Automatic Vectorization of Tree Traversals

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

***Abstract*—Repeated tree traversals are ubiquitous in many domains such as scientific simulation, data mining and graphics. Modern commodity processors support SIMD instructions, and using these instructions to process multiple traversals at once has the potential to provide substantial performance improvements. Unfortunately these algorithms often feature highly diverging traversals which inhibit efficient SIMD utilization, to the point that other, less profitable sources of vectorization must be exploited instead.**

Previous work has proposed *traversal splicing*, a locality transformation for tree traversals, which dynamically *reorders* traversals based on previous behavior, based on the insight that traversals which have behaved similarly so far are likely to behave similarly in the future. In this work, we cast this dynamic reordering as a scheduling for efficient SIMD execution, and show that it can dramatically improve the SIMD utilization of diverging traversals, close to ideal utilization. For five irregular tree traversal algorithms, our techniques are able to deliver speedups of 2.78 on average over baseline implementations. Furthermore our techniques can effectively SIMDize algorithms that prior, manual vectorization attempts could not.

***Index Terms*—automatic vectorization, tree traversals, SIMD, irregular programs**

I. Introduction

An effective way to improve the performance of a program run on modern commodity architectures is to *vectorize* the application by exploiting Single-Instruction Multiple-Data (SIMD) instructions. By using SIMD instructions, multiple scalar operations can be replaced with a single vector operation that acts on multiple pieces of data at once. Because SIMD extensions typically require relatively little extra hardware, exploiting SIMD parallelism is effectively "free" from a power-consumption standpoint, making SIMD an attractive approach to improving performance per watt on modern architectures.

While modern vectorizing compilers (*e.g.*, Intel's icc, IBM's xlc, and GCC) can target SIMD architectures, they primarily rely on identifying simple loop-based control structures with predictable access patterns to generate effective SIMD code [17]. While these patterns arise frequently in *regular* programs, which operate on dense matrices and arrays, they are far less common when dealing with *irregular* programs, which operate over pointer-based data structures. These programs are characterized by complex, data-dependent access patterns, hence complicating vectorization, which requires regularizing computation so that multiple pieces of data are simultaneously operated on by a single instruction. As a result, vectorization of such pointer-based programs has been the province of careful, handwritten, application-specific implementations [4–6, 8, 13, 20, 23, 24]. Generally-applicable approaches to "SIMDization" of irregular programs are scarce.

There are two chief obstacles to general-purpose SIMDization techniques for irregular programs. First, irregular programs are a broad class of applications, with seemingly few commonalities that could be exploited by a general technique. However, within this broad class of programs, there exist subclasses that nevertheless have common structure. In this paper, we focus on *tree traversal algorithms*, wherein a series of "points" each traverses a tree. Such algorithms are prevalent in scientific simulation, where n-body codes have bodies traversing an oct-tree to compute forces [1]; in data-mining, where points traverse kd-trees to locate their nearest neighbor [3]; and in graphics, where ray tracers use bounding volume hierarchies to accelerate ray-object intersection tests [23].

Crucially for our purposes, in these algorithms a single tree is traversed multiple times by multiple points. Hence, if multiple points that visit the same tree node can be processed simultaneously, there will be several common operations that differ only in the (point-specific) data they operate on, a ripe opportunity for SIMD acceleration. In essence, a general approach to exploiting SIMD requires carefully scheduling traversals to ensure that multiple points are at a single tree node at once, and can be processed in a SIMD manner.

This strategy brings up the second obstacle to SIMDization of irregular programs: even if an opportunity for SIMDization can be identified, *exploiting* that opportunity is difficult. Scheduling traversals so that multiple points are visiting a node of the tree simultaneously is non-trivial. A natural scheduling approach is to group four points with similar traversals into a single *packet*. This packet then traverses the tree in lockstep, allowing each of the points in the packet to interact with the tree nodes the packet visits [13, 23]. However, there are two drawbacks to packetized traversals. First, writing a packet-based traversal requires making substantial changes to the code, often in application-specific ways. Second, packet-based approaches break down if packets of similarly-behaving points cannot be identified, or if apparently-similar points later *diverge* as they traverse the tree (*e.g.*, because two rays reflect off a surface in different directions). In such situations, *SIMD utilization* (the fraction of useful operations performed during one SIMD instruction) drops, as most SIMD instructions end up operating on fewer than four points:

> In situations like physical simulation, collision detection or raytracing in scenes, where rays bounce into multiple directions (spherical or bumpmapped

> surfaces), coherent ray packets break down very quickly to single rays or do not exist at all. In the above mentioned tasks, packet oriented SIMD computations is much less useful. [8]

Clearly, it is difficult to exploit this seemingly-simple source of SIMD opportunity, even in hand-written implementations, let alone generally. As a result, most hand-written SIMD implementations focus on highly application-specific sources of vectorization. Similarly, prior attempts to generalize SIMDization of irregular applications have focused on alternative sources of vectorization opportunities [4–6, 8, 20].

In this paper, we will describe *generally-applicable, systematic approaches to scheduling that expose SIMD opportunities and maximize utilization*. In fact, our approaches enable packet-based SIMD computation even in algorithms where careful, hand-written implementations could not exploit it.

Automatic SIMDization of tree traversals

The problem of carefully scheduling traversals to expose opportunities for SIMDization has parallels with improving locality in tree-traversal codes. A key first step in improving temporal locality is to schedule traversals so that multiple traversals are accessing the same parts of the tree at similar times. Previous work presented two scheduling transformations, *point blocking* [11] and *traversal splicing* [12] that automatically performed this scheduling. Section II describes these transformations in more detail.

The key insight in this paper is that these locality transformations can be readily adapted to both transform traversal codes so they can be SIMDized and schedule traversals so that SIMD utilization is maximized. Section III illustrates how point blocking can be used to transform code to enable packetized traversals.

Point blocking alone, however, is not sufficient. As described in the quote above, the divergence of points during traversals leads packet-based approaches to break down, producing very poor SIMD utilization. While our use of point blocking mitigates this effect, it does not fully ameliorate it. Section IV shows how a particular aspect of traversal splicing, its ability to dynamically sort points during traversal, can be leveraged to dramatically improve SIMD utilization. Section V describes an automatic transformation framework called SIMTREE that leverages simple annotations to restructure a tree-traversal application so it can be readily vectorized.

Section VI shows, across several tree-traversal benchmarks, that our transformations can deliver significant performance gains through effective SIMDization, yielding speedups of up to 6.59, and 2.78 on average. Section VII discusses related work, and Section VIII concludes.

Contributions

The primary contributions of this paper are:

- We show how tree-traversal applications can be systematically transformed to both expose SIMD opportunities, through point blocking, and enhance SIMD utilization, through traversal splicing.
- We present a framework for automatically restructuring traversals and data layouts to enable SIMDization.
- We demonstrate that, with our transformations, tree-traversal applications can be effectively vectorized, delivering substantial performance gains and, in some cases, exploiting more SIMD opportunities than previous manual approaches could.

To our knowledge, this work presents the first generally-applicable transformations to packetize and vectorize tree-traversal applications. While the exact amount of benefit that SIMD can provide is highly algorithm dependent, we demonstrate that *our scheduling transformations can organize computations to extract nearly the maximum amount of SIMD utilization from these applications*.

II. BACKGROUND

A. Traversal algorithms

The class of algorithms that we target in this paper are *tree traversal* algorithms, which arise in a number of domains, ranging from astrophysical simulation [1] to data mining [4] to graphics [23]. One example of a tree traversal algorithm is *photon mapping* in ray tracing [10]. To simulate realistic lighting, photon mapping fires many photons from the light sources in the scene. After accounting for Monte-Carlo reflection and refraction, these photons map to various objects in the scene. When eye rays are later cast from the camera, they can return a color value when they encounter a mapped photon.

