SCI INSTITUTE
TECHNICAL REPORT

SIMD Ray Stream Tracing - SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering -

Ingo Wald[*†], *Christiaan Gribble*[‡], *Solomon Boulos*[§] *and Andrew Kensler*[*]
[*]SCI Institute, University of Utah [†]currently with Intel Corp
[‡]Grove City College [§]School of Computing, University of Utah

UUSCI-2007-012

Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA

Aug 2, 2007

Abstract:

Achieving high performance on modern CPUs requires efficient utilization of SIMD units. Doing so requires that algorithms are able to take full advantage of the SIMD width offered and to not waste SIMD instructions on low utilization cases. Ray tracers exploit SIMD extensions through packet tracing. This re-casts the ray tracing algorithm into a SIMD framework, but high SIMD efficiency is only achieved for moderately complex scenes, and highly coherent packets. In this paper, we present a stream programming oriented traversal algorithm that processes streams of rays in SIMD fashion; the algorithm is motivated by breadth-first ray traversal and implicitly re-orders streams of rays on the fly by removing deactivated rays after each traversal step using a stream compaction step. This improves SIMD efficiency in the presence of complex scenes and diverging packets, and is, in particular, designed for potential wider-than-four SIMD architectures with scatter/gather support.

THE UNIVERSITY OF UTAH

Technical Report No UUSCI-2007-012

SIMD Ray Stream Tracing
— SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering —

Ingo Wald[12], Christiaan Gribble[3], Solomon Boulos[4], and Andrew Kensler[1]

[1]SCI Institute, University of Utah
[2]currently with Intel Corp
[3]Grove City College
[4]School of Computing, University of Utah

ABSTRACT

Achieving high performance on modern CPUs requires efficient utilization of SIMD units. Doing so requires that algorithms are able to take full advantage of the SIMD width offered and to not waste SIMD instructions on low utilization cases. Ray tracers exploit SIMD extensions through packet tracing. This re-casts the ray tracing algorithm into a SIMD framework, but high SIMD efficiency is only achieved for moderately complex scenes, and highly coherent packets. In this paper, we present a stream programming-oriented traversal algorithm that processes streams of rays in SIMD fashion; the algorithm is motivated by breadth-first ray traversal and implicitly re-orders streams of rays "on the fly" by removing deactivated rays after each traversal step using a stream compaction step. This improves SIMD efficiency in the presence of complex scenes and diverging packets, and is, in particular, designed for potential wider-than-four SIMD architectures with scatter/gather support.

1 INTRODUCTION AND MOTIVATION

Ray tracing is a widely used technique in rendering, and is known for its ability to produce highly convincing images. Though originally an off-line algorithm, the exponential growth in compute power has made interactive ray tracing possible on commodity hardware. While the compute power of commodity CPUs continue to grow, this no longer happens through increasing the processor clock rate. More likely, future CPUs will focus less on serial performance and may even decrease the clock rate of processors. The CPU road-map indicates that future processors will increasingly rely on parallelism through multi-core architectures. For an algorithm that is as parallel as ray tracing, exploiting multiple cores is rather straightforward.

1.1 SIMD Parallelism

On a finer scale, CPUs offer a way of exploiting data level parallelism (ILP) using SIMD extensions like Intel's SSE or IBM/Motorola's Altivec. SIMD instructions provide parallelism by executing the *same* instruction in parallel on multiple data items. From a processor designer's standpoint, SIMD instructions are a cheap way of increasing a processor's compute performance: since all data are subject to the same instruction, large parts of the core's logic can be shared among the multiple data paths in the wide units. Said more simply, making an arithmetic unit n-wide is much cheaper than making n copied arithmetic units.

Both SSE and Altivec expose a relatively small *SIMD width* of four single precision floating point elements. As the complexity of mapping an algorithm to a SIMD programming model increases with the SIMD width, a small SIMD width of four offers a nice trade-off between SIMD benefit and mapping complexity. However, given the small cost and potentially high benefit of increasing the SIMD width, it seems likely that future architectures will explore larger SIMD widths.

Though not immediately obvious, one such example is NVidia's G80 architecture. Though the G80 is commonly viewed as a massively multi-threaded *scalar* architecture, each *warp* of 32 "threads" is essentially run in a SIMD fashion: high performance can only be achieved if all of a warp's threads execute the same instruction. This essentially makes the G80 behave more like a 32-wide SIMD architecture than the 128-core scalar architecture it is often believed to be. It seems quite possible that other architectures will follow the same trend and eventually use wider-than-four SIMD instructions.

1.2 SIMD Extensions in Ray Tracing

Unfortunately, SIMD instructions are far harder to exploit in software than they are to add in hardware. Except for few notable exceptions like linear algebra operations, algorithms often have to be completely re-designed to make use of these SIMD instructions, which often is far from trivial.

In ray tracing, the core concept that allowed the efficient use of SIMD instructions was the introduction of *packet tracing* (aka Coherent Ray Tracing) [24]. In packet tracing, rays are no longer traced sequentially (and individually) through the acceleration data structure. Originally, packet tracing was introduced for kd-tree traversal and triangle intersection only [24], but has been extended to different primitive types and acceleration structures (see, e.g., [3, 12, 13, 23]). Packet tracing has also been successful for special-purpose ray tracing hardware [20, 25]. Having large packets of rays also allows for algorithmic optimizations like frustum culling [19] or interval arithmetic [7, 22]; as these exploit the same kind of ray coherence that SIMD ray tracing does—but do so in scalar or low-width SIMD form—frustum or interval arithmetic techniques get particularly problematic for wider SIMD width. In this paper, we will not consider frustum or interval arithmetic techniques, and will *only* consider SIMD ray tracing.

