPU resource utilization at the BFT phase. In the case of *VP* with relatively smaller

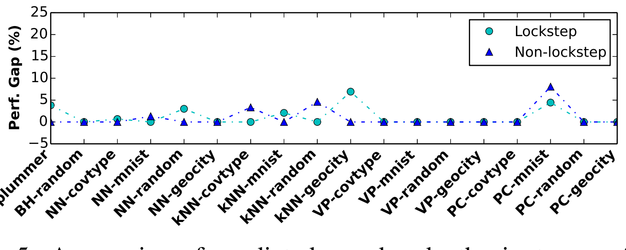


Fig. 5: Accuracies of predicted reorder depths in terms of the execution times achieved (with the optimal REGTT as the baseline).

reorder depths used, performing BFT before DFT without applying query reordering is not beneficial.

3) Performance Metrics: Let us provide some insights on the results given in Table II, by examining the correlations between the speedups achieved by REGTT over AUTOROPES with the three performance metrics, $\mathcal{M}_{\rm TI}$, $\mathcal{M}_{\rm MEM}$ and $\mathcal{M}_{\rm BD}$, introduced in Section III-D2. Recall that $\mathcal{M}_{\rm TI}$ represents the percentage of the threads idling on visiting a truncated node or no node at all, $\mathcal{M}_{\rm MEM}$ captures the percentage of transactions over all memory requests made, and $\mathcal{M}_{\rm BD}$ measures the percentage of divergent branches (at Line 5 of Algorithm 1)).

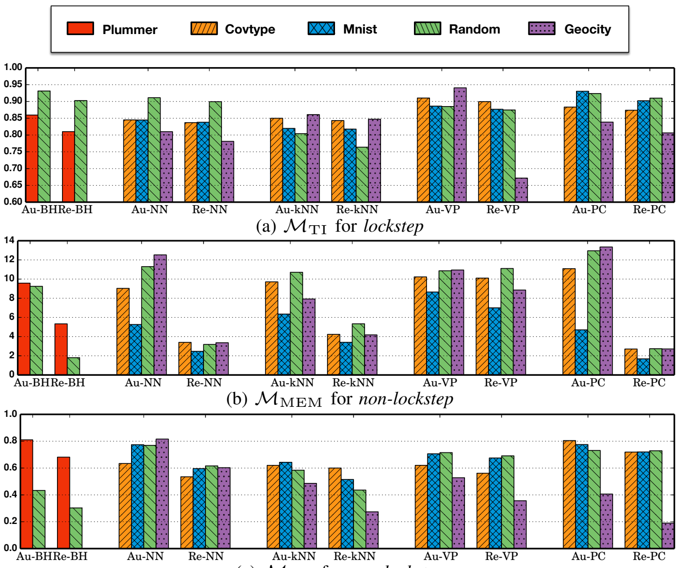
As discussed in Section III-D2, $\mathcal{M}_{\rm TI}$ is relevant to *lockstep* while $\mathcal{M}_{\rm MEM}$ and $\mathcal{M}_{\rm BD}$ are relevant to *non-lockstep*. In *non-lockstep*, there are some threads idling at the end of a warp execution, as shown in Figure 3. However, $\mathcal{M}_{\rm TI}$ is insignificant relative to $\mathcal{M}_{\rm MEM}$ and $\mathcal{M}_{\rm BD}$ and thus ignored in (7) and our analysis below. For *lockstep*, $\mathcal{M}_{\rm MEM}$ and $\mathcal{M}_{\rm BD}$ can be regarded as 1 and 0, respectively. When parallelizing an algorithm with *lockstep*, we strive to reduce its $\mathcal{M}_{\rm TI}$. When parallelizing an algorithm with *non-lockstep*, we strive to reduce its $\mathcal{M}_{\rm MEM}$ and $\mathcal{M}_{\rm BD}$ simultaneously.

Figure 6 depicts the measurements of the three metrics for AUTOROPES and REGTT for all threading configurations in Table II. For each metric, the lower a bar is, the better. For each benchmark, we obtain $\mathcal{M}_{\rm TI}$ by instrumentation, and $\mathcal{M}_{\rm MEM}$ and $\mathcal{M}_{\rm BD}$ by tracking *nvprof*'s *gld_transactions_per_request* and *branch_efficiency* events, respectively.

a) lockstep: Idle Threads: In this execution mode, AUTOROPES avoids branch divergence and the need to perform memory coalescing across the tree nodes at the expense of generating idle threads. REGTT improves its thread under-utilization by rearranging queries so that more queries with similar traversals are performed by the threads in the same warp. As shown in Figure 6(a), REGTT has achieved a smaller $\mathcal{M}_{\rm TI}$ than AUTOROPES for all the five benchmarks under all the inputs tested, with an average reduction of 3.60%.

By reducing $\mathcal{M}_{\rm TI}$, REGTT enables more threads to run on more GPU cores, as $ThreadCnt$ is always no smaller under REGTT than AUTOROPES in their best threading configurations (Table II). Consider *kNN* with `Mnist`. By using the same configuration as REGTT, AUTOROPES cannot fully utilize the resources available, causing it to drop from its peak performance by 15.05%. Note that a small average reduction of 3.60% in $\mathcal{M}_{\rm TI}$ is significant, since this enables REGTT to more than double the number of threads utilized by AUTOROPES, thereby improving the hardware resource utilization and achieving an average speedup of 1.34x (Table II).

VP represents an interesting case. Due to the nature of its *VP*-tree used, as discussed in Section IV-B1, its optimal reorder depths are expected to be small, which are indeed the smallest in Table II. In *lockstep*, $d = 2$ for all the four inputs tested. Despite this, REGTT outperforms AUTOROPES under the three real-world inputs, `Covtype`, `Mnist` and `Geocity`. As the objects in these inputs tend to be highly clustered, a fairly short truncation history can produce a performance-enhancing better for DFT. For example, REGTT reduces $\mathcal{M}_{\rm TI}$ by 28.59% under the input *Geocity* compared to AUTOROPES, resulting in a speedup of 1.42x. For `Random`, we see only a small reduction of 1.16% in $\mathcal{M}_{\rm TI}$, due to a random distribution of its objects. However, REGTT is slower than AUTOROPES by 0.91x. In this case, the benefit obtained by REGTT from a small reduction in $\mathcal{M}_{\rm TI}$ is out-weighed by the extra cost incurred due to REGTT's visiting (redundantly) on some nodes that happen to have been truncated by AUTOROPES.



(a) $\mathcal{M}_{\rm TI}$ for *lockstep*
(b) $\mathcal{M}_{\rm MEM}$ for *non-lockstep*
(c) $\mathcal{M}_{\rm BD}$ for *non-lockstep*

Fig. 6: Comparing AUTOROPES and REGTT in terms of $\mathcal{M}_{\rm TI}$, $\mathcal{M}_{\rm MEM}$ and $\mathcal{M}_{\rm BD}$ for all the configurations in Table II (*X*–*B*: benchmark B parallelized by $X \in \{\rm Au, Re\}$, where Au stands for AUTOROPES and Re for REGTT).

b) non-lockstep: Memory Coalescing and Warp Divergence: In this execution mode, AUTOROPES aims to improve GPU resource utilization by allowing different threads in the same warp to visit different nodes at the expense of memory-access irregularity and branch divergence. REGTT attempts to improve memory coalescing and reduce branch divergence by reorganizing the queries processed so that similarly-behaved queries can be executed by the threads in the same warp.

As shown in Figures 6(b) and (c), REGTT has improved AUTOROPES in reducing $\mathcal{M}_{\rm MEM}$ and $\mathcal{M}_{\rm BD}$ by 51.17% and 19.10%, respectively, on average. These reductions are largely responsible for the speedups achieved by REGTT (Table II). Note that REGTT has also obtained a better overall load balance than AUTOROPES by utilizing 19.49% more threads due to improved intra-warp load balance (from query reordering).

In particular, REGTT has successfully reduced $\mathcal{M}_{\rm MEM}$ and $\mathcal{M}_{\rm BD}$ for all the five benchmarks under all the inputs except for *VP* under `Random`. In this exceptional case, REGTT has increased $\mathcal{M}_{\rm MEM}$ by 3.36% but deceased $\mathcal{M}_{\rm BD}$ by 2.22%. Overall, no performance variation is observed.

Finally, REGTT is more effective in boosting the performance of AUTOROPES in *non-lockstep* than *lockstep* (Table II). This is because reducing $\mathcal{M}_{\rm MEM}$ and $\mathcal{M}_{\rm BD}$ together in *non-lockstep* is more beneficial than reducing $\mathcal{M}_{\rm TI}$ in *lockstep*.

V. Related Work

A. Domain-Specific Techniques

In many applications where tree traversals are frequently used, domain-specific knowledge has been used to guide optimization. For ray tracing in graphics, two kd-tree traversal algorithms, *kd-restart* and *kd-backtrack*, are demonstrated to alleviate per-ray stack limits on GPUs [17]. With the same objective, a tree traversal implementation for GPUs [18] creates neighbor cell links on the tree pointing to adjacent nodes, and executes in the SIMT fashion for packets of rays. For nearest neighbor classification in graphics and machine learning, a kd-tree construction algorithm on GPUs [13] is presented that selects a splitting plane heuristically and treats large and small nodes differently. For n-body simulations in astrophysics, one CUDA implementation [12] is designed to minimize memory accesses and thread divergence with various global and kernel-specific optimization techniques.

To reduce cache misses caused by irregular tree traversals, sorting the queries into some order was tried before, such as the domain-specific efforts made for ray tracing [14] and n-body simulations [19]. RegTT accelerates tree traversals without considering any application-specific knowledge.

B. General-Purpose Techniques

Many compiler techniques have been proposed to transform tree traversal applications automatically to better utilize the underlying hardware. We examine several below.

1) CPUs: To improve cache locality, *point blocking* [20] applies loop tiling [21] to block frequently accessed nodes, based on the queries sorted according to application semantics. *Traversal splicing* [4] makes point blocking more general by sorting the queries on-the-fly with a node accessing history (rather than a truncation history as introduced in this paper). These two approaches are later combined to exploit SIMD parallelism [22]. To facilitate traversing irregular data structures, an intermediate language and a runtime scheduler are developed for efficient SIMD execution [23]. Code transformations and scheduling strategies are combined to vectorize recursive, task-parallel programs [24].