To efficiently perform photon mapping, ray tracers use an $O(log\ n)$ algorithm. The objects in the scene are inserted into a spatial acceleration tree, constructed as follows: The scene is recursively subdivided, with each node in the tree representing a subspace, and a node's children representing the division of that subspace into two smaller subspaces. Objects reside in the leaf of the tree that corresponds to the portion of the scene they are in. Photon mapping is then performed by traversing the tree to find which object the photon hits. If a photon definitely cannot intersect with any objects in a given subspace, that entire subtree can be skipped during traversal.

Photon mapping, and all tree traversal codes, can be abstracted according to the pseudocode in Figure 1. Each of a set of points traverses the tree recursively. Various truncation conditions (*e.g.*, a photon will not intersect a node's subspace) may cut off a portion of the traversal, while other conditions (*e.g.*, a photon hits an object in a leaf node) may result in some objective value being calculated. If a point is at an interior node and is not truncated, it continues its traversal by recursing on the node's children. Note that while each point performs a different traversal, these traversals are all independent of each other and the overall algorithm is embarrassingly parallel.

These algorithms are highly irregular: the layout of the tree is dynamic and input dependent, the access patterns of a given point are dynamic and involve chasing pointers down the tree, and the overall traversals are complex and unpredictable. Traditional vectorizing/SIMDizing compilers, which look for dense loops over arrays with predictable access patterns, would

```
1  TreeNode root;
2  Set<Point> points;
3  foreach (Point p : points)
4    recurse(p, root);

6  void recurse(Point p, TreeNode n) {
7    if (truncate(p, n)) {
8      updatePoint(p, n);
9      return;
10   }
11   recurse(p, n.child1);
12   recurse(p, n.child2);
13 }
```

Fig. 1. Abstract pseudocode for traversal algorithms

find no opportunities for vectorization. While there is a dense loop in line 3, the recursion and pointer chasing of the recursive method destroy any chance of vectorization.

B. Scheduling transformations

In prior work, we showed that that these algorithms' common structure, despite its seeming irregularity, nevertheless enabled general *scheduling* transformations that rearrange accesses and improve locality. We developed a pair of transformations, *point blocking* [11] and *traversal splicing* [12], that can be applied to irregular tree-traversal algorithms to promote locality of reference.

In our earlier work, we explained these transformations in terms of loop tiling, a traditional locality optimization for regular programs that operate over dense arrays and matrices. Here, we present an alternate interpretation of these transformations that makes their applicability to SIMDization more clear.

In the abstract, we can think of each traversal in a traversal algorithm as being executed by a separate thread. The problem we must tackle is how to *schedule* the execution of these threads on a single processor. The original schedule of execution runs an entire thread (traversal) to completion before switching execution to the next thread. There are no context switches between threads. Even this simple schedule gives us some flexibility: after a thread finishes executing, we can choose any of the remaining threads to execute next. If we choose a thread whose traversal is likely to touch similar portions of the tree as the just-finished traversal, those parts of the tree are likely to still be in cache, improving locality. This is equivalent to *sorting* the points before execution so that points with similar traversals will be executed consecutively [18, 22].

1) Point blocking: When a thread's traversal gets too large to fit in cache, point sorting no longer suffices; even if the next thread's traversal is exactly the same, the nodes of the traversal will already have been evicted from cache. We proposed point blocking to ameliorate this effect [11]. Intuitively, point blocking no longer executes a single thread to completion before running the next thread. Instead, point blocking context switches between threads to improve locality in the tree.

In point blocking, threads are grouped into blocks. A thread visits a single node of the tree (*i.e.*, it executes the body of the recursive method), but immediately after recursing down a child, control is transferred to another thread in the block that is at the same tree node, where the process repeats. If a thread in the block does not visit a particular tree node, its

```
1  TreeNode root;
2  Set<Point> points;
3  foreach (Block block : points)
4    recurse(block, root);

6  void recurse(Block block, TreeNode n) {
7    Block nextBlock;
8    foreach (Point p : block) {
9      if (truncate(p, n)) {
10       updatePoint(p, n);
11       continue;
12     }
13     nextBlock.add(p);
14   }
15   if (!nextBlock.isEmpty()) {
16     recurse(nextBlock, n.child1);
17     recurse(nextBlock, n.child2);
18   }
19 }
```

Fig. 2. Abstract pseudocode for point blocking

turn is "skipped." It will next execute when the threads in the block return to next tree node that its traversal would visit. Thus, all of the threads that must access a particular tree node access that node consecutively, giving good locality in the tree regardless of the size of the threads' traversals.

Constant context switches to achieve this scheduling would have high overhead. Hence, point blocking implements the schedule statically: points are grouped into blocks, and entire blocks traverse the tree, visiting a node if *any* of the points in the block want to visit it. Figure 2 shows how the abstract code of Figure 1 is transformed to implement point blocking.

We note three salient facts. First, the transformation is straightforward: all returns in the main body of the loop are replaced with continues, and points that continue recursion are deferred until all the points have processed the current node. Second, if no points want to continue recursion, the "next" block is empty, so recursion stops; the block traverses the union of the sets of tree nodes that would have been visited by its constituent points. Finally, and most importantly, the main body of the recursive function now contains a simple for loop over a dense block of points (line 8). Each iteration of that loop performs the same set of instructions (disregarding any truncation), and the memory accesses within the loop are predictable: there is no pointer chasing involved. As Section III discusses, point blocking naturally enables SIMD execution.

2) Traversal splicing: Traversal splicing is an alternate approach to scheduling traversal codes to improve locality [12]. Intuitively, while point blocking interleaves traversals at the granularity of individual tree nodes, context switching between threads in a block at every instance of the recursive function, traversal splicing performs its context switching at a coarser granularity. In traversal splicing, various nodes in the tree are marked as *splice nodes*. A thread executes until its traversal reaches one of the splice nodes, or until its traversal truncates at a node that is an ancestor of the splice node. Then control is transferred to another thread, which similarly executes until it reaches the splice node. This process continues until *all* threads in the program have been "paused" at the splice node or truncated. Then control transfers back to the first thread, and execution continues until the next splice node, and so on. Hence, threads are interleaved at the granularity of "partial traversals" that traverse *between* splice nodes.

Though traversal splicing is more complicated than point blocking, it has a few advantages. Context switches in point blocking can only switch between threads in the same block of points. This restriction is good for locality: if control could switch between arbitrary threads, then thread-local (*i.e.*, point-specific) data may fall out of cache. However, it means the schedule is less likely to find points that will visit the same node in the tree. Because traversal splicing context switches between *all* threads in the traversal algorithm, if *any* threads visit the same node, they will be found and their accesses to the node will happen in close succession. Traversal splicing can also be combined with point blocking: while executing a partial traversal, multiple threads can be blocked and interleaved according to the point blocking strategy.

In addition to the locality-enhancing scheduling described above, traversal splicing allows a schedule to be *dynamically* tuned. When context switching between threads, it is useful if the next thread that is executed will have similar behavior as the previous thread (recall the purpose of sorting the points). Because threads are interleaved at the granularity of splice nodes, the scheduler can take the opportunity, at each splice node, to adjust the interleaving order based on the previous history of each thread's execution. Threads that have behaved similarly up until a particular splice node are likely to behave similarly in the future, and can hence be reordered to execute consecutively. This technique is called *dynamic sorting*.

Interestingly, our prior investigations found that dynamic sorting does not improve locality significantly; because partial traversals tend to be small, the precise order of threads within those traversals tends not to matter. In this paper, we capitalize on a key insight: while dynamic sorting may not improve locality, it does improve the similarity of consecutive threads' traversals. As Section IV elaborates, this can dramatically improve a traversal algorithm's SIMD potential.

III. Point blocking to enable SIMD

As described in Section II-B1, applying point blocking exposes a simple for loop over the block of points within the recursive function. Each iteration of that loop performs the same set of instructions, exposing an opportunity for SIMD execution. Point blocking with a block size of S (where S is the SIMD width) is directly analogous to packet-based SIMD approaches [13, 23]. A point block, or packet, traverses the tree, and the algorithm moves on to the next packet after the traversal of the previous packet is complete.

Packet-based approaches break down when the points in the packet behave differently. For example in Nearest Neighbor (NN) each point traverses either the left child or right child of a tree node first based on properties of the point and the node. Hence, traversals *diverge*, and a packet of S points can quickly degenerate into a single point after a few steps, in which case SIMD execution is no longer beneficial. This effect can be ameliorated by having *multiple* packets traverse the tree simultaneously, and compacting packets at each level to repopulate sparse packets and increase SIMD utilization.

```
1  void processVec(Block block, Block
2      nextBlock, int si, TreeNode n, int mask) {
3    __m128 vec_valid;
4    truncateVec(block, si, n, vec_valid);
5    updatePointVec(block, si, n, vec_valid);
6    mask = _mm_movemask_ps(vec_valid) & mask;
7    if (mask & 0x1) nextBlock.add(block, si);
8    if (mask & 0x2) nextBlock.add(block, si + 1);
9    if (mask & 0x4) nextBlock.add(block, si + 2);
10   if (mask & 0x8) nextBlock.add(block, si + 3);
11 }

13 void recurse(Block block, TreeNode n) {
14   Block nextBlock;
15   int si = 0;
16   for (; si < block.size - S + 1; si += S) {
17     processVec(block, nextBlock, si, n, 0xf);
18   }
19   int mask = getValidMask(si, block.size);
20   if (mask > 1) {
21     processVec(block, nextBlock, si, n, mask);
22   } else if (mask == 1) {
23     processPoint(block, nextBlock, si, n);
24   }
25   if (!nextBlock.isEmpty()) {
26     recurse(nextBlock, n.child1);
27     recurse(nextBlock, n.child2);
28   }
29   nextBlock.copyUp(block);
30 }
```

Fig. 3. SIMDized point blocking

While multi-packet traversal and compaction sounds like a complicated optimization, a simple modification of point blocking achieves this goal. Rather than applying point-blocking with a block size of S, we can apply point blocking with a larger block size, and then *strip mine* the resulting loop, adding an inner loop of S iterations. This inner loop can be SIMDized as a single packet, while at each step of the block's traversal, any non-truncated points can be compacted into full packets at the next level. The compaction is implemented by copying points from the current block into a "next block" that visits the children; points in this next block will be contiguous.

Figure 3 shows the result of SIMDizing the strip-mined loop, with $S = 4$. Note that in place of the inner (strip-mined) loop, we use a call to to `processVec` (line 17) that operates on the S points within the packet simultaneously with SIMD instructions (line 1). Within `processVec`, control flow is converted into masking operations represented by `vec_valid`. `vec_valid` is passed to `truncateVec` and `updatePointVec` so that operations are correctly applied according to the original loop's control flow. Finally, any non-truncated points are added to the next block.

Note that strip-mining introduces cleanup code (lines 19–24) that operates over less than a full packet of points. However, as long as there is more than one point, it is still profitable to use SIMD execution. `getValidMask` computes a mask where the bits of valid points are set. This mask is passed to `processVec` to ensure that only valid points are added to the next block. If the cleanup loop only has one point, we call `processPoint` instead.

The process of replacing the strip-mined inner loop with a SIMD implementation is that of vectorizing a dense loop using techniques such as if-conversion. These transformations can be performed by vectorizing compilers [17].

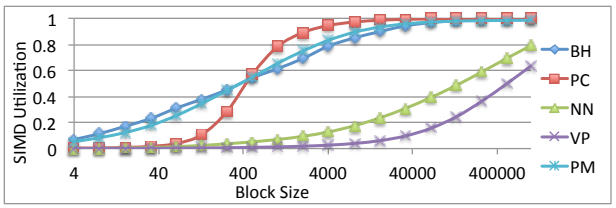


Fig. 4. SIMD utilization versus block size ($S = 4$)

SIMD utilization

We have referred to the concept of SIMD utilization in prior sections; here we formally define it. Let W be the total amount of work (*i.e.*, the total number of nodes accessed by all the points). Let W_s be the the amount of work done in full SIMD packets (*i.e.*, S times the number of calls to `processVec` in line 17). We can define SIMD utilization $U = W_s/W$[1]. If all points follow the exact same traversal, there will be no divergence and all points can be processed as part of a full SIMD packet; hence, $U = 1$. Figure 4 shows the SIMD utilization for five tree traversal algorithms[2]. A larger block size results in better SIMD utilization as there are more points to search through to create full SIMD packets.

Note that a block size equal to the total number of points yields the maximum, *ideal*, SIMD utilization. This schedule of computation extracts the most SIMD potential from the algorithm and input, producing as many full SIMD packets as possible. This ideal utilization measures inherent properties of the algorithm and input. Some algorithms feature so much divergence between traversals that even with extremely large block sizes, it is difficult to find full SIMD packets to process. For example, in NN, the ideal utilization reaches only 0.8. This is because the algorithm itself is highly divergent, and there are many nodes in the tree that fewer than 4 points ever access.

It is clear that in order to maximize SIMD utilization we would like to use a large block size. However a block size that is too large has adverse effects on locality [11] (by way of analogy, think of choosing an overly-large tile size for tiled matrix multiply). Unfortunately, the block sizes at which maximum SIMD utilization is attained feature exactly this poor locality. SIMD cannot help us when the program is memory bound and stalled on load/stores, so poor locality directly counteracts any benefits from vectorization. A schedule that can attain SIMD utilization without compromising locality is necessary.

IV. Traversal Splicing to Enhance Utilization

In Section II-B, we described how locality in tree traversals can be cast as a thread scheduling problem, where we would like to schedule threads which access similar tree nodes consecutively for better locality. The same scheduling problem arises in maximizing SIMD utilization. We would like to schedule multiple threads to access the same tree node consecutively so that they can be executed simultaneously with SIMD instructions.

[1]Note that it is also possible to exploit SIMD in the cleanup loop when it has two or more points; we focus on full SIMD utilization for simplicity.

[2]The **Dragon** input is used for PM, and **Random** inputs are used for the other benchmarks (see Section VI). Each input has one million points.

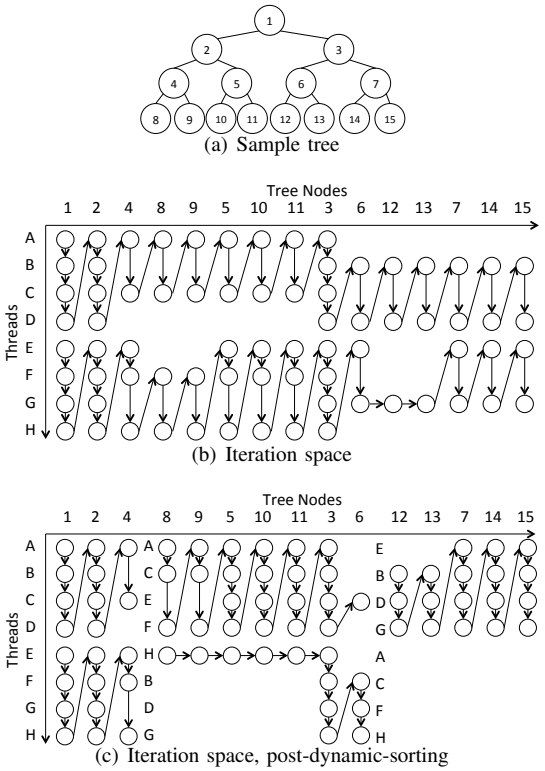


Fig. 5. Sample tree and iteration spaces

We refer to an example for illustration. Figure 5(a) shows a sample tree and Figure 5(b) shows an iteration space diagram for two SIMD packets of four threads (*i.e.,* points) each. Each thread accesses a different set of nodes. For example thread A accesses the entire subtree rooted at ②, but is truncated at ③ (due to truncation condition at line 7 of Figure 1), and does not traverse further to ⑥ or ⑦. There are three distinct sets of traversals, with different behaviors: threads $\{A, C, F, H\}$, $\{B, D, G\}$, and $\{E\}$.

SIMD utilization is poor because these distinct traversals are interleaved. At ⑧, both packets have two points. If the threads were scheduled so that A, C, F, H are executed consecutively, ⑧ could be processed with a single *full* packet of four points.