1.3 SIMD Efficiency and SIMD Packet Tracing

While quite successful, packet tracing is efficient only for highly *coherent* packets in which all the rays in the packet will execute the same traversal steps and primitive intersections. As soon as rays *diverge* during traversal, some of the rays become *inactive* during traversal. Though we still perform N operations with every instruction (where N is the SIMD width), only a smaller portion ($n < N$) of these operations are performing useful work on active rays. The remaining slots of the SIMD instruction ($N - n$) are performing wasted work and would not have occured in a single ray based system. This allows us to define the *SIMD efficiency* as the relative number of useful operations: $\frac{n}{N}$.

In the worst case, only a single ray remains active leading to a worst-case efficiency of $\frac{1}{N}$. Thus larger SIMD widths can lead to

lower utilization and efficiency. This is perhaps the primary reason that architectures may not increase their SIMD width: utilization is key in hardware design.

The number of active rays in a given traversal or intersection step depends on many parameters. Obviously the more coherent the rays are, the higher the resulting SIMD efficiency. Secondary rays are, usually, less coherent than primary rays, but the exact degree of coherence is hard to predict. Boulos et al. have reported very similar rays/second rate for distribution ray tracing (including soft shadows, motion blur, depth of field, and glossy reflections [6]) as for Whitted-style ray tracing; on the other hand, Reshetov has shown that even for a small SIMD width of four, and even for perfectly specular reflections in moderately complex scenes, multiple-bounce reflections can quickly lead to almost completely incoherent ray packets and $\frac{1}{N}$ SIMD efficiency.

1.4 Outline

In this paper, we propose a new approach to SIMD ray tracing that combines elements from stream computing, packet tracing, breadth-first ray tracing, and ray re-ordering. In Section 2, we briefly summarize related work; Section 3 introduces the core ideas of our approach, followed with its application to BVH traversal and triangle intersection in Section 4. Section 5 provides extensive experimental data for both our algorithm and traditional packet tracing, followed by a detailed discussion in Section 6. Finally, we summarize and conclude in Section 7.

2 Related Work

SIMD packet tracing The use of packets to support SIMD extensions was first introduced by Wald et al. under the name of "Coherent Ray Tracing" [24]. Wald et al.'s original implementation was targeted towards the SSE instruction set with a SIMD width of 4, and consequently used packets of 4 rays. Later implementations also used larger packet sizes of 4×4 rays [2], but kept the same concept of fixed-size packets that never got split, shortened, or reordered.

Packet tracing was originally proposed for kd-trees and triangle primitives [24], but was later used for other primitive types including iso-surfaces [15], free-form surfaces [4], and for other acceleration structures like bounding volume hierarchies [13].

In his Multilevel Ray Traversal Algorithm, Reshetov et al. [19] proposed using much larger packets than used in traditional packet tracing, and to split packets on the fly during traversal. Splitting was performed in screen space, and extending the concept to general ray packets is an open problem. In addition, MLRT traversal uses a combination of frustum culling and interval arithmetic to reduce the number of traversal steps, but the resulting operations are inherently scalar, and do not work well in wide SIMD. Similarly, the large-packet schemes proposed for grids [23] and BVHs [22] are either scalar or 4-wide SIMD.

Ray re-ordering Ray reordering for coherence was first investigated by Pharr et al. [16]. This reduced disk operations when rendering massively complex scenes that would not fit into main memory, but no consideration for finer scales (e.g. L2 caches) was considered.

Breadth-first ray tracing Instead of tracing rays depth-first through the hierarchy, multiple researchers have investigated breadth-first traversal concepts. Mahovsky [14] has proposed breadth-first traversal of BVHs to render massively complex models through progressively compressed BVHs. This approach, however, used breadth-first traversal to amortize BVH decompression cost, and did not target real-time rates or SIMD.

3 Basic Computing Paradigm

The two core concepts in this paper are *streams* of rays (or ray IDs), and sets of *filters* that successively extract *sub-streams*–with certain properties–from a parent stream. For example, a BVH traversal filter may extract a sub-stream of rays that intersect a given BVH node and passes only this sub-stream to child nodes.

A stream only contains data of the same type (say, a stream of rays, or a stream of IDs), but can be of arbitrary length. Streams are processed sequentially, albeit in SIMD manner[1]; non-sequential memory accesses are possible through *gather read/scatter write* operations (in which a stream of non-sequential addresses or ray IDs is used to generate a sequential stream of data).

For ray tracing, our core observation is that most operations in traversal, intersection, and shading can be written as a sequence of conditionals that are applied to each ray. Instead of applying conditional statements to individual rays, we can execute a conditional as a stream filter across an entire packet of rays. Given the arbitrary ordering and length of the streams, we can then effectively use SIMD processing for long streams. An important purpose of this work is to demonstrate that such long streams exist for ray tracing.

Stream Filtering Taking an input stream of rays and extracting those with a given property into a resulting sub-stream is what we refer to as a *filter*. In a traditional non-SIMD architecture, a filter operating on a given input stream would be written as

```
filter<test>(inputstream) -> outputstream {
  for each e in inputstream
     if (test(e) == true)
        outputstream.push(e)
}
```

In a data parallel or SIMD environment, the stream filter can be implemented as a two-step process: Instead of processing each element individually, many elements of the input stream are tested in parallel, and a Boolean output *mask* is computed for the input stream. This mask is then used to *compact* the input stream to only those elements whose mask is "true". Ideally, the compaction operation would be supported in hardware (e.g., a SIMD vector compaction operation in our hypothetical SIMD architecture); otherwise, efficient data-parallel algorithms for compacting an entire stream (typically based on parallel prefix sums) are available in both the data parallel [5] and in the GPGPU literature [10].

SIMD Sub-stream Processing If the underlying SIMD instruction set supports gather read operations, the above successive stream filtering paradigm can be very efficiently implemented in SIMD. We start with an initial input stream of data, and generate an ID list $(0,1,2...M-1)$, where M is the number of elements in the initial stream. In each subsequent filter operation, we process the input stream of IDs and compute an output stream of IDs of those elements which "pass" the filter.

```
filter<test>(element[M],inputstream<IDs>)
                    -> outputstream<IDs> {
  for each ID in inputstream
     if (test(element[ID}) == true)
        outputstream.push(ID)
```

[1] All stream elements are independent from each other, and can be processed in arbitrary order. In particular, an arbitrary number of elements can be processed in parallel, which allows us to process the stream in arbitrarily-wide SIMD

```
}
```

Given that a gathering read is available to a SIMD unit, we can process the entire stream in a SIMD fashion. This is due to the single level of indirection available for each filter and the independence of the elements of the stream. For any filter, we follow the following steps:

1. Read a SIMD wide chunk of ray IDs from the stream
2. Collect data elements via a gather read operation
3. Apply the filter to the SIMD wide chunk
4. Generate and compact a portion of output based on the filter mask
5. Append the new "active" IDs to the full output-stream

In this basic outline, we assume that our hardware provides efficient compaction on a SIMD chunk. If this is not the case, we could instead generate a stream length mask and perform a traditional stream compaction operation as mentioned above.

There are two main reasons for this stream processing approach. First, if input streams are larger than the available SIMD width we will usually operate on *N* items in parallel and achieve high utilization. Second, the filtering removes elements from our stream that would perform useless work. This allows subsequent filter operations to operate only on active elements and yields higher utilization.

The main requirement for this stream processing approach is an instruction set supporting gather reads. Though this is not currently supported in traditional SIMD instruction sets such as SSE, we believe that future hardware will support this operation. NVidia's G80, if viewed as a 32-wide SIMD architecture, already does support gather and scatter, and so does—albeit at a much different scale—Intel's SSE4. Hardware support for compaction/reduction would also be desirable, but current prefix-sum based reduction for full streams would probably perform similarly.

4 SIMD Stream Tracing

Once the basic stream approach is understood, applying it to BVH traversal, primitive intersection, and shading is fairly straightforward. We now specifically detail each of these operations in sequence.

4.1 Traversal

For traversal, our initial input stream is a large number of rays, ideally far larger than the SIMD width—say, 1K rays. We then generate the initial ID stream $(0,1,2,\ldots)$ and start recursive BVH traversal. In each traversal step, we take the input stream of ray IDs, test all of the corresponding rays with the current BVH node's bounding box, and reduce the ID stream to produce the output stream, which contains the IDs of all those rays that want to traverse that sub-tree. If this stream is empty, we backtrack to the last sub-tree on the stack; else we either intersect the output stream with the node's triangles if it is a leaf node, or we recursively traverse the output stream down the node's children, ideally in front-to-back order.

In pseudo-code, the algorithm can be written very compactly:

```
BVHTraverse(ray[M]) {
  stream init = (0,1,2,...M-1);
  traceStream(root,ray,init);
}
recTrv(node,ray[M],inputIDs) {
  activeIDs = filter<boxtest(node)>(inputIDs);
  if (empty(activeIDs)) return;
  if (isLeaf(node))
    for (all triangles t)
      intersectStreamWithTriangle(t,ray,activeID);
  else
    traceStream(firstChild(node),ray,activeIDs);
    traceStream(secndChild(node),ray,activeIDs);
}
```

Here, *firstChild* and *secondChild* are the first and second child to be traversed, with order being determined through any given heuristic, ideally through ordered traversal based on the first active ray's direction vector signs. Note that the *filter* operation above is where most of the time is spent, and that this operation is executed strictly in SIMD manner.

All in all, the code is very simple, looks almost like single-ray traversal (except that we work on variable-length streams of rays), and can be written in a few lines of code.

4.2 Intersection

In the most simple way, the triangle intersection code takes a SIMD chunk of *N* ray IDs, *gather read*'s the respective rays, and performs *N* complete SIMD ray/triangle intersections in parallel. This results in an *N*-wide mask of which ray-triangle intersections have been successful, which is then used to store the respective rays' new hit information using conditional masks. The new hit information is then written back into the initial ray population using a scatter write. As with gather reads, we assume scatter writes are supported by the hardware.

Instead of performing entire triangle intersections in SIMD chunks, one can also write the triangle test itself as a sequence of filter operations. This should yield higher SIMD utilization than simply performing complete ray/triangle tests in SIMD fashion. Though there are a variety of triangle tests, any method will eventually have to perform a distance test to determine whether the distance to the triangle is within the valid ray interval. Furthermore, any test will need to determine that the ray passes through the inside of the triangle (e.g., using barycentric coordinates or Plücker tests against each edge). Using filters, each of these four tests can be applied in succession with every subsequent test only operating on active rays that have passed previous tests. This sequence of filters approach would provide higher SIMD efficiency for large streams of rays, but also requires more gather and compaction operations. Which of the two variants performs better will depend strongly on the actual hardware used to execute them.

4.3 Shading

By applying a filter of shader ids to input streams, we can generate streams of rays that all execute the same shader. To do so, we grab the first element of the stream and apply a filter to separate the stream into rays that match the shader id and those that do not. The rays to be shaded are all passed to the shader at once, which may apply further filters during shading. The rays that did not match the original shader id now form a stream of "remaining rays" that require shading for other shader ids. We continue this process until the "remaining rays" stream is empty and we have shaded all input rays.

4.4 Managing Stream Memory

In the examples above, we have abstracted memory management and assumed that we have a memory system that supports a "push"

into an output stream at any time. In our implementation, we currently use one large chunk of pre-allocated stream memory. Each new output stream is started immediately after its "parent" stream, and a stream is represented with a pair of pointers that point to the beginning and end of the stream elements. In this setup, only only the head and tail need to be pushed onto a stack. This makes creating, freeing and appending stream elements trivial. The total required stream memory is then bounded by the size of the input ID stream multiplied by the maximum number of successive reduction operations. For all operations we have considered this has been trivial to compute; for example, a BVH traversal will at most generate D sub-streams where D is the maximum depth of the tree.

Alternatively, since each filter output is a *sub*-stream of the input stream, the stream can be sorted "in place". Elements that pass the filter are moved to the start of the stream, and those that fail are moved to the end (possibly using double-buffering to simplify read/write conflicts to the input stream). In this setup, a filter operation is similar to a quick-sort "partition" step (also see, e.g., [5]). As our algorithm does not depend on the ordering of the elements, the ensuing permutations of the ray IDs in the ID streams are not a problem. Though actual real-time implementations of our approach will likely use in-place sorting, our simulator is neither time- nor space-critical and does not use in-place sorting, yet.

4.5 Expected Behavior

Before interpreting actual measurements, it is worthwhile to first discuss the *intended* behavior of our algorithm.

The main intention of the algorithm is to increase the SIMD efficiency of SIMD ray traversal by filtering out inactive rays immediately upon detecting that they miss a sub-tree. In a certain sense, compacting the output stream then fills up the gaps caused by deactivated rays, which obviously means that some other rays need to be available. This requires that the initial stream is sufficiently long. If we are always able to find enough rays to feed into the next traversal step, SIMD efficiency should be significantly higher than for traditional packet traversal.

Implicit on-the-fly re-ordering. This filtering at each traversal step can also be viewed as on-the-fly reordering of rays during traversal. In fact, the reordering performed is in some sense *optimal* with respect to the original stream of rays: *all* rays from the original stream that would perform the same traversal or primitive intersection will always perform that operation together. It is important to note that this is true despite the order in which the rays occurred in the initial input stream or which path rays take to reach the common operation. Thus, no other re-ordering algorithm would be able to combine more operations of the same kind than our algorithm given the same input. By definition, this reordering scheme is optimal wrt. this efficiency measure. While this does not imply that actual performance will be optimal, we find it to be a nice theoretical point.

Additionally, this on-the-fly reordering does not rely on either costly pre-sorting nor on any heuristic for estimating coherence. Furthermore, its insensitivity to the initial ordering of rays seems quite attractive, at least theoretically. It relies solely on what ultimately defines whether rays do the same operations–the actual scene geometry and acceleration structure. Note that this concept is valid not only for BVHs and triangles, but for any kind of hierarchical data structure and primitive type.

5 Experiments and Results

To evaluate the impact of our method, we compare it against plain packet traversal, and measure the SIMD efficiency of both methods—for a hypothetical N-wide SIMD architecture with scatter/gather support—in a variety of experiments. In particular, we compare the respective algorithms' SIMD utilization for three fundamentally different types of rays: primary rays, N-bounce specular rays, and one bounce diffuse rays. Primary rays should be considered a best-case example for packet tracing, while the N-bounce specular ray test is more like a worst case (as proposed by Reshetov et al. [18]). The one-bounce diffuse reflection allows us to consider rays more in the style of full distribution ray tracing [6].

Because we explicitly want to abstract from any underlying hardware specifics, we make several assumptions: First, we do not consider memory bandwidth or cache effects *at all*. This is a rather grave abstraction for any physical architecture our algorithm might eventually run on; however, not considering these effects at all arguably is not worse than simulating with an arbitrary memory subsystems and then using that to derive claims for other. Second, we assume that all operations are equally costly; this again is unrealistic, as, for example, a scatter/gather operation is likely to be more costly than, say, arithmetic operations. To abstract from this problem, we do not count individual operations at all, and instead only determine the average SIMD utilization for the two most abstract, high-level operations in ray tracing: a traversal step (i.e., a ray-box test), and triangle intersection (both in SIMD, of course). Even for these operations we only give the average utilization, not absolute numbers. However, as both algorithms perform exactly the same number of *active* operations, the ratio of utilizations is inversely proportional to the ratio of actual operations (in other words: if algorithm A has twice the average utilization of algorithm B, B will perform twice as many operations).

5.1 Experimental Setup

To be able to simulate different SIMD widths, we do *not* make use of the respective processor's SIMD hardware. Instead, we use a generic C++ *SIMD* class in which the SIMD width is as a template parameter. This allows for simulating features that are not supported in SSE, such as wider SIMD units and support for *scatter*, *gather*, and *compact* operations. We believe that all of these features will be extensively available in future hardware architectures (they already are, at least to some degree, in both a G80, and in SSE4) Both out new method and packet traversal are implemented using this generic SIMD class, and track the SIMD efficiency–i.e., the percentage of active rays–in each traversal and triangle intersection step. Being designed for flexibility and ease of generating simulation data, the implementation is rather slow. Consequently, we report only statistics like SIMD efficiency, but no absolute timings.

We will primarily compare out method to standard packet tracing in the spirit of "traditional coherent ray tracing"; both traversal schemes operate on a bounding volume hierarchy built using a surface area heuristic [22], and all examples are run at 1024×1024 pixels. Data will be reported for a set of typical ray tracing test scenes: ERW6, conference, blade, and fairy-forest (see Figure 1).