2) GPUs: G-Streamline [25] is designed to remove memory access and control flow irregularities through data reordering and job swapping. *HP-RDA* [26] is introduced to parallelize a reuse distance analysis by flattening a traditional tree representation of memory access traces. To the best of our knowledge, Autoropes [11], which is significantly improved by RegTT in this paper, is the only general approach for parallelizing tree traversals on GPUs previously.

VI. Conclusion

We introduce a general approach, RegTT, for efficiently parallelizing tree traversals on GPUs. By exploiting tree traversal regularities, which are neglected by the prior work, we demonstrate that RegTT can outperform the state-of-the-art significantly on five representative benchmarks tested. In future work, we will explore new ways to reorder queries for better performance. We will also investigate how to better parallelize vp-tree-based algorithms with gradually-refinable truncation conditions. Finally, we will generalize RegTT for multiple GPUs and heterogeneous CPU-GPU systems.

Acknowledgment

This research is supported by Australian Research Council Grants DP110104628 and DP130101970.

References

[1] Y. Su, D. Ye, J. Xue, and X. Liao, "An efficient GPU implementation of inclusion-based pointer analysis," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 2, pp. 353–366, 2015.

[2] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *PPoPP '12*, pp. 117–128.

[3] J. Barnes and P. Hut, "A hierarchical O(N log N) force-calculation algorithm," *Nature*, vol. 324, pp. 446–449, 1986.

[4] Y. Jo and M. Kulkarni, "Automatically enhancing locality for tree traversals with traversal splicing," in *OOPSLA '12*, pp. 355–374.

[5] P. N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces," in *SODA '93*, pp. 311–321.

[6] C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *MICRO '09*, pp. 45–55.

[7] P. Di and J. Xue, "Model-driven tile size selection for DOACROSS loops on GPUs," in *Euro-Par '11*, pp. 401–412.

[8] S. Lee and R. Eigenmann, "OpenMPC: Extended OpenMP programming and tuning for GPUs," in *SC '10*, pp. 1–11.

[9] P. Di, H. Wu, J. Xue, F. Wang, and C. Yang, "Parallelizing SOR for GPGPUs using alternate loop tiling," *Parallel Comput*, vol. 38, no. 6, pp. 310–328, 2012.

[10] P. Di, D. Ye, Y. Su, Y. Sui, and J. Xue, "Automatic parallelization of tiled loop nests with enhanced fine-grained parallelism on GPUs," in *ICPP '12*, pp. 350–359.

[11] M. Goldfarb, Y. Jo, and M. Kulkarni, "General transformations for GPU execution of tree traversals," in *SC '13*, pp. 10:1–10:12.

[12] M. Burtscher and K. Pingali, "An efficient CUDA implementation of the tree-based barnes hut N-body algorithm," *GPU Computing Gems Emerald Edition*, p. 75, 2011.

[13] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time KD-tree construction on graphics hardware," *ACM Trans Graph*, vol. 27, no. 5, p. 126, 2008.

[14] E. Månsson, J. Munkberg, and T. Akenine-Möller, "Deep coherent ray tracing," in *RT '07*, pp. 79–85.

[15] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[16] T. Bozkaya and M. Ozsoyoglu, "Distance-based indexing for high-dimensional metric spaces," in *SIGMOD '97*, pp. 357–368.

[17] T. Foley and J. Sugerman, "KD-tree acceleration structures for a GPU raytracer," in *HWWS '05*, pp. 15–22.

[18] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek, "Stackless KD-tree traversal for high performance GPU ray tracing," in *Eurographics '07*, pp. 415–424.

[19] 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*, vol. 27, no. 2, pp. 118–141, 1995.

[20] Y. Jo and M. Kulkarni, "Enhancing locality for recursive traversals of recursive structures," in *OOPSLA '11*, pp. 463–482.

[21] J. Xue, *Loop Tiling for Parallelism.* Boston: Kluwer Academic Publishers, 2000.

[22] Y. Jo, M. Goldfarb, and M. Kulkarni, "Automatic vectorization of tree traversals," in *PACT '13*, pp. 363–374.

[23] B. Ren, G. Agrawal, J. R. Larus, T. Mytkowicz, T. Poutanen, and W. Schulte, "SIMD parallelization of applications that traverse irregular data structures," in *CGO '13*, pp. 1–10.

[24] B. Ren, Y. Jo, S. Krishnamoorthy, K. Agrawal, and M. Kulkarni, "Efficient execution of recursive programs on commodity vector hardware," in *PLDI '15*, pp. 509–520.

[25] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen, "On-the-fly elimination of dynamic irregularities for GPU computing," in *ASPLOS '11*, pp. 369–380.

[26] H. Cui, Q. Yi, J. Xue, L. Wang, Y. Yang, and X. Feng, "A highly parallel reuse distance analysis algorithm on GPUs," in *IPDPS '12*, pp. 1080–1092.