Prior work has used various sorting heuristics to schedule similar traversals together [18, 22]. Unfortunately such *a priori* sorting is highly application specific and can require not only semantic information but substantial programmer effort. We would like an application-agnostic scheduling technique.

A. Dynamic sorting

As described in Section II-B2, traversal splicing enables *dynamic sorting* that uses a thread's past behavior, rather than semantic knowledge, to continuously reorder threads so those with similar traversals will be grouped together. Figure 5(c) shows how this dynamic sorting can be applied to the iteration space of Figure 5(b). We designate ④ and ⑥ as splice nodes, at which each thread should be paused and reordered. Two SIMD packets traverse up to the first splice node ④ in their original order. At ④, the threads are reordered based on which node each thread last reached. Threads A, C, E, F and H,

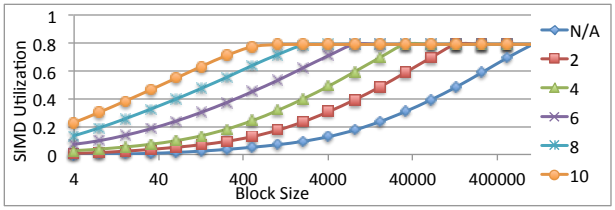


Fig. 6. SIMD utilization with dynamic sorting for Nearest Neighbor

which reached ④, are moved to the front, and threads B, D and G, which were truncated at ②, are moved to the end. These threads form two new SIMD packets which traverse up to the second splice node ⑥. Then all threads are reordered again, with threads E, B, D and G, which reached ⑥, moved to the front. These threads again form a new SIMD packet.

Dynamic sorting naturally groups threads with similar traversals together. For example at ⑦, whereas previously we had two partial SIMD packets of two points each, we now have a single full SIMD packet of four points, and hence more efficient SIMD execution. This can be generalized beyond packet-based SIMD to point blocking with a block size larger than 4. Dynamic sorting groups similar points together so that more points can be compressed to yield more full SIMD packets. Crucially, *this scheduling is performed without any application-specific knowledge*.

The primary advantage to using dynamic sorting to maximize SIMD utilization is that during the sort phase at each splice node, *all points* can be examined to rearrange the traversals, not just those in a particular block. This yields the same benefits as performing compaction on a very large block of points. However, because splicing only performs this reordering at splice nodes, locality is not harmed. Between splice nodes, smaller block sizes yield good locality.

B. Improved SIMD utilization

Recall that *ideal* SIMD utilization is defined as the utilization achieved when all threads are available for scheduling at each tree node. Figure 6 shows the SIMD utilization for pure point blocking (N/A), and point blocking combined with traversal splicing with splice nodes placed at various uniform depths[3] in the tree for Nearest Neighbor (NN). The dynamic sorting of traversal splicing can significantly improve utilization, so that we need not use as large a block size to get close to ideal. By using traversal splicing, utilization can be improved without sacrificing locality.

Figure 7 compares the SIMD utilization of various implementations of our benchmarks (see Section VI for a description of the benchmarks and inputs). We compare the baseline utilization to the utilization of three variants: (i) application-specific, *a priori sorting*[4]; (ii) automatic, *dynamic sorting* using traversal splicing[5]; and (iii) *ideal* utilization. The baseline and variants (i) and (ii) use a block size of 512 (a reasonable size to maintain locality).

It is apparent that dynamic sorting substantially improves SIMD utilization from the baseline, and is competitive with *a priori* sorting. For Vantage Point (VP), dynamic sorting is *substantially better* than *a priori* sorting, illustrating the advantages of traversal splicing. Most importantly, for most benchmarks dynamic sorting nears ideal utilization. In other words, *dynamic sorting can automatically extract almost the maximum amount of SIMD utilization from our benchmarks*.

C. Extensions to traversal splicing

In the original traversal splicing work, we found that combining point blocking and traversal splicing was frequently not useful for locality; as a result, point blocking was not applied to all partial traversals in traversal-spliced code. Because our goal is to SIMDize applications, we extend traversal splicing to support applying blocking to all phases.

Some algorithms dynamically generate new points, such as reflected and refracted photons in Photon Mapping. We handle these new points by deferring them to a subsequent outer-loop iteration. Essentially, we treat the root node of the tree as another splice node: as new points visit the root, they are paused until all earlier phases of the computation have finished, after which any points paused at the root begin traversing again. Dynamically generated points tend to have worse coherence than initial points, and dynamic sorting can be even more effective in increasing utilization for such points.

V. Automatic transformation with SimTree

We implemented our transformations in a source-to-source C++ compiler called SimTree[6]. SimTree is built on top of the ROSE compiler infrastructure[7].

SimTree applies point blocking and traversal splicing as in the Java transformation framework, TreeSplicer[8], with the extensions described in Section IV-C. A key addition in SimTree to the features implemented in TreeSplicer is a layout transformation to facilitate SIMDization that changes the layout of the points within a block from an array of structures (AoS) to a structure of arrays (SoA). SoA layout has two advantages: (i) vector load/stores (with SIMD instructions) can be used on packed data; and (ii) packed data has better spatial locality. The disadvantage is that adding points to a point block (*e.g.*, lines 7–10 in Figure 3) has higher instruction overhead, as the actual point data must be copied, instead of just a pointer to the point structure.

A whole-program AoS-to-SoA layout transformation is challenging to automate in general, because it is difficult to ensure correctness in the presence of arbitrary aliasing. We overcome this difficulty by limiting the scope of the layout transformation to the traversal code only, copying point data in to SoA-formatted blocks before the traversal and copying from those blocks back to the original layout for the remainder

[3]While arbitrary splice node placement is possible, we consider only uniform depth placement as it can be implemented more efficiently.

[4]Photon Mapping (PM) is naturally sorted in order of light source, so there is no separate *a priori* configuration

[5]Using empirically determined optimal splice depths as shown in Table I for Opteron.

[6]https://engineering.purdue.edu/plcl/simtree/

[7]http://rosecompiler.org

[8]https://engineering.purdue.edu/plcl/treesplicer/

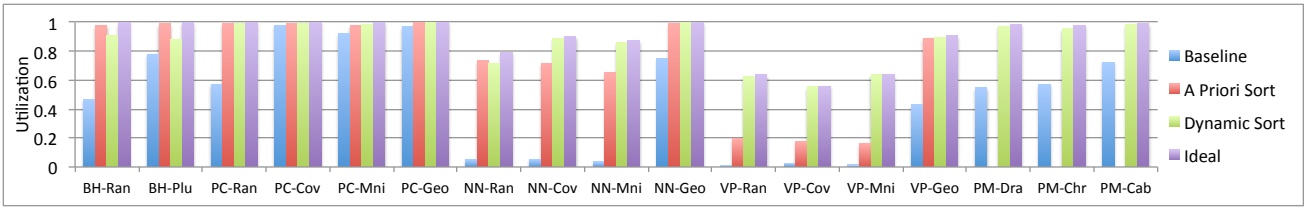


Fig. 7. Comparison of SIMD utilization

```
1  struct Point {
2    float f1, f2, f3;
3  }

5  struct TreeNode {
6    TreeNode *child1, *child2;
7    Point *point;
8  }

10 bool truncate(Point *p, TreeNode *n) {
11   if (p->f1 == n->point->f1) return true;
12   return false;
13 }

15 void updatePoint(Point *p, TreeNode *n) {
16   p->f2 += n->point->f3;
17   // p->f3 = n->point->f1;
18 }

20 TreeNode *root;
21 Point* points[N];
22 for (int i = 0; i < N; ++i)
23   recurse(points[i], root);

25 void recurse(Point *p, TreeNode *n) {
26   if (truncate(p, n)) {
27     updatePoint(p, n);
28     return;
29   }
30   recurse(p, n->child1);
31   recurse(p, n->child2);
32 }
```

Fig. 8. Concrete example traversal code

```
1  struct Block {
2    Point *p[maxSize];
3    int size;
4    void add(Point *point) {
5      p[size++] = point;
6    }
7  }
```

(a) Block as array of structures

```
1  struct Block {
2    Point *p[maxSize]; // allocated for top level only
3    float f1[maxSize];
4    float f2[maxSize];
5    int map[maxSize];
6    int size;
7    void add(Point *point) {
8      p[size] = point;
9      f1[size] = point->f1;
10     f2[size] = point->f2;
11     ++size;
12   }
13   void add(Block *upBlock, int i) {
14     f1[size] = upBlock->f1[i];
15     f2[size] = upBlock->f2[i];
16     map[size] = i;
17     ++size;
18   }
19   void copyUp(Block *upBlock) {
20     for (int i = 0; i < size; ++i) {
21       upBlock->f2[map[i]] = f2[i];
22     }
23   }
24   void copyBack() {
25     for (int i = 0; i < size; ++i) {
26       p->f2 = f2[i];
27     }
28   }
29 }
```

(b) Block as structure of arrays

Fig. 9. Block code

of the code. We need not consider aliasing between points, as the parallel point loop precludes destructive aliasing.

SIMTREE uses an inter-procedural, flow-insensitive analysis to identify which fields must be part of the SoA-formatted block. The necessary fields are those transitively accessed through loop-variant arguments to the recursive call in the enclosing point loop. For example, in Figure 8, the first argument to `recurse` in line 23 is loop variant, and hence a point argument, while the second is not. SIMTREE identifies `f1` and `f2` as point fields (as they are accessed through point argument `p` in lines 11 and 16 respectively), but not `f3`. SIMTREE allocates space in the SoA block for the point fields (Figure 9(b)) and transforms the corresponding field accesses in the recursive code to access the data in SoA form.

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

SIMTREE uses the same field analysis to identify any fields accessed via *non-point-based* access paths in the recursive code. In Figure 8, `f1` is read via a non-point argument in line 11, while `f3` is read via a non-point argument in line 16. SIMTREE deems the transformation safe as the only field accessed both through points and non-points, `f1`, is read-only.

If there is a conflict between the field accesses, SIMTREE does not apply the SoA transformation. For example, if line 17 in Figure 8 were uncommented, then the SoA transformation may not be safe: while the write to `f3` in line 17 through `p` will access the SoA block, the untransformed read of `f3` in line 16 will access the original storage. If there is aliasing between the `point` field of `TreeNode` and any of the `Points` in the array, these reads and writes may be inconsistent. While a more complex alias analysis [21] may more precisely determine the safety of SIMTREE's SoA transformation, this conservative approach is sufficient to prove the transformation safe in all of the applications we have studied.

Figures 9(a) and 9(b) show concrete code for the block before and after SIMTREE's SoA transformation. In addition

to the new arrays for the point fields, the block has two `add` functions. The first copies point data to the top level block prior to the traversal (the initial translation from AoS to SoA), and the second adds to the "next block" prior to a recursive call (lines 7–10 of Figure 3). This second `add` tracks the index of the source block in a `map` array so that as recursive calls return, writes can be correctly propagated back up the stack by calling `copyUp` (line 29 of Figure 3). `copyUp` uses `map` to copy *written fields* (*e.g.,* `f2`) up the stack. `f1` need not be copied back as it is only read. At the top level, `copyBack` (inserted after line 4 of Figure 2) copies the block back to the original point storage (translating from SoA back to AoS).

VI. Evaluation

To demonstrate the efficacy our SIMD transformations, we study five tree traversal algorithms from various domains ranging from scientific applications to data-mining to graphics[9]. The benchmarks are:

1) **Barnes-Hut (BH)** (413 LoC (lines of code)) is a scientific kernel for performing n-body simulation [1]. We use two inputs. *Random* is one million randomly generated bodies. *Plummer* is one million bodies generated from a Plummer model.
2) **Point Correlation (PC)** (285 LoC) is a data mining kernel for finding the number of pairs of points in a dataset that lie within a given radius of each other [4]. We use four inputs. *Random* has 1,000,000 randomly generated points in 3 dimensions. *Covtype* is forest cover data, and *Mnist* is handwritten digits data [7], each with 200,000 points reduced by random projections to 7 dimensions. *Geocity* is city coordinates data with 200,000 points in 2 dimensions.
3) **Nearest Neighbor (NN)** (327 LoC) is a data-mining kernel for finding closest points in metric spaces. We use the same inputs as PC, with separate training and test sets of 200,000 points each.
4) **Vantage Point (VP)** (262 LoC) is nearest-neighbor using a vantage-point tree [25] rather than a kd-tree. We use the same inputs as NN.
5) **Photon Mapping (PM)** (8498 LoC) is described in detail in Section II. We use three inputs: *Dragon* has 100,000 triangles, *Christmas* has 1,091,067 triangles and *Cabin* has 422,735 triangles. Each scene was rendered with 10 lights, and 100,000 photons per light.

BH, PC and NN are adapted from the Lonestar suite [15], and VP[10] and PM[11] are adapted from open source implementations. We evaluate seven variants of each benchmark:

1) **Base**: the baseline described for each benchmark above.
2) **Block4**: point blocking with a block size of 4.
3) **Block4+SIMD**: packet based SIMD traversals [13, 23]. This is equivalent to SoA and SIMD applied to **Block4**.

[9]We do not use exactly the same benchmarks as in our prior work [11, 12], as SimTree targets C++ rather than Java.

[10]http://stevehanov.ca/blog/index.php?id=130

[11]http://code.google.com/p/heatray

Benchmark	Input	Average Reach	Splice Depth		
			Xeon	Opteron	Auto-0.5
Barnes-Hut	Random	5.88	2	2	3
	Plummer	6.88	3	3	3
Point Correaltion	Random	17.78	8	8	9
	Covtype	15.86	7	8	8
	Mnist	15.23	7	7	8
	Geocity	15.94	9	8	8
Nearest Neighbor	Random	14.82	9	9	7
	Covtype	14.95	9	9	7
	Mnist	14.88	9	9	7
	Geocity	15.86	8	7	8
Vantage Point	Random	19.62	9	10	10
	Covtype	17.39	9	9	9
	Mnist	17.40	9	9	9
	Geocity	17.19	8	8	9
Photon Mapping	Dragon	9.13	7	8	5
	Christmas	12.55	15	13	6
	Cabin	13.14	17	18	7

TABLE I
Optimal and autotuned splice depths

4) **Block**: point blocking with an autotuned block size [11][12].
5) **Block+SIMD**: SoA and SIMD applied to **Block**.
6) **Block+Splice**: traversal splicing with dynamic sorting applied to **Block** [12]. We place splice nodes at an empirically determined optimal splice depth for each application/input/platform combination as given in Table I.
7) **Block+SIMD+Splice**: SoA and SIMD applied to **Block+Splice**.

Our baseline benchmarks are true baselines: no a priori *sorting is performed.* SIMDization is performed manually, using standard techniques such as if-conversion. While the SIMDization can be done automatically by a vectorizing compiler, such compilers may miss readily-exploitable SIMD opportunities [17], as discussed in Section VI-E.

The benchmarks were written in C++ and compiled with gcc 4.4.6 with -O3. Each configuration was run 4 times enforcing a coefficient of variation[13] less than 0.02, by extending the number of runs until steady state if necessary. This yields errors of at most $\pm 3.18\%$ of the mean with 95% confidence.

We used two platforms for evaluation:

- **Xeon** runs Linux 2.6.32 and contains four eight-core Intel Xeon E5-4650 chips.
- **Opteron** runs Linux 2.6.32 and contains four sixteen-core AMD Opteron 6282 chips.

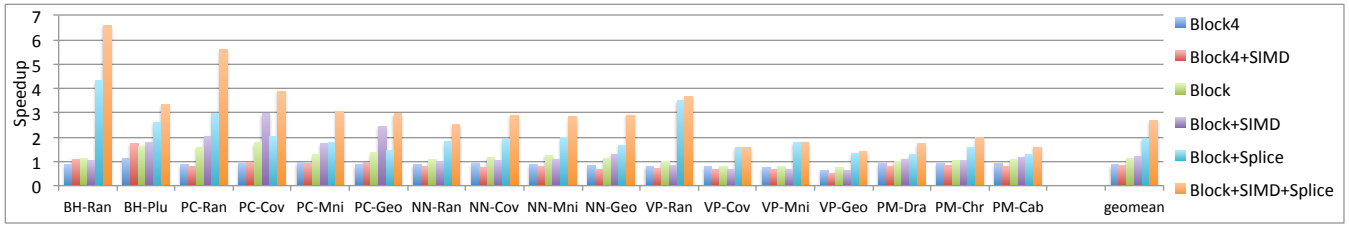
A. Speedups compared to baseline

We will first examine the *single-thread* speedups of our transformed versions compared to the baseline, shown in Figure 10. The rightmost bars are the geometric means of all 17 benchmark/input sets. We also measured multi-thread runs where both SIMD and threads are utilized, and found that SIMD improvements scale well with multiple threads: SIMD augments the parallelism achievable from multithreaded execution. We do not show these results due to space limitations.

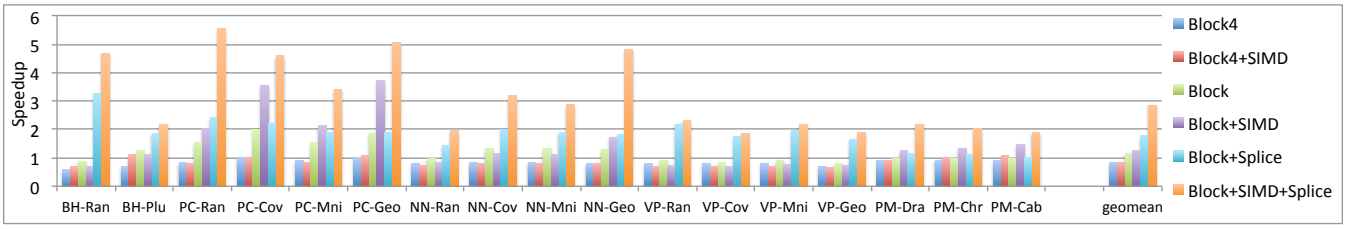
Block4 and **Block4+SIMD** perform similarly or worse than the baseline. This is because a block size of 4 does

[12]We use 1% of the points to test various block sizes 5 times each, and choose the best block size. This results in a block size of 128 for the **Covtype**, **Mnist** and **Geocity** inputs, and a block size of 512 for all other inputs.

[13]CoV is the sample standard deviation divided by the mean.



(a) Xeon



(b) Opteron

Fig. 10. Speedup over **Base**

not improve locality and only adds additional overhead for point blocking. Applying SIMD fails to provide improvements because the naïve packet based SIMD implementation has poor utilization as illustrated in Figure 4. **Block** is able to provide small speedups from enhanced locality—on average, 1.127 and 1.154 on Xeon and Opteron, respectively. **Block+SIMD** provides additional gains for speedups of 1.188 and 1.266 respectively. This is because point blocking with compression improves utilization so that SIMD execution is profitable.

Block+Splice adds the locality benefits of traversal splicing to **Block** and results in larger speedups of 1.920 and 1.783. **Block+SIMD+Splice** adds SIMD to **Block+Splice** for substantially larger speedups of 2.689 and 2.864. The average improvement from adding SIMD to **Block+Splice** is much larger than from adding SIMD to **Block**, because the dynamic sorting of traversal splicing dramatically improves utilization, as illustrated in Figure 7. For example in NN-Random, adding SIMD to **Block** *degrades* performance because SIMD execution is unable to make up the instruction overhead of the structure-of-arrays layout transformation. This is unsurprising as SIMD utilization is only 0.052 even for a block size of 512. However dynamic sorting improves utilization to 0.716, making SIMDization of **Block+Splice** profitable.

B. Performance counters

Our optimizations achieve speedups because the locality and SIMD execution benefits they provide outweigh the costs incurred by our scheduling and layout transformations. To analyze this cost-benefit tradeoff in more detail, we collected performance counter results on Opteron with PAPI[14].

One indicator of the effectiveness of SIMDization is instruction count reduction. Figure 11(a) shows the instruction counts normalized to the baseline. **BlockSOA** denotes the structure of arrays (SoA) transformation added to **Block**. On average, point blocking results in instruction overhead of 1.266. The SoA transformation increases this overhead to 1.617. Adding SIMD decreases instruction overhead to 1.204 (as SIMD replaces multiple instructions with a single instruction). Adding traversal splicing to point blocking *reduces* instruction overhead to 1.079 (from 1.266), because the dynamic sorting makes the blocks denser and results in fewer blocks traversing the tree. Finally adding SoA increases the instructions to 1.380, but then adding SIMD halves it to 0.643.

The savings in instruction overhead from adding SIMD is much larger with traversal splicing (2.14×) than without (1.34×), as traversal splicing significantly enhances SIMD utilization. We do not expect instruction counts to decrease by a full-SIMD-width factor of 4 because: (i) we measure instruction counts for the whole traversal, while SIMD only affects the computation portion; (ii) SoA adds instruction overhead; and (iii) SIMD utilization is less than 1.

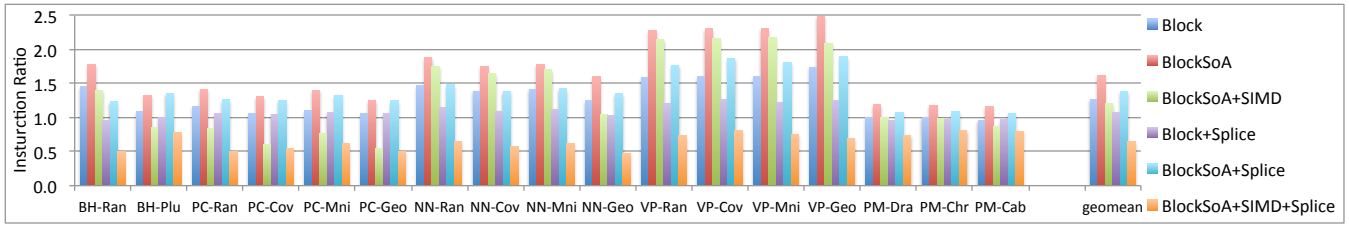
Figure 11(b) shows the cycles per instruction (CPI) of the transformations with and without SoA layout and/or traversal splicing. CPI is a proxy for locality: a program with poor locality will have higher CPI[15]. The average baseline CPI is 2.243. Point blocking and traversal splicing reduce this to 1.563 and 1.170 respectively through enhanced locality. The SoA layout also improves spatial locality as the point data is packed into the blocks instead of being spread out over the heap. This further reduces CPI to 1.349 and 1.043 when applied on top of point blocking and traversal splicing respectively. Hence, a combination of instruction decrease through SIMD execution, and enhanced locality through SoA layout accounts for our performance improvements.

C. SIMD performance improvement from dynamic sorting

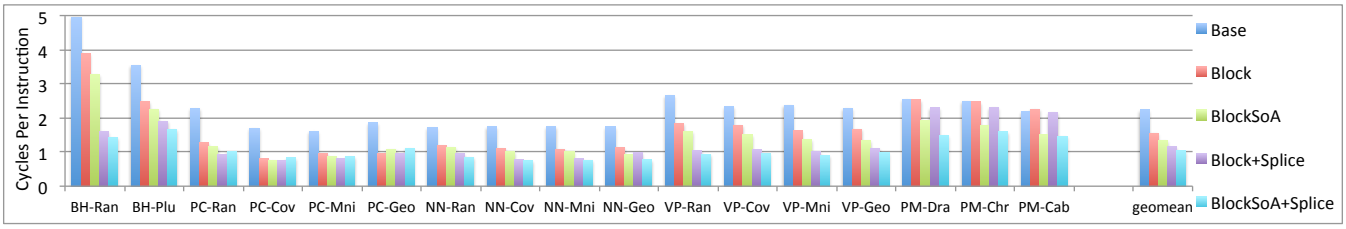
Because traversal splicing provides both locality benefits and SIMD utilization benefits, we isolate how much dynamic sorting in particular improved SIMD performance. Let us define the speedups of **Block**, **Block+SIMD**, **Block+Splice** and **Block+SIMD+Splice** compared to the baseline as B_b, B_{bs}, B_{bp} and B_{bsp} respectively (where p is used for S*p*lice).

[14]http://icl.cs.utk.edu/papi/

[15]This CPI comparison does not use SIMD; we expect **BlockSOA** and **BlockSOA+SIMD** to have similar locality.



(a) Instruction counts normalized to **Base**



(b) Cycles per instruction

Fig. 11. Performance counters

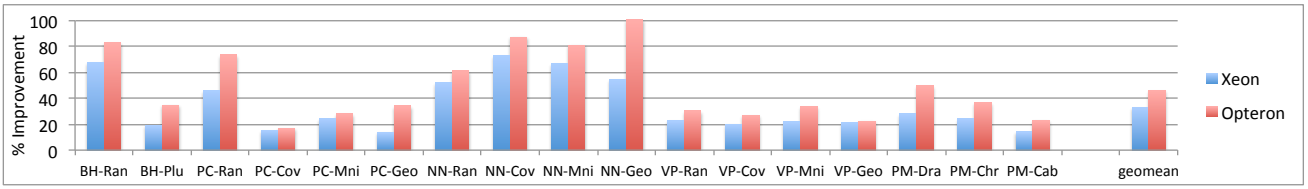


Fig. 12. Improvement in SIMD performance from dynamic sorting

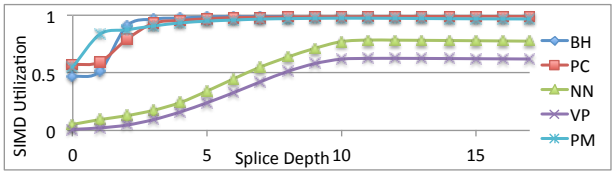


Fig. 13. SIMD utilization for various splice depths ($B = 512$)

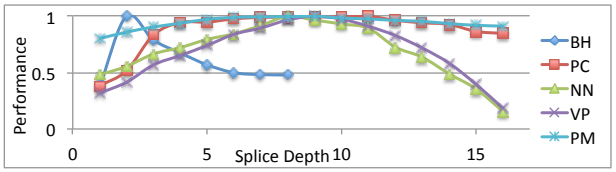


Fig. 14. Performance normalized to optimal splice depth ($B = 512$)

The benefit of applying SIMD to **Block** and **Block+Splice** can be defined as $S_b = B_{bs}/B_b$ and $S_{bp} = B_{bsp}/B_{bp}$, respectively. The ratio of SIMD performance improvements in both cases, $D = S_{bp}/S_b$, lets us quantify how much dynamic sorting improves SIMD quality independent of locality effects (which apply to both SIMD and non-SIMD variants).

Figure 12 shows D in % improvement for the two systems. Dynamic sorting is ineffective for BH-Plummer because the baseline has a high utilization of 0.779, and dynamic sorting is able to improve that only to 0.882 (Figure 7). Dynamic sorting is very effective for NN-Geo because the baseline has a utilization of 0.747 which dynamic sorting improves to 0.995, nearly perfect utilization. On average D in % improvement is 33.0% and 46.4% respectively for Xeon and Opteron.

D. Automatically selecting splice depth

The results discussed thus far have used an autotuned block size with an empirically determined optimal splice depth. We would like to automatically select the splice depth as well, to fully automate our transformation.

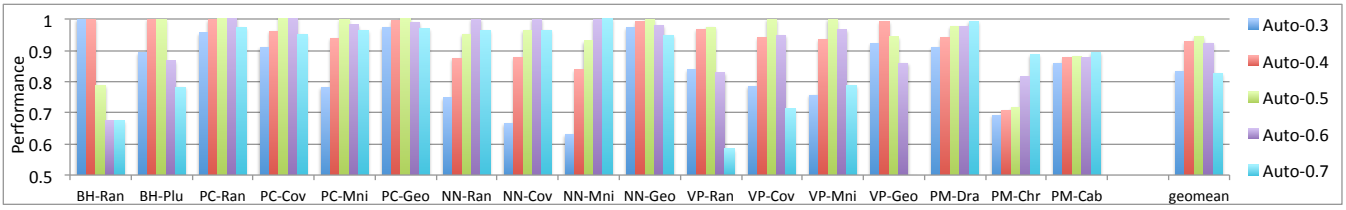
While a good block size can be autotuned by using a small portion (1%) of the points to test blocks of varying sizes, and choosing the best performing block size [11], a similar approach is infeasible for traversal splicing because we would like to apply splicing over the full set of points.

When considering locality, there is a tradeoff in splice depth placement: placing splice nodes too shallow or too deep results in worse locality, and there is a sweet spot in the middle [12]. On the other hand, utilization increases almost monotonically with splice depth, as deeper and more numerous splice nodes allow for more dynamic sorting[16]. Figure 13 shows utilization for varying splice depths, with a fixed block size of 512 (using the same inputs as in Figure 4). We expect to see the best performance when striking the right balance between locality and SIMD utilization.

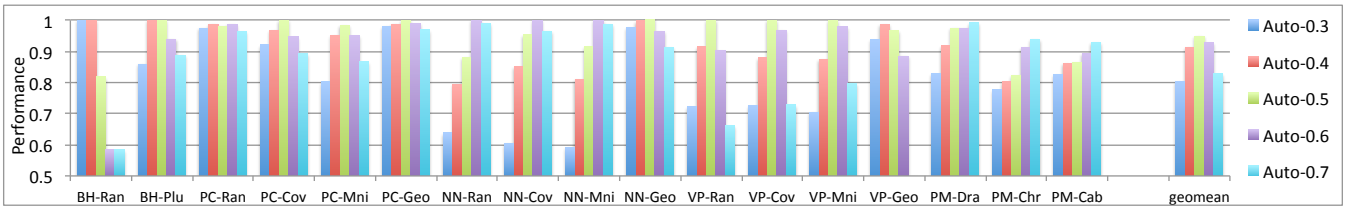
Figure 14 shows the performance at varying splice depths normalized to the performance obtained with the optimal splice depth for each benchmark. We see a clear peak where utilization and locality strike a good balance, surrounded by splice depths which perform worse.

In prior work we demonstrated that combining point blocking with traversal splicing reduces a program's sensitivity to splice depth, and found empirically that a depth at half of the average reach (the depth of the nodes where a point's recursion is stopped) is a good splice depth [12]. We evaluate a range of parameters for splice depth, ranging from 0.3 to 0.7 times the average reach. Figure 15 shows the performance of these

[16]Utilization can decrease if the splice depth is deeper than the tree itself.



(a) Xeon



(b) Opteron

Fig. 15. Performance at automatically selected splice depth normalized to performance at optimal splice depth

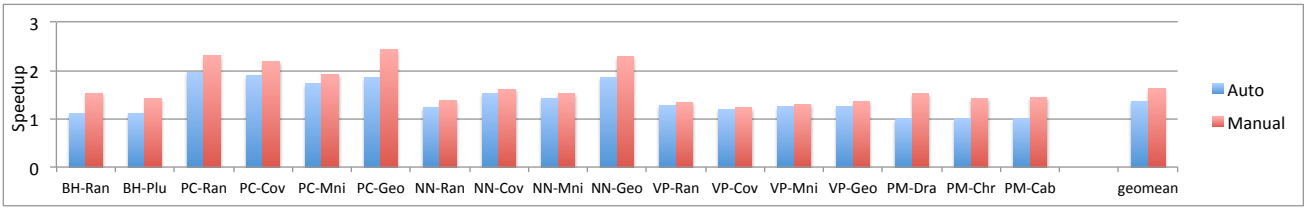


Fig. 16. SIMD speedup from automatic vectorization

parameters compared to the performance at an optimal splice depth. Table I shows the average reach of each benchmark/input, the optimal depth on each system, and the automatically selected depth at half of the average reach. It can be seen that **Auto-0.5** performs best, attaining 95% of optimal performance on average on both systems.

E. Automatic vectorization with icc

SIMTREE does not generate SIMD code. As described previously, SIMTREE's goal is to generate *vectorizable* code; the resulting inner loop (line 17 of Figure 3) must be vectorized in a separate pass. To extract maximum SIMD performance in our results, we vectorized this loop by hand. We also investigated the ability of icc (the Intel compiler) to do the vectorization. Because the presence of any non-vectorizable code or nested loops precludes vectorization by icc [17], we applied a combination of loop distribution and interchange to split the target loop into a series of icc-vectorizable loops (this process can be readily automated as the target loop is fully parallel). Figure 16 shows the performance of icc-vectorized code and hand-vectorized code for the main SIMD loop (ignoring the cleanup loop), relative to the SIMTREE-transformed baseline (which applies blocking, splicing and SoA) on Xeon. We see that auto-vectorization attains a 1.354 speedup, while our hand vectorization attains a 1.618 speedup; auto-vectorization attains on average 84% of the hand-vectorized performance.

VII. RELATED WORK

Much of the work on using SIMD for tree traversals has come from the graphics domain. Prior work proposes various techniques for manual, application specific SIMDization of ray-object intersection test traversals. Wald *et. al.* present an interactive ray tracer by using packet-based SIMD traversals on packets of four rays [23]. Such packet-based SIMD traversals work well for coherent eye rays, but break down for secondary rays, which are necessary to model realistic lighting effects. To circumvent this incoherency issue, subsequent work has looked to other sources of SIMD. Dammertz *et. al.* use a quad-BVH (bounding volume hierarchy with four children per node) and SIMDize computation across the four children for each *individual* ray [6]. Pixar used SIMD to test all three dimensions (x, y, z) of a bounding box at once in rendering the movie "Cars" [5]. Havel and Herout propose a modified ray-object intersection algorithm which is better suited for SSE4 which supports a SIMD dot product instruction [8]. They note that applying SIMD to individual rays get only 10-25% improvement whereas applying SIMD to coherent packets can get 300%. Our dynamic sorting can make incoherent packets coherent, and opens up opportunities for substantial performance gains in the ray tracing domain. Furthermore our technique can be applied automatically.

In the database domain, Kim *et. al.* propose FAST (Fast Architecture Sensitive Tree) to accelerate index searches by using SIMD to compare a search value to N keys at once, where N is the number of keys that can fit in a SIMD register [13]. For example, with $N = 4$, a point can compare with a node, the node's left child, and the node's right child at once, and decide which grandchild to move on to. Yamamuro *et. al.* extend FAST by compressing keys at lower depths so that more keys can be compared at once [24]. These techniques capitalize on specific properties of index search: that each point takes a single path from root to leaf, and that the point-key comparison can be done at multiple levels simultaneously.

For Barnes-Hut, Hernquist vectorizes across the nodes of the tree so that each point accesses all nodes at the same level simultaneously, effectively changing the traversal from depth-first to breadth-first [9]. Makino vectorizes across the points, in a manner similar to the loop interchange described in Section III [16]. For point correlation, Chhugani *et. al.* propose a novel SIMD friendly histogram update algorithm, exploiting an alternate source of SIMD instead of vectorizing the traversals themselves [4].

Ren *et. al.* focus on applications where points traverse *many* pointer based data structures (*e.g.,* random forests, regular expressions), and *manually* SIMDize a single point visiting nodes of many structures [20]. Kim and Han propose techniques for efficient SIMD code generation for irregular loops with array indirection [14]. Barthe *et. al. synthesize* SIMD code given pre/post conditions for irregular loop kernels in C++ STL or C# BCL [2]. The latter two works do not consider pointer-based irregular structures.

Nuzman and Zaks examine a direct unroll-and-jam approach for vectorization of outer loops which could help auto-vectorize our transformed point loop [19]. Maleki *et. al.* perform a thorough study of modern vectorizing compilers [17]. They find that current compilers are still lacking: xlc, icc and gcc are able to individually vectorize only 18-30% of loops extracted from real codes, while they can collectively vectorize 48%. In particular, they note that traditional vectorizing compilers primarily target regular loops; our transformations hence open traversal codes to automatic vectorization.

VIII. Conclusions

We presented automatic transformations to reschedule tree traversal algorithms to effectively SIMDize them. Our scheduling techniques build on previous work that enhanced locality for these algorithms. Point blocking exposes a loop structure within the traversal function which can be SIMDized (as in manually transformed packet-based SIMD traversals), and compacts points to improve SIMD utilization. Traversal splicing pauses points at predesignated nodes, and reorders the points based on their past traversal history. This dynamic sorting groups points with similar traversals together so that SIMD utilization is further enhanced.

We demonstrated that dynamic sorting dramatically improves SIMD utilization to the point that it is very close to ideal. This enhanced utilization results in significantly better SIMD performance, yielding average speedups of 2.78 over the baseline. Our techniques can deliver SIMD speedups even in algorithms where the incoherency of points have previously forced researchers to look for other sources of SIMD.

Acknowledgments

This work was supported in part by an NSF CAREER Award (CCF-1150013) and grants from the Purdue Research Foundation and Intel. The authors would like to thank Malek Musleh and Vijay Pai for providing support on the Opteron machine. We are grateful to Todd Mytkowicz and the anonymous referees for insightful discussions.

References

[1] J. Barnes and P. Hut. A hierarchical $o(nlogn)$ force-calculation algorithm. *Nature*, 324(4), 1986.

[2] G. Barthe, J. M. Crespo, S. Gulwani, C. Kunz, and M. Marron. From relational verification to simd loop synthesis. In *PPoPP*, 2013.

[3] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18, 1975.

[4] J. Chhugani, C. Kim, H. Shukla, J. Park, P. Dubey, J. Shalf, and H. D. Simon. Billion-particle simd-friendly two-point correlation on large-scale hpc cluster systems. In *SC*, 2012.

[5] P. Christensen, J. Fong, D. Laur, and D. Batali. Ray tracing for the movie 'cars'. In *RT*, 2006.

[6] H. Dammertz, J. Hanika, and A. Keller. Shallow bounding volume hierarchies for fast simd ray tracing of incoherent rays. In *EGSR*, 2008.

[7] A. Frank and A. Asuncion. UCI machine learning repository, 2010.

[8] J. Havel and A. Herout. Yet faster ray-triangle intersection (using sse4). *Visualization and Computer Graphics, IEEE Transactions on*, 2010.

[9] L. Hernquist. Vectorization of tree traversals. *J. Comput. Phys.*, 87, 1990.

[10] H. W. Jensen. Global illumination using photon maps. In *Eurographics workshop on Rendering techniques*, 1996.

[11] Y. Jo and M. Kulkarni. Enhancing locality for recursive traversals of recursive structures. In *OOPSLA*, 2011.

[12] Y. Jo and M. Kulkarni. Automatically enhancing locality for tree traversals with traversal splicing. In *OOPSLA*, 2012.

[13] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *SIGMOD*, 2010.

[14] S. Kim and H. Han. Efficient simd code generation for irregular kernels. In *PPoPP*, 2012.

[15] M. Kulkarni, M. Burtscher, K. Pingali, and C. Cascaval. Lonestar: A suite of parallel irregular programs. In *ISPASS*, 2009.

[16] J. Makino. Vectorization of a treecode. *J. Comput. Phys.*, 87, March 1990.

[17] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua. An evaluation of vectorizing compilers. In *PACT*, 2011.

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

[19] D. Nuzman and A. Zaks. Outer-loop vectorization: revisited for short simd architectures. In *PACT*, 2008.

[20] 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*, 2013.

[21] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, 1999.

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

[23] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive rendering with coherent ray tracing. In *Computer Graphics Forum*, 2001.

[24] T. Yamamuro, M. Onizuka, T. Hitaka, and M. Yamamuro. Vast-tree: a vector-advanced and compressed structure for massive data tree traversal. In *EDBT*, 2012.

[25] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA*, 1993.