5.2 Comparison for Primary Rays

5.2.1 Efficiency of Plain Packet Tracing

To provide a baseline for comparisons, we first determine the SIMD efficiency of plain packet tracing for primary rays (Table 1). Since

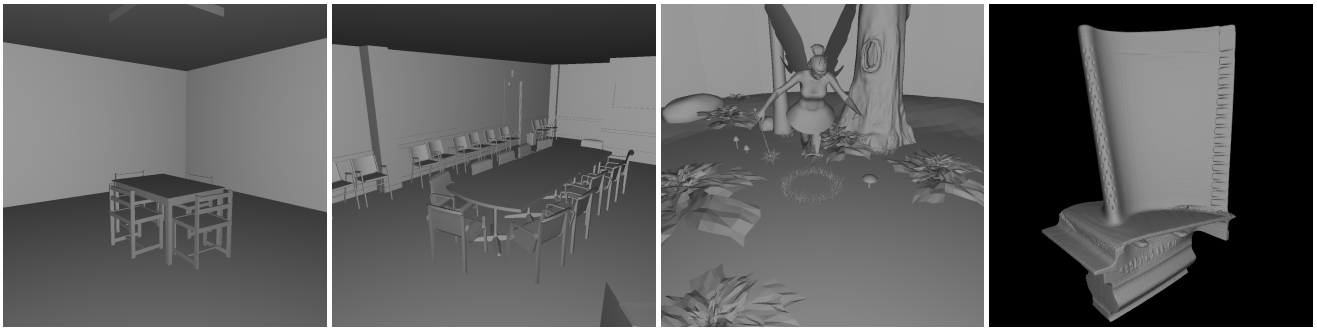

Figure 1: The four test scenes used for our experiments: erw6, 806 triangles; conference 282K triangles; fairy, 174K triangles (time-step 100); and blade, 1.76m triangles. Since we only generate simulated statistical data, all scenes are used in static configurations only. The erw6 and conference scenes are closed scenes, fairy-forest is mostly closed (the sky is open), and blade is a single object without environment.

primary rays are widely acknowledged as being highly coherent, this experiment can be considered a best-case experiment for both packet and SIMD stream tracing.

Intersection vs Traversal Efficiency. SIMD efficiency for triangle intersection is always lower than for BVH traversal, in particular for increasing model complexity. This is because triangle intersections are always performed at the leaves where the rays are most diverged, whereas for traversal at least some of the traversal steps operate close to the root, where rays are still coherent. This has been speculated about before (see, e.g. [21]), and our measurements empirically prove it to be the case.

Model complexity. Table 1 also shows the impact of rising model complexity on SIMD efficiency. While SIMD efficiency remains high for the ERW6 with it's huge triangles, it drops significantly for the conference and fairy scenes, and drops even further for the blade scene. Even disregarding the acknowledgedly extreme experiment of using 8×8 packets, the data for 4×4 packets indicate that for a SIMD width of 16, only an average of 9.1 out of 16 rays would be active during traversal, and only 2.1 out of 16 during triangle intersection (57% respectively 13%)–even for the best-case setting of high-density primary rays. Even for the smallest reasonable packet size of 2×2 rays, intersection SIMD efficiency drops to almost one third in the blade model, and even in the conference and fairy scenes, almost one ray in four is un-utilized.

Increasing SIMD width has an even more adverse effect on efficiency. Efficiency remains reasonable for the SSE-like 2×2 case, but drops considerably for the 4×4 case. This is particularly true for the blade model, but applies for all models.

In the 8×8 packet traversal case, intersection efficiency drops to less than 50% in all but the ERW6 scene, and in the blade scene, to a horrendous 3% (i.e., in this case, only 1.9 out of 64 units are doing productive work!). Note that while a simulated SIMD width of 64 may seem unlikely to enter the CPU domain any time soon, an NVidia G80 GPU already has a warp size of 32, and the 8×8 results are a good indicator on SIMD efficiency that could be expected from a plain packet traversal algorithm on such an architecture.

packet size	erw	conf	fairy	blade
2×2	99 / 97	97 / 87	96 / 83	83 / 39
4×4	97 / 93	93 / 69	88 / 61	57 / 13
8×8	94 / 84	84 / 48	75 / 35	30 / 3

Table 1: SIMD efficiency (for traversal/intersection, respectively; in percent, rounded down) for packet tracing using 2×2 packets for 4-wide SIMD, 4×4 packets for 16-wide SIMD, and 8×8-packets for 64-wide SIMD, respectively.

5.2.2 Efficiency of the SIMD Stream Traversal

The SIMD efficiency for our new algorithm–again with primary rays only so far–is given in Table 2. If the initial stream size matches the SIMD width, the stream traversal achieves exactly the same efficiency as the packet traversal. For larger initial stream sizes of 32×32 or 64×64 rays, the SIMD utilization increases significantly, in particular in cases where standard packet traversal suffers (i.e., for large SIMD width and/or high model complexity). This is due to the fact that even if certain rays in a SIMD ray-box test may miss the box, the compress operation will automatically remove these newly deactivated rays from the output stream, meaning the next stream-box test will again operate on a fully active stream of rays. Thus, there are only two sources of potential imbalance: first, due to shrinking during traversal the streams sizes will, in general, not be multiples of the SIMD width, implying that the last chunk of rays will be only partially filled; second, eventually streams *may* become shorter than the SIMD width, so SIMD utilization will necessarily dip below 100%.

5.3 N-Bounce Specular Rays

Though helpful to get a best-case estimate, primary rays are not the most representative set of rays to evaluate a ray tracing algorithm. In particular, primary rays are highly coherent, and algorithms that work well on primary rays may still break down for less coherent secondary rays. For that reason, Reshetov [18] proposed an effective yet easily reproducible benchmark for secondary rays based on computing N bounces of specular reflections for every ray (independent of the scene's material properties). In his experiments, SIMD efficiency for traditional packet tracing of a kd-tree dropped rapidly after only a few bounces.

In Table 3, we give the SIMD efficiency for both SIMD packet tracing and SIMD stream tracing for a varying number of specular bounces for both 4-wide and 16-wide simulated SIMD units. For packet tracing, we use a SIMD width matching the packet sizes

packet size	N	erw	conf	fairy	blade
2×2	4	99 / 97	97 / 88	96 / 83	83 / 39
32×32	4	99 / 99	99 / 98	99 / 97	96 / 61
64×64	4	99 / 99	99 / 99	99 / 97	96 / 62
4×4	16	97 / 93	93 / 70	88 / 61	57 / 13
32×32	16	99 / 99	98 / 93	97 / 85	80 / 18
64×64	16	99 / 99	99 / 95	98 / 87	82 / 18

Table 2: SIMD efficiency for the SIMD stream traversal for simulated 4-wide and 16-wide SIMD units, and for various initial stream sizes. While actually designed for initial stream sizes that are far larger than SIMD width, we also include data for a SIMD-width stream for reference purposes.

of 2×2 and 4×4; the stream tracing uses input streams of 32×32 rays in both cases.

Packet tracing. As expected, the data for reflection depth 0 matches the data for primary rays; increasing the number of bounces leads to a severe drop in efficiency, depending on the scene. For the erw6 scene, efficiency remains high even for 6 levels of reflections. For the conference and fairy scenes, SIMD efficiency drops to roughly one half for 2×2 packets and roughly one quarter for 4×4 packets. Also as expected, SIMD efficiency is still somewhat reasonable for the smaller 2×2 packets, but can drop significantly for 4×4 rays.

Stream tracing. With the stream tracing approach, the reordering works quite well. For the erw6 scene, the reordering maintains nearly full utilization even after multiple bounces, and even for the conference and fairy scenes utilization is significantly higher than for packet tracing. In particular, the traversal coherence is greatly improved for higher bounce depths in both the 4-wide and 16-wide SIMD cases. Obviously, the re-ordering cannot extract coherence if none is available. Eventually, SIMD efficiency drops even in the stream version due to a lack of rays requiring common operations. The important point, however, is that decreased utilization will occur at a much later stage than for packet tracing.

#refl./method		erw6	conf	fairy	blade
SIMD Width 4					
0	packet 2×2	99 / 97	97 / 87	96 / 83	83 / 39
0	stream 4 / 32	99 / 99	99 / 98	99 / 97	96 / 61
1	packet 2×2	98 / 92	94 / 81	90 / 74	76 / 37
1	stream 4 / 32	99 / 99	99 / 97	98 / 93	92 / 55
2	packet 2×2	97 / 91	91 / 76	85 / 67	70 / 35
2	stream 4 / 32	99 / 99	99 / 96	97 / 89	89 / 52
4	packet 2×2	95 / 89	82 / 64	76 / 59	67 / 35
4	stream 4 / 32	99 / 99	97 / 90	94 / 82	88 / 50
6	packet 2×2	94 / 87	74 / 55	71 / 55	67 / 34
6	stream 4 / 32	99 / 98	95 / 83	92 / 78	87 / 49
SIMD Width 16					
0	packet 4×4	97 / 93	93 / 69	88 / 61	57 / 13
0	stream 16/32	99 / 99	98 / 93	97 / 85	80 / 18
1	packet 4×4	95 / 84	86 / 58	75 / 46	46 / 12
1	stream 16/32	99 / 98	97 / 88	92 / 71	66 / 15
2	packet 4×4	92 / 82	77 / 49	63 / 36	39 / 11
2	stream 16/32	99 / 97	95 / 80	86 / 59	60 / 14
4	packet 4×4	88 / 77	60 / 34	48 / 27	35 / 10
4	stream 16/32	98 / 96	87 / 62	74 / 45	55 / 13
6	packet 4×4	83 / 72	46 / 25	40 / 23	34 / 10
6	stream 16/32	97 / 94	79 / 46	67 / 38	54 / 13

Table 3: *Comparison for n-bounce specular rays.* SIMD efficiency (for traversal/intersection) for both packet tracing and SIMD stream tracing, for 4-wide and 16-wide SIMD, using a forced number of n specular bounces. Packet tracing uses square packets of 2×2 resp. 4×4; SIMD stream tracing operates on initial ray sets of 32×32 pixels.

5.4 One-Bounce Diffuse Rays

Finally, we report data for a single diffuse bounce per ray, which is to simulate typical global illumination algorithms like final gathering, ambient occlusion, or path tracing. Most global illumination algorithms would arguably trace more coherent rays than one-bounce diffuse (e.g. highly specular materials or caustic), but one-bounce diffuse rays should nonetheless be roughly representative. Because real global illumination algorithms would arguably have a higher ratio of secondary to primary rays than the 1:1 ratio used in this experiment, we–for this particular experiment–use 16 samples per pixel, and include only the secondary rays in our measurements.

method	config	erw6	conf	fairy
SIMD Width 4				
packet	2×2	92 / 85	89 / 72	71 / 53
stream	$4/32 \times 32$	99 / 99	98 / 92	95 / 84
stream	$4/64 \times 64$	99 / 99	99 / 95	97 / 90
SIMD Width 16				
packet	4×4	86 / 70	69 / 42	40 / 21
stream	$16 / 32 \times 32$	99 / 96	91 / 70	79 / 49
stream	$16 / 64 \times 64$	99 / 98	95 / 80	88 / 63

Table 4: *Comparison for one-bounce diffuse ray distributions.* SIMD efficiency (traversal/intersection) of both packet tracing and SIMD stream tracing, for one-bounce diffuse rays (16 rays/pixel). Data is for the diffusely bounced rays only, primary rays are intentionally excluded.

As can be seen from Table 4, this experiment reinforces the results from the previous experiment: again, higher model complexity reduces the SIMD efficiency, as does increasing the SIMD width. Also as expected, for 16-wide SIMD, the *relative* impact of reordering is higher than for 4-wide SIMD ($> 2\times$ higher efficiency for the conference scene), but even re-ordering on 64×64 rays (16×16 pixels with 16 samples per pixel) still produces noticeable underutilization at 63% utilization in the fairy scene. Nevertheless, the reordering improves SIMD efficiency by more than $2\times$, and is generally much higher than expected for the seemingly random rays used in this experiment[2]

6 Discussion

We have demonstrated that for initial packet sizes larger than SIMD width, our stream tracing approach can yield significantly higher SIMD utilization in both BVH node traversal triangle intersection than standard packet tracing. However, several issues have to be discussed in order to see the whole picture.

6.1 Extendability

The framework discussed above is very general, and can be extended in many ways.

General secondary ray packets. Though only demonstrated for primary, N-bounce specular, and one-bounce diffuse rays, the algorithm can handle arbitrarily sized, unstructured soups of rays, and no explicit or implicit ordering is required.

Other data structures and primitives. Though presented only for BVHs and triangles, the algorithm extends easily to other hierarchical data structures and primitives.

Operations other than traversal and intersection. The same concept–repeated stream reduction–can also be applied to other operations like shading. In that case, one would successively generate streams that share the same shader, that access the same texture, etc. Though our shaders already do this for the shader, our scenes have no "real" shading and texture information, so we decided to intentionally not include any—arguably misleading—utilization data for shading. Most generally, the core concept could also be applied to other, completely non-ray tracing related techniques. For example, collision detection, k-nearest neighbor queries (photon map), database queries, etc, could use this method as long as these techniques use a hierarchical data structure and can be written such as

[2] In particular the efficiency for packet tracing is higher than expected. We believe that this is due to rays not only intersect geometry where they eventually end up (and where they are arguably most incoherent), but also where they originate, and where they are still coherent.

to perform its primitive operations in a (data-)parallel way. Eventually, we are re-addressing the question of how ray tracing maps to a stream processing framework, which was previously addressed, for example, by Purcell et al. [17]. However, Purcell investigated a particular hardware architecture (a third-generation, only partially programmable GPU) and was constrained by the particular hardware constraints of that architecture (e.g., no scatter write); instead, we investigate the question on a higher abstraction level, but in addition investigate the SIMD side of any such architecture[3]

Extendability to IA/frustum techniques. Our algorithm extends to frustum- or interval-arithmetic driven culling schemes [7, 19, 22, 23]. Given conservative bounds (e.g., interval data) for the input set of rays, in each traversal step one can first perform the conservative culling test. If that test is successful, one can directly proceed to the next traversal step without performing a stream reduction; if not, one performs the stream reduction step as described above, and could–though this is not absolutely required–then re-compute the bounding information for the reduced ray stream.

Suitability for stream architectures. Being motivated by a streaming compute paradigm, the algorithm would ideally fit stream-oriented hardware architectures such as Imagine [11] or the Stanford Streaming Supercomputer [9]. In such a framework, the scatter/gather operations would be performed by the memory controller. Eventually, the combination of streaming and SIMD processing for generalized ray packets would also be an interesting paradigm for designing special-purpose ray tracing hardware.

6.2 Relation to Existing Techniques

The proposed technique is a generalization of traditional ray traversal, packet tracing, and breadth-first traversal: for `PACKET_SIZE=1`, the algorithm specializes to standard recursive single-ray traversal; for `PACKET_SIZE>1` and `SIMD_WIDTH=1`, the algorithm specializes to standard breadth-first ray traversal as; and finally, for `PACKET_SIZE>1` and `SIMD_WIDTH=PACKET_SIZE`, the algorithm specializes to standard packet traversal as described in [24].

6.3 Limitations

Maximally extractable SIMD parallelism. Our algorithm can effectively extract the SIMD parallelism available in the initial ray set. However, if the initial ray set does not contain any such parallelism at all, SIMD efficiency will still be low. In particular, if triangles become sub-pixel size, then even a very large initial ray stream will end up with very short ray streams for each individual triangle (this effect is already visible in Table 2 for the blade data-set).

Eventually, this suggests a different SIMD processing scheme at least for the triangle intersection stage: for example, instead of only operating on streams of rays that request intersection with the *same* triangle, one could also combine streams from multiple different triangles; due to the scatter capabilities, each of the stream element could eventually operate on a different triangles (though merging the results would require additional care).

Similarly, low SIMD efficiencies suggest building shallower hierarchies. Instead of intersecting two small BVH leaves with 3 triangles each, intersecting a larger one with 6 triangles has the same intersection cost, but saves some traversal steps. This observation will require more consideration.

[3] In Purcell's target hardware, pixels were processed independently, even though all pixels associated with one of his "quads" could be considered an extremely wide SIMD vector.

Initial stream size. Our algorithm depends on a reasonably large number of rays in the initial ray set. If this is not the case, no efficiency gains through re-ordering can be expected. On the flipside, overly large initial ray sets may have a high theoretical SIMD efficiency, but might nonetheless perform badly on real machines due to cache spilling. As a result, practical implementations will likely enforce a maximum initial stream size, which may become problematic for shaders with a variable number of output rays.

Programming model. In particular for secondary rays, having a large enough initial ray set to start with implies that multiple different shaders pool their rays before tracing them together. It is a strength of our algorithm that this is supported at all, and that different shaders can pool their rays without any restrictions on number or ordering of these rays. Still, the programming model and API for writing such shaders may prove to be non-trivial.

7 Summary and Conclusions

In this paper, we have presented a new approach to SIMD ray tracing that yields higher SIMD efficiency than traditional packet tracing schemes. The approach supports both larger-than-four SIMD widths and general, arbitrary sized "packets" of arbitrary rays.

The technique operates on rather large streams of rays, and implicitly reorders the rays after each traversal step by discarding newly de-activated rays from each traversal step's output stream. This ensures that each new traversal step's input stream only contains active rays.

The key *strengths* of our algorithm are that

it can re-order on the fly, thus achieving higher SIMD utilization than packet tracing, and eventually extracts all of the SIMD parallelism available. In particular, the ordering in which the rays is passed is completely irrelevant.

re-ordering is done on the geometry hierarchy itself, and does not depend on hard-to-control and possibly faulty ray coherence heuristics

it supports arbitrary SIMD widths, and is thus applicable for a wide range of future hardware architectures, and

it is general, and not restricted to BVHs and triangles.

The key *challenge* with the presented approach is that so far it has only simulated results. Achieving more realized performance than packet tracing with existing hardware may not be trivial. In particular, our algorithm relies on efficient support for scatter and gather operations, which are not yet available in contemporary instruction sets like SSE. However, our algorithm is not designed for existing hardware, but for hardware architectures with much wider SIMD units than used today. Nevertheless, ultimately ray tracing on wide-SIMD architectures will remain a challenge: while our algorithm can significantly increase the SIMD efficiency compared to traditional packet tracing, for complex scenes and/or mostly incoherent rays there eventually isn't enough SIMD parallelism in the input set of rays that our algorithm could extract. Ultimately, this suggests a different data-parallel approach to SIMD ray tracing, in which a single operation traverses a set of rays through *different* BVH nodes respectively intersects them with *different* rays in the same operation. This technique has been explored by Pixar's PhotoRealistic Renderman, where ray coherence is assumed to minimal [8]. Existing techniques have not investigated this possibility mostly due to the restrictions of the SSE instruction set, which does not allow for gather reads (which is essential for operating on different, random triangles); if scatter and gather was supported in hardware—as it is, for example, on a G80—such an approach would be feasible.

Pixar's system avoids this by knowing that the geometry and BVH nodes are stored together in their geometry caches.

Future work. As a next step, we would like to build a real-time implementation of our algorithm. First results on a SSE-variant are available, and, albeit significant overhead in software-emulated scatter/gather, results are promising. In addition, we would like to perform more simulations for different kinds of ray distributions–in particular, global illumination rays–and start work on designing an appropriate shader programming model. Finally, we have already suggested a different data-parallel approach to SIMD ray tracing in which each operation operates on *different* triangles respectively BVH nodes; investigating this should be straightforward using the generic SIMD simulation tools we already have at hand.

REFERENCES

[1] *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 2006.

[2] Carsten Benthin. *Realtime Ray Tracing on current CPU Architectures*. PhD thesis, Saarland University, 2006.

[3] Carsten Benthin, Ingo Wald, Michael Scherbaum, and Heiko Friedrich. Ray Tracing on the CELL Processor. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* [1], pages 15–23.

[4] Carsten Benthin, Ingo Wald, and Philipp Slusallek. Interactive Ray Tracing of Free-Form Surfaces. In *Proceedings of Afrigraph*, pages 99–106, November 2004.

[5] G.E. Blelloch and J.J. Little. Parallel solutions to geometric problems in the scan model of computation. *Journal of Computer and System Sciences*, 48(1):90–115, 1994.

[6] Solomon Boulos, Dave Edwards, J Dylan Lacewell, Joe Kniss, Jan Kautz, Peter Shirley, and Ingo Wald. Packet-based Whitted and Distribution Ray Tracing. In *Proceedings of Graphics Interface 2007*, May 2007.

[7] Solomon Boulos, Ingo Wald, and Peter Shirley. Geometric and Arithmetic Culling Methods for Entire Ray Packets. Technical Report UUCS-06-010, SCI Institute, University of Utah, 2006.

[8] Per H. Christensen, Julian Fong, David M. Laur, and Dana Batali. Ray tracing for the movie 'Cars'. In *Proc. IEEE Symposium on Interactive Ray Tracing*, pages 1–6, 2006.

[9] Bill Dally, Pat Hanrahan, and Ron Fedkiw. Stanford streaming supercomputer whitepaper, 2001.

[10] GPGPU Forum. Stream compaction on G80. http://www.gpgpu.org/forums.

[11] Brucek Khailany, William J. Dally, Scott Rixner, Ujval J. Kapasi, Peter Mattson, Jinyung Namkoong, John D. Owens, Brian Towles, and Andrew Chang. Imagine: Media processing with streams. *IEEE Micro*, pages 35–46, March/April 2001.

[12] Aaron Knoll, Ingo Wald, Steven G Parker, and Charles D Hansen. Interactive Isosurface Ray Tracing of Large Octree Volumes. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 115–124, 2006.

[13] Christian Lauterbach, Sung-Eui Yoon, David Tuft, and Dinesh Manocha. RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* [1], pages 39–45.

[14] Jeffrey Mahovsky and Brian Wyvill. Memory-Conserving Bounding Volume Hierarchies with Coherent Raytracing. *Computer Graphics Forum*, 25(2), June 2006.

[15] Gerd Marmitt, Heiko Friedrich, Andreas Kleer, Ingo Wald, and Philipp Slusallek. Fast and Accurate Ray-Voxel Intersection Techniques for Iso-Surface Ray Tracing. In *Proceedings of Vision, Modeling, and Visualization (VMV)*, pages 429–435, 2004.

[16] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering Complex Scenes with Memory-Coherent Ray Tracing. *Computer Graphics*, 31(Annual Conference Series):101–108, August 1997.

[17] Timothy Purcell, Ian Buck, William Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, 2002. (Proceedings of ACM SIGGRAPH).

[18] Alexander Reshetov. Omnidirectional ray tracing traversal algorithm for kd-trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* [1], pages 57–60.

[19] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-Level Ray Tracing Algorithm. *ACM Transaction on Graphics*, 24(3):1176–1185, 2005. (Proceedings of ACM SIGGRAPH 2005).

[20] Jörg Schmittler, Ingo Wald, and Philipp Slusallek. SaarCOR – A Hardware Architecture for Ray Tracing. In *Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, pages 27–36, 2002.

[21] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University, 2004.

[22] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, 26(1):1–18, 2007.

[23] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G. Parker. Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics*, 25(3):485–493, 2006. (Proceedings of ACM SIGGRAPH).

[24] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3):153–164, 2001. (Proceedings of Eurographics).

[25] Sven Woop, Jörg Schmittler, and Philipp Slusallek. RPU: A programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics*, 24(3):434–444, 2005. (Proceedings of SIGGRAPH).