Coherent Ray Tracing via Stream Filtering

Christiaan P. Gribble
Department of Computer Science
Grove City College

Karthik Ramani
School of Computing
University of Utah

Figure 1: *Coherent ray tracing via stream filtering.* Stream filtering combines breadth-first ray traversal and elimination of inactive ray elements to exploit the coherence exhibited when processing arbitrarily-sized groups of rays in SIMD fashion. Given an appropriate hardware architecture, we show that interactive performance is achievable for a variety of ray tracing scenarios.

Abstract

We introduce an approach to coherent ray tracing based on a new stream filtering algorithm. This algorithm, which is motivated by breadth-first ray traversal and elimination of inactive ray elements, exploits the coherence exhibited by processing arbitrarily-sized groups of rays in SIMD fashion. These groups are processed by a series of filters that partition rays into active and inactive subsets throughout the various stages of the rendering process. We present results obtained with a detailed cycle-accurate simulation of a hardware architecture that supports wider-than-four SIMD processing and efficient scatter/gather memory and stream partitioning operations. In this context, stream filtering achieves frame rates of 15-25 fps for scenes of high geometric complexity rendered with path tracing and a variety of advanced visual effects.

Index Terms: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Ray tracing I.3.1 [Computer Graphics]: Hardware Architecture—Parallel processing

1 Introduction

Ray tracing is a well-known rendering algorithm, acclaimed for its ability to produce highly realistic and even predictive images. In the two decades since the introduction of ray tracing in its classic form [26], exponential growth in the available compute power and a variety of algorithmic developments have combined to realize real-time ray tracing on commodity processors. While the compute power of these processors continues to grow, the improved performance is no longer the result of simply increasing clock rates; rather, improved performance is realized by duplicating the computational elements that support previous CPU designs. In fact, many current multicore CPUs run at slower clock rates than previous single core processors, and future processors will likely rely on increased parallelism rather than improved sequential processing.

For an embarrassingly parallel algorithm such as ray tracing, effectively exploiting multiple cores is straightforward. However, on a finer scale, current CPUs also offer a way to exploit parallelism using Single Instruction Multiple Data (SIMD) processing. SIMD instruction sets provide the opportunity to exploit parallelism by executing the same instruction on multiple data items at the same time. Current CPUs expose a relatively small SIMD width, permitting manipulation of four single precision floating-point elements in parallel. The complexity of mapping an algorithm to a SIMD programming model increases with the width of the SIMD unit, so these narrow SIMD units attempt to balance the tradeoff between parallelism and software complexity. However, given the small cost and potentially high benefit of SIMD processing, it seems likely that future architectures will incorporate wider-than-four SIMD units.[1]

Coherent ray tracing [22], or packet-based ray tracing, enables the efficient use of SIMD processing in ray tracing. In this approach, rays are processed in coherent groups utilizing SIMD extensions such as x86 SSE or PowerPC Altivec instructions. While highly successful, packet-based ray tracing is efficient only for highly coherent packets in which all of the rays in a packet execute the same traversal, intersection, and shading operations. When rays begin to diverge, potentially large subsets of the rays do not actively participate in the computations. In these cases, packet-based ray tracing still performs N operations with every instruction (where N is the SIMD width), but only a smaller number of these operations ($n < N$) involve active rays. The remaining elements of the SIMD unit perform work that would not occur when tracing single rays.

This observation forms the basis for one metric that can be used to evaluate coherent ray tracing algorithms: we define *SIMD efficiency* to be the relative number of useful operations that are performed by a SIMD unit, $\frac{n}{N}$. In the worst case, only a single ray is active for a given sequence of operations, leading to an efficiency of $\frac{1}{N}$. In these situations, wide SIMD units lead to lower utilization and, thus, to lower performance. This problem is perhaps the primary reason that ray tracing may not employ wide SIMD processing: utilization is a key component in the design of such hardware.

Conventional wisdom dictates that secondary rays are much less coherent than primary rays, but the exact degree of coherence is difficult to predict. In fact, recent research [3, 10, 17] provides con-

[1] In this context, we refer to such SIMD units simply as *wide SIMD units*. While we recognize that many architectures support SIMD vectors with potentially thousands of elements, the width N implied by our use of this terminology is restricted to those with $N \in [8, 64]$ elements.

flicting results. Wide SIMD units thus present a dilemma: on the one hand, such units may provide the computational power necessary to achieve interactive rates for advanced visual effects. On the other, current coherent ray tracing algorithms running on these same units will likely perform poorly: first, because the impact of incoherent rays is more pronounced in wide SIMD environments; and second, because larger packets are more likely to contain incoherent rays than smaller packets.

We address these issues with a new algorithm for coherent ray tracing called *stream filtering*. This approach recasts the basic ray tracing algorithm as a series of filter operations that partition arbitrarily-sized groups of rays into active and inactive subsets to exploit coherence in a manner amenable to SIMD processing. An initial exploration of wide SIMD processing for ray tracing [24] has already demonstrated improved efficiency for traversal and intersection operations when compared to traditional packet-based ray tracing. In this work, we demonstrate that sufficiently long ray streams exist in each of the major stages of ray tracing (traversal, intersection, and shading) and show that SIMD utilization remains high enough to make the use of wider-than-four SIMD units a viable option for ray tracing.

We also explore the requirements for a hardware architecture to deliver interactive frame rates with stream filtering. We present results obtained with a cycle-accurate simulator that implements wide SIMD processing and efficient scatter/gather and stream partitioning operations, and demonstrate that interactive rates (15-25 fps) are achievable for a variety of scenes and visual effects, such as those depicted in Figure 1.

2 Related Work

Stream filtering integrates and extends several existing techniques in ray tracing, and we briefly review the relevant works below.

Coherent ray tracing. The use of ray packets to exploit SIMD processing was first introduced by Wald et al. [22]. The original implementation targets the x86 SSE extensions, which execute operations using a SIMD width of four, and consequently uses packets of four rays. Later implementations use larger packet sizes of 4×4 rays [1], but these fixed-size packets are neither split nor reordered.

Packet-based ray tracing was originally designed for kd-tree traversal and triangle intersection operations [22], but recent research has extended the algorithm to a wide variety of acceleration structures, primitive types, and hardware architectures. Packet-based ray tracing has also been exploited successfully in special-purpose ray tracing hardware projects [20, 27]. We generalize packet-based ray tracing to process arbitrarily-sized groups of rays efficiently in wide SIMD environments.

Packet-based optimizations. Processing large packets of coherent rays also permits a wide variety of algorithmic optimizations such as frustum culling [19], interval arithmetic [4, 23], and vertex culling [18]. These techniques exploit the same coherence employed by the basic coherent ray tracing algorithm—though often in scalar or narrow SIMD fashion—and so become problematic with wide SIMD units. Although these optimizations can be combined with stream filtering, we do not consider these techniques and instead focus only on SIMD ray tracing.

Secondary rays. Several recent works have investigated the problem of coherence in secondary rays. Boulos et al. [3] describe packet assembly techniques that achieve similar performance (in terms of rays/second) for distribution ray tracing as for standard recursive ray tracing. Similarly, Mansson et al. [10] describe several coherence metrics for ray reordering to achieve interactive performance with secondary rays. In contrast, Reshetov [17] has shown that even for narrow SIMD units, perfectly specular reflection rays undergoing multiple bounces quickly lead to almost completely incoherent ray packets and $\frac{1}{N}$ SIMD efficiency. Thus, worst-case SIMD efficiency is not only a theoretical possibility, but has been demonstrated in current packet-based ray tracing algorithms. We show that stream filtering maintains high efficiency when processing seemingly incoherent groups of rays, including the secondary rays required for a number of important visual effects.

Ray scheduling. Improved coherence can also be achieved by considering ray traversal as a scheduling problem. For example, early work by Pharr et al. [14] explores this idea to reduce disk operations when rendering complex scenes that are larger than main memory, but coherence at finer scales is not considered. Recently, Navratil et al. [13] have shown that ray scheduling can be used to improve cache utilization and reduce the necessary DRAM-to-cache bandwidth. As noted, we focus only on improving SIMD utilization via stream filtering, but the approach does not preclude the use of scheduling techniques to further improve performance.

Breadth-first ray tracing. Instead of tracing rays in a depth-first manner, several works have investigated breadth-first ray traversal. Nakamaru and Ohno [12] describe one such algorithm designed to minimize accesses to scene data and maximize the number of rays processed at a time. Mahovsky and Wyvill [9] have explored breadth-first traversal of bounding volume hierarchies (BVHs) to render complex models with progressively compressed BVHs. This approach, however, uses breadth-first traversal to amortize decompression cost and does not target either interactive performance or SIMD processing. Stream filtering builds on these ideas to extract maximum coherence in arbitrarily-sized groups of rays.

3 Stream Filtering

The stream filtering approach recasts the basic ray tracing algorithm as a series of filter operations that exploit coherence by partitioning arbitrarily-sized groups of rays into active and inactive subsets. In this section, we elucidate two core concepts in this approach: *streams* of rays, and sets of *filters* that extract substreams with certain properties. We also describe the application of this approach to the major stages of ray tracing and provide details of a simulated hardware architecture that supports N-wide SIMD processing and efficient scatter/gather memory and stream partitioning operations.

3.1 Computational Framework

A stream contains data of the same type and can be of arbitrary length. Stream elements are independent of each other and can thus be processed in an arbitrary order. *Ray streams* are defined to be arbitrarily-sized groups of rays, and a *stream filter* is a set of conditional statements executed across such a stream:

```
out_stream filter<test>(in_stream)
{
  foreach e in in_stream
    if (test(e) == true)
      out_stream.push(e)
  return out_stream
}
```

We observe that the core operations in ray tracing, including traversal, intersection, and shading, can be written as a sequence of conditional statements that are applied to each ray. With stream filtering, instead of applying conditional statements to individual rays, the statements are executed in SIMD fashion across groups of N rays to isolate those rays exhibiting some property of interest. An important purpose of this work is to demonstrate that streams of sufficient length exist for traversal, intersection, and shading to maintain high utilization with wide SIMD units.

SIMD processing. In an N-wide SIMD environment, filters are implemented as a two-step process: conditional statements are first applied to groups of N elements from the input stream, generating a Boolean mask. To create the output stream, the input stream is then partitioned into active and inactive subsets based on these results. This process is depicted in Figure 2.

Non-sequential memory access patterns require scatter/gather operations to generate a sequential stream of ray data from the stream elements. Thus, one important requirement for the stream

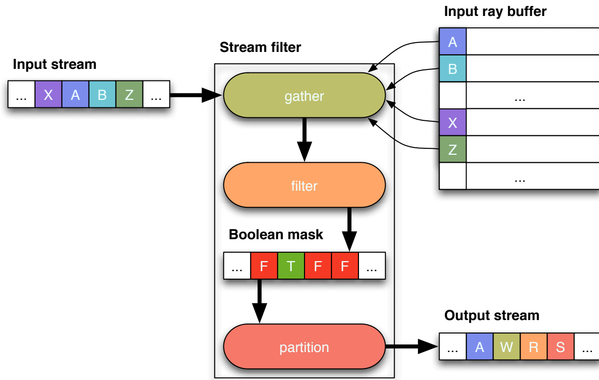

Figure 2: *Stream filtering in ray tracing.* Core operations are written as a sequence of conditional statements that are applied to arbitrarily-sized groups of rays, isolating those rays that exhibit some property of interest. In this way, inactive elements are removed from the stream, yielding higher SIMD utilization.

filtering approach is hardware support for these operations. Hardware support for partitioning operations is also desirable, but efficient algorithms for stream reduction are available in the literature [2, 7]. A second purpose of this work is to explore the hardware requirements necessary to achieve interactive frame rates with stream filtering in a wide SIMD environment.

Coherence. With stream filtering, wide SIMD units can be used to process arbitrarily-sized groups of rays with high efficiency for two reasons: first, the algorithm exploits parallelism when processing streams as a sequence of groups with N elements; second, stream filtering removes any elements that would perform unnecessary work in subsequent stages of the rendering process.

Stream filtering eliminates inactive rays on-the-fly during rendering. In fact, the output stream created by stream filtering is optimal with respect to the input stream: all rays from the stream that would perform the same sequence of operations will always perform those operations together. Note that this observation is true regardless of the order in which rays occur in the input stream or the sequence of operations that these rays undergo to reach the common operations. Thus, given the same input rays, no existing algorithm will be able to combine more operations of the same kind: stream filtering is optimal with respect to this definition of coherence.

Likewise, stream filtering requires neither potentially costly presorting operations nor heuristics to estimate coherence. Instead, coherence is defined solely by what ultimately determines the operations to which any particular ray is subjected—the scene geometry, the acceleration structure, the material shaders, and so forth. Note also that this observation is valid for any hierarchical acceleration structure, primitive type, or material model.

Memory management. To this point, the details of memory management have been abstracted to assume a memory system with a *push* operation that will place an element in the correct position within an output stream at any time. In practice, stream elements are managed in-place, using the memory that is currently allocated to the input stream. The elements in each output stream constitute a subset of the input, and these elements can be reordered with a *partition* operation. In our implementation, elements that pass the filter are moved to the start of the current input stream, and those that fail are moved to the end. Output streams are then determined by pointers to the start and end of the appropriate partition.

3.2 Application to Ray Tracing

Armed with an understanding of this computational framework, applying the approach to ray tracing becomes straightforward.

Traversal. For traversal, the input stream is recursively traced through a hierarchical acceleration structure such as a BVH. In each

```
traverse(node, in_stream)
{
  BoxTest node_test(node);
  out_stream = filter<node_test>(in_stream)
  if (empty(out_stream))
    return

  if (is_leaf(node))
    intersect(primitives, out_stream)
  else
    traverse(front_child(node), out_stream)
    traverse(back_child(node),  out_stream)
}
```

Figure 3: *Traversal in a BVH with stream filtering.* In each traversal step, inactive rays are filtered from the stream before it is forwarded to subsequent operations with the relevant BVH nodes.

traversal step, the stream is tested against the bounding box of the current node, and a stream filter partitions the input stream so that only those rays intersecting the node are included in subsequent traversal operations. If the output stream is empty, the next node is popped from a traversal stack and the process continues with that node. However, if the output stream contains active rays, the output stream is either intersected with the geometry in a leaf node, or the stream is recursively traversed through child nodes in a front-to-back order. As shown by the pseudocode in Figure 3, BVH traversal can be written very compactly with stream filtering.

Intersection. In their simplest form, stream filters for primitive intersection process an input stream by performing ray/primitive intersection tests in N-wide SIMD fashion. This process generates an N-wide Boolean mask indicating which rays intersect the primitive, and the mask is then used to store intersection information in the ray buffer with conditional scatter operations.

However, rather than perform primitive intersection operations with groups of N elements, the intersection test could instead be decomposed into a sequence of stream filters, or *filter stack*, for the relevant substages. This approach will potentially yield higher efficiency than simply performing the complete intersection test in SIMD fashion. Using a filter stack, each test is applied in succession with only those rays that have passed previous filters, thereby increasing SIMD utilization for a particular input stream. However, as shown in Section 4, ray streams processed during intersection are typically too short to warrant additional filtering operations, particularly for highly complex models, so our implementation does not employ filter stacks for primitive intersection and instead relies on the simpler approach described above.

Shading. Similarly, material shaders could process an input stream by simply performing operations in N-wide SIMD fashion. However, to maintain higher SIMD efficiency, filter stacks are used to extract rays requiring the same operations from the input stream. For example, Figure 4 depicts the complete filter stack for a Lambertian material shader from a path tracer. In the stack, input rays are processed by stream filters that extract shadow rays, rays that do not intersect geometry, and rays intersecting a light source. The remaining rays are then processed by the shader, which adds secondary rays as necessary. Additional filtering operations can be applied within each shader to group similar operations; for example, the Lambertian shader probabilistically samples either direct or indirect illumination, and the corresponding ray data are extracted using additional stream filters so that the required operations can be performed together.

3.3 Implementation

To understand the hardware requirements necessary to achieve interactive performance with stream filtering, we have implemented a cycle-accurate simulator similar to the SimpleScalar tool set [5].

Simulator components. The simulator is composed of four major subsystems that closely model an actual hardware implementation:

- A distributed memory consisting of two buffers (512 KB each) for current and next-generation rays, and a dual ported scratch pad for storing intermediate results.

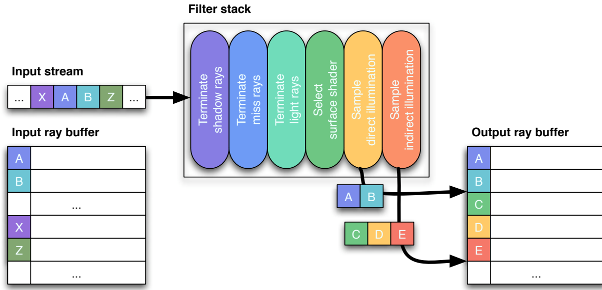

Figure 4: *Filter stacks for improved efficiency.* Within each stage of the rendering process, stream filters can be applied to avoid operations with inactive rays. Here, several filters are applied in a Lambertian material shader that determines the next generation of rays.

- An execution subsystem supporting N-wide SIMD operations and in-place stream partitioning.
- Address generation units that perform address fetch and drive integer functional units [16] for address computations and data alignment.
- A hierarchical interconnect composed of program-controlled multiplexers and pipelined registers [16] that orchestrates data movement between subsystems and sustains the bandwidth requirements of the address fetch and execution units.

Stalls resulting from data alignment operations are modeled accurately, as is contention on the interconnects and for functional units and the various memory hierarchy resources.

The multiplexers are carefully sized to allow for single-cycle operation at a frequency of 1 GHz. Likewise, the address fetch and execution units are guaranteed to be synthesized at 1 GHz with 130 nm technology [11]. However, clocking the flexible interconnect at the same frequency would require an aggressive design. We thus assume a conservative frequency of 500 MHz at 90 nm to accommodate the flexible interconnect and partitioning units.

Memory subsystem. Buffers for ray data consume 512 KB for 64×64 streams, which was empirically found to deliver the best balance between area and performance. A smaller buffer (256 KB) imposes a 30% performance degradation and requires a higher frequency to sustain execution parallelism. Increasing the size beyond 512 KB provides a marginal improvement in the hit rate at the cost of increased complexity, so the simulator employs 512 KB buffers.

To efficiently support a large number of SIMD units, these buffers are banked into 32 16 KB buffers. Banking is an efficient alternative to multi-ported buffers, which are expensive in terms of both area and power. Provided that requests do not collide frequently, this design sustains 32 different SIMD data fetch operations at the same time and incurs lower latency because banks are only 16 KB. Each bank is also supported by an address generator unit that computes the addresses for data fetch operations.

Data access requires two cycles, but the operations are pipelined. The overhead of pipelining is modeled accurately, as is the flexible interconnect subsystem that delivers data to and from the execution and address fetch subsystems. Execution stalls arising from SIMD data alignment are also taken into account during simulation.

Execution subsystem. One of the salient features of the architecture is that all address computations are isolated from the actual data computations. In addition to reducing data movement, this approach reduces contention for the flexible interconnect. Both computations occur in parallel and thus provide significantly higher performance. The cycle-accurate simulation models overheads accurately, and a minimal degradation in SIMD efficiency demonstrates that the architecture approaches the performance of ideal isolation between address and data computations.

Stream partitioning is supported by SIMD comparator units that determine which elements pass the filter, and the resulting mask is employed to form the active and inactive partitions.

Stream filter `apply` function template

```
RayStream StreamFilter::apply(RayStream& stream) const
{
  bool  out_mask[MAX_STREAM_LEN] = {};
  bool* mask_end = out_mask;

  int* in_begin  = stream.begin;
  int* in_end    = stream.end;
  int* out_begin = in_begin;
  int* out_end   = out_begin;

  for (int* block = in_begin; block < in_end; block += SIMD_WIDTH)
  {
    const int     nremain = (in_end - block);
    const simd_it duplicate(block[nremain-1]);
    const simd_bt active  = (get_ids<SIMD_WIDTH>() < nremain);
    const simd_it ids     = ifthen(active, (simd_it)block, duplicate);
    const simd_bt mask    = active && test(stream, ids);

    if (anytrue(mask)) store(mask_end, mask);
    mask_end += SIMD_WIDTH;
  }

  out_end += partition(stream.begin, stream.size(), out_mask);
  return RayStream(stream, out_begin, out_end);
}
```

Ray/box filter test

```
simd_bt BoxTest::operator()(const RayStream& stream,
                             const simd_it& ids) const
{
  const simd_ft org_x = gather(stream.in->org_x, ids);
  const simd_ft org_y = gather(stream.in->org_y, ids);
  const simd_ft org_z = gather(stream.in->org_z, ids);
  const simd_ft inv_x = gather(stream.in->inv_x, ids);
  const simd_ft inv_y = gather(stream.in->inv_y, ids);
  const simd_ft inv_z = gather(stream.in->inv_z, ids);
  const simd_ft min_t = zero;
  const simd_ft max_t = gather(stream.in->t_min, ids);
  return box.intersect(org_x, org_y, org_z, inv_x, inv_y, inv_z,
                       min_t, max_t);
}
```

Figure 5: *Programming model for stream filtering.* Programmable stream filters export an interface to generate output streams. Filter tests perform the necessary operations and return a mask indicating whether or not individual rays pass the test.

Programming model. Using the framework provided by this simulator, programmable stream filters are implemented as C++ class templates, and export an interface to generate an output stream corresponding to the active partition (Figure 5). Ray streams are processed in parallel by N-wide SIMD units and are then partitioned into active and inactive subsets before subsequent processing. The `partition` method employs a comparison sort to move active elements to the start of the stream, and the resulting output stream includes only those elements that pass the corresponding test.

Filter tests are implemented as C++ functors and serve as the template parameter to stream filter objects. Typically, these tests utilize gather operations to process rays in N-wide SIMD units and return a mask indicating the result for each element (Figure 5). Filter tests that modify rendering state use conditional scatter operations to update ray data.

4 Results

To evaluate the potential role of stream filtering in interactive ray tracing, we measure SIMD utilization and predict rendering performance for the test scenes depicted in Figure 1 using the cycle-accurate simulator described above.

Rendering details. Images are generated using a Monte Carlo path tracer compiled for the simulated N-wide SIMD architecture. Currently, the renderer supports three different material models: a coupled model for glossy reflections, dielectrics, and simple Lambertian surfaces. The rendered also uses a thin-lens camera model to simulate depth-of-field effects.

More importantly, the renderer implements the computational paradigm described in Section 3: traversal operations use a BVH constructed with a surface area heuristic [23], ray/primitive intersection tests are processed in N-wide SIMD fashion, and the material shaders use filter stacks to maximize coherence. Thus, throughout all stages of the rendering process, rays that require the same sequence of operations always perform those operations together.

To render a frame, the image is divided into tiles of pixels, and the input ray buffer is populated with M primary rays. Typically,

scene	# prims	# lights	memory geometry	memory textures	material shaders coupled	material shaders dielectric	material shaders lambert	per-frame stats # rays	per-frame stats # trav ops	per-frame stats # isec ops
rtrt	83845	2	8.32 MB	5.84 MB	•	•	•	1.18×10^8	9.77×10^6	1.26×10^6
conf	282644	72	20.62 MB	4.11 KB	•		•	1.62×10^8	3.25×10^8	6.51×10^7
kala	2124001	2	210.66 MB	404.42 MB			•	1.57×10^8	7.63×10^8	9.01×10^7

Table 1: *Characteristics of the test scenes.* Scenes of varying geometric complexity are used to evaluate the potential role of stream filtering in interactive ray tracing. These scenes employ three different material shaders to capture a variety of visual effects, including diffuse interreflection.

tiles of $\sqrt{M} \times \sqrt{M}$ pixels are used, with $M = 1024$ or $M = 4096$. These values were empirically determined to balance memory requirements for the stream against the coherence it exhibits. In fact, the memory requirements for processing such large streams are actually quite low; in our current implementation, the initial streams of ray identifiers require just over 4 KB (32×32 rays) and 16 KB (64×64 rays), and the buffers that store actual ray data require just over 96 KB (32×32 rays) and 384 KB (64×64 rays). Moreover, ray streams are partitioned in-place, so only two buffers for ray data are required.

Ray streams are traced in a breadth-first manner: primary rays are traced to completion, populating an output buffer with secondary rays as necessary. Pointers to the input and output buffers are swapped, and each subsequent generation of rays is traced in a similar manner. This process continues until the input stream contains no elements. Rendering proceeds by processing image tiles in this fashion until the image is complete.

Experimental setup. In the experiments that follow, data is gathered using the path tracer described above for images that are 1024×1024 pixels in resolution. The results correspond to the scenes and viewpoints depicted in Figure 1, and other relevant characteristics are given in Table 1. As can be seen, these scenes cover a wide range of geometric and illumination complexity, and are representative of common interactive ray tracing scenarios.

4.1 SIMD Utilization

As described in Section 3, stream filtering is applied at various points throughout the rendering process, with the intention of extracting maximum coherence within arbitrarily-sized groups of rays. The combination of stream filtering and partitioning operations extracts rays exhibiting some property of interest and removes inactive rays before performing subsequent operations. This approach requires input streams that are long enough to maintain high efficiency. If such streams exist, then stream filtering becomes a viable parallel processing paradigm for interactive ray tracing in wide SIMD environments.

Primary rays. We first determine utilization for primary rays with a variety of initial stream sizes not less than the SIMD width. Primary rays are widely acknowledged to be highly coherent, so this experiment represents a best-case scenario for any coherent ray tracing method. Table 2 reports the SIMD utilization for the traversal and intersection (T / I) stages of rendering using a single sample per pixel. Data for secondary rays are not included in these results.

As can be seen, utilization is quite high, achieving nearly 100% utilization during traversal, even with extremely wide SIMD units ($N = 64$). In addition, increasing the size of the initial stream improves utilization for all three scenes. If the initial stream size matches the SIMD width, stream traversal and intersection achieve exactly the same efficiency as in packet-based ray tracing. This result is expected, as stream filtering generalizes the packet-based ray tracing algorithm (see Section 5). For larger initial streams, however, utilization increases significantly because inactive rays are automatically removed from the output stream, and subsequent operations process only active rays. Thus, there are only two possible sources of underutilization: first, input streams will not, in general, be multiples of the SIMD width, so the last SIMD unit may be only partially filled; and second, if insufficient coherence exists within a stream, substreams may become shorter than the SIMD width. Utilization necessarily drops below 100% in these cases.

size	rtrt	conf	kala
	SIMD width $N = 4$		
2×2	95 / 84	94 / 79	92 / 70
32×32	97 / 89	95 / 80	95 / 90
64×64	97 / 87	95 / 89	94 / 92
	SIMD width $N = 16$		
4×4	90 / 70	91 / 64	87 / 55
32×32	94 / 80	97 / 87	96 / 80
64×64	93 / 79	96 / 88	96 / 83
	SIMD width $N = 64$		
8×8	80 / 43	82 / 41	80 / 33
32×32	89 / 50	91 / 72	89 / 55
64×64	90 / 52	90 / 74	91 / 57

Table 2: *SIMD utilization (T / I) for primary rays.* Even with highly complex scenes and extremely wide SIMD units, stream filtering maintains high SIMD utilization for primary rays.

size	rtrt	conf	kala
	SIMD width $N = 8$		
32×32	70 / 47 / 86	57 / 23 / 90	56 / 26 / 96
64×64	77 / 55 / 93	70 / 31 / 95	67 / 31 / 95
	SIMD width $N = 12$		
32×32	60 / 38 / 82	45 / 17 / 89	45 / 19 / 92
64×64	70 / 46 / 92	61 / 24 / 96	56 / 25 / 97
	SIMD width $N = 16$		
32×32	52 / 34 / 81	38 / 13 / 87	38 / 14 / 91
64×64	65 / 42 / 89	54 / 18 / 94	49 / 20 / 96

Table 3: *SIMD utilization (T / I / S) for secondary rays.* When compared to the ideal case, degradation in efficiency ranges from 2%-21% due to overheads that arise from address fetch, alignment, and partition operations.

Secondary rays. Though these results are promising, algorithms that work well with primary rays have been shown to perform poorly with secondary rays [17]. We thus measure utilization for general secondary rays using the test scenes described above.

Table 3 reports the SIMD utilization for traversal, intersection, and shading (T / I / S) in the path tracer for secondary rays. Although global illumination renderers would arguably attempt to generate more coherent rays than those produced by path tracing, this algorithm offers an opportunity to stress the stream filtering approach. Nevertheless, we use 64 samples per pixel in these experiments to better approximate rays that might be generated in practice. Data for primary rays are not included in these results.

As can be seen, utilization remains reasonably high under a variety of SIMD widths, with larger initial ray streams leading to higher utilization in all stages and for all scenes. Likewise, Table 4 shows that larger initial ray streams also lead to longer input streams for each stage of the rendering process.

Compared to the ideal case, utilization degrades by 5%-10% for traversal, 10%-21% for intersection, and 2%-5% for shading due to the overheads arising from address fetch, alignment, and data partitioning operations in this architecture. The address fetch overhead arises due to the time involved in calculating the addresses of the

scene	size	trav	isec	shade
rtrt	32×32	17.0	8.5	56.6
	64×64	29.1	12.2	147.7
conf	32×32	9.3	2.7	64.9
	64×64	17.4	3.9	242.1
kala	32×32	9.6	3.0	133.9
	64×64	14.9	4.3	535.5

Table 4: *Average stream length for secondary rays.* In general, input streams are sufficiently long to make wide SIMD environments an attractive alternative for ray tracing.

current stream elements and in populating the buffer with ray identifiers. Ideally, all address computations occur in parallel with the SIMD operations; however, in practice, a small percentage of the operations introduces stalls because of the time required to compute and fetch addresses. For SIMD widths of eight and 12, alignment overhead arises due to the larger line size of the ray buffers; it can be inferred from the data that this overhead is minimal for a SIMD width of 16. Finally, stream partitioning requires an in-place comparison sort and incurs additional computation and data movement operations before subsequent processing.

In this data, we see the potential downfall of wide SIMD environments for ray tracing: highly complex scenes with many small triangles lead to lower utilization during traversal and intersection. Intersection suffers the greatest reduction in stream length, which sometimes falls below the threshold of even narrow SIMD units. In contrast, material shaders typically process the longest streams, and filter stacks are used to good effect in this stage, resulting in high utilization during shading computations.

Table 5 shows the distribution of major operations for initial streams of 64×64 elements and a SIMD width of 16. In this data, the integer operations required by address fetch are subsumed by the load and store operations. As can be seen, actual data computations account for as much as 31%-35% of the total. While varying SIMD widths change the absolute number of operations for a given frame, the ratios are preserved.

4.2 Predicted Rendering Performance

These data necessitate an important observation: stream filtering successfully extracts any coherence exhibited by the rays in a particular stream, but if streams do not inherently possess such coherence, utilization will remain low. This observation places an upper bound on the width of the SIMD units that will be useful, although factors such as the number and cost of key operations, as well as the types and frequencies of potential hazards, must be considered to accurately predict rendering performance. As discussed in Section 3, the simulator closely models an actual hardware implementation, so we evaluate predicted rendering performance using the earlier test scenes.

In these experiments, results are gathered using 64 samples per pixel with initial stream sizes of 32×32 and 64×64 rays under various SIMD widths. We note that diffuse interreflection is approximated using a constant ambient term when rendering the *rtrt* scene, but is sampled to a maximum depth of three bounces for the other scenes. Additional per-frame statistics for each scene, including the total number of rays traced, are given in Table 1.

Assuming no stalls or dependencies of any kind, the theoretical maximum achievable frame rate is 100 fps for *rtrt*, 64 fps for *conference*, and 32 fps for *kalabsha* at an operating frequency of 500 MHz. As noted, the simulator accurately models the overheads present in an actual hardware implementation of the architecture, and Table 6 shows the resulting frame rates for each scene.

As can be seen, frame rates increase with the SIMD width, due to reductions in the overall alignment and partitioning overhead. On the other hand, wider SIMD units require more time for address computation and necessitate higher address fetch overhead. Thus, there exists a tradeoff between SIMD width and the overheads that

scene	load	store	comp	scat/gath	part
rtrt	24.1	14.9	35.4	19.6	5.0
conf	23.2	19.7	34.5	18.7	3.8
kala	23.8	20.3	31.9	21.7	2.0

Table 5: *Distribution of major operations as % of total.* Here, the compute-related operations refer to those involving actual ray data; integer operations are subsumed by the load and store operations.

penalize ideal performance. In particular, for *rtrt*, a SIMD width of eight balances the overheads sufficiently, and performance exceeds the 10 fps threshold. However, in moving from 8-wide to 12-wide SIMD units, significant improvements are observed for all three scenes, resulting from the reduction in overheads due to SIMD alignment and stream partitioning. Beyond a width of 12, the overhead of address computation begins to dominate, and improvements diminish accordingly. Thus, a SIMD width of about 12 elements balances the various overheads in this architecture.

These results demonstrate that, given an appropriate hardware architecture, stream filtering achieves interactive frame rates for complex scenes using path tracing and visual effects such as glossy reflections, dielectric materials, diffuse interreflection, and depth-of-field. As processors continue to rely on increasing levels of fine-grained parallelism, we believe that hardware support for wider-than-four SIMD processing and non-sequential memory access will become commonplace. With these architectures, then, stream filtering becomes a viable alternative for interactive ray tracing.

5 Discussion

We have demonstrated that for sufficiently large input streams, stream filtering maintains high utilization in wide SIMD environments throughout all stages of rendering. We have also demonstrated that, given an appropriate hardware architecture, stream filtering can achieve interactive performance for scenes of varying complexity rendered with advanced visual effects. We now highlight several key features and potential limitations of the algorithm.

5.1 Extensibility

The stream filtering framework discussed in Section 3 is very general and can be extended in many ways.

Material shaders. We have examined stream filtering with a renderer based on Monte Carlo path tracing that supports three material shaders, but the algorithm can handle any collection of rays that may be generated by material shaders other than those explored here. Stream filtering can thus accommodate a rich assortment of material shader commonly used in ray tracing applications.

Other acceleration structures and primitives. We have demonstrated stream filtering using a BVH to subdivide scenes composed of triangles and spheres, but other hierarchical acceleration structures and primitives types can be integrated quite easily. Although stream filtering does not as obviously apply to iterative traversal

size	rtrt	conf	kala
	SIMD width $N = 8$		
32×32	16.60 fps	8.15 fps	6.73 fps
64×64	18.78 fps	12.78 fps	8.34 fps
	SIMD width $N = 12$		
32×32	21.82 fps	12.56 fps	11.78 fps
64×64	24.52 fps	18.32 fps	13.45 fps
	SIMD width $N = 16$		
32×32	22.36 fps	14.35 fps	13.34 fps
64×64	26.35 fps	20.32 fps	15.65 fps

Table 6: *Rendering performance.* Stream filtering delivers interactive performance with the test scenes, rendered with path tracing and advanced visual effects such as diffuse interreflection.

method	stream size	SIMD width
Recursive ray tracing	$size = 1$	$width = 1$
Breadth-first ray tracing	$size > 1$	$width = 1$
Packet-based ray tracing	$size > 1$	$width = size$

Table 7: *Stream filtering as a generalization of existing techniques.* With appropriate values for key parameters, stream filtering specializes to several common ray tracing algorithms.

schemes in structures such as multilevel grids, an extension to frustum-based grid traversal [25] may be possible. Stream filtering can thus be combined with techniques used for ray packets, again accommodating many different ray tracing applications.

Operations besides traversal, intersection, and shading. The computational paradigm introduced in Section 3 can also be applied to other operations not strictly related to ray tracing, provided these operations employ a hierarchical data structure and can be written in SIMD fashion. Thus, stream filtering can potentially be used with a wide range of other rendering algorithms as well.

IA and frustum techniques. Interval arithmetic or frustum-driven culling schemes can be integrated with stream filtering in a straightforward manner. Given conservative bounds for each stream, appropriate culling tests can be performed before the necessary traversal and intersection operations. Combining stream filtering with these techniques may improve performance beyond what can be expected by processing rays with wide SIMD units alone.

Non-traditional hardware architectures. Stream filtering implements a streaming computational paradigm, so the algorithm is ideally suited for hardware architectures such as Imagine [8] and Merrimac [6], or perhaps Intel's upcoming Larrabee processor [21]. The results obtained with the cycle-accurate simulator suggest that the combination of streaming filtering and wide SIMD processing for generalized ray packets is also an intriguing design for special-purpose ray tracing hardware.

5.2 Relation to Existing Techniques

As noted in Section 2, stream filtering integrates and extends several techniques in ray tracing. However, as shown in Table 7, stream filtering actually generalizes many of these techniques. In particular, with appropriate values for stream length and SIMD width, stream filtering specializes to standard recursive ray tracing, to standard breadth-first ray traversal, or to packet-based ray tracing. A hardware architecture supporting stream filtering can thus be used to implement other ray tracing algorithms.

In certain respects, this work addresses many of the problems associated with mapping ray tracing to streaming architectures. Although some of these issues have been addressed previously (for example, by Purcell et al. [15]), these investigations rely on particular hardware architectures and are thus constrained by the sometimes limited programming models. Using the cycle-accurate simulator, we have investigated many of the same issues, although at a higher level of abstraction and explicitly from the perspective of efficient processing in wide SIMD environments.

Moreover, we have examined the hardware requirements necessary to realize interactive performance using stream filtering. The simulated architecture demonstrates the potential benefits of designing a custom core for various operations required by the algorithm. The salient features include nearly ideal isolation of address, data, and partitioning computations to reduce data movement and contention, thereby delivering high performance; and use of efficient address generation mechanisms to populate the ray identifier buffers that facilitate wide SIMD operations.

While the current design delivers interactive frame rates for a variety of scenes and visual effects, stream filtering opens a vast design space that may lead to interesting implementation alternatives. For example, multicore design choices include heterogeneous and homogeneous cores. In the former, each core might be responsible for one particular stage, communicating with other cores via a dual-buffered output memory, which permits data to be simultaneously read by the next core in the pipeline.

In a homogeneous multicore system, the single core design discussed in Section 3 can be replicated to provide additional, though coarser, levels of parallelism. In addition to low design complexity and core scalability, this design requires only minimal changes in the execution subsystem. For highly interactive frame rates, we believe that core complexity will be significantly higher in a heterogeneous system, so the transition to a multicore design with core replication is likely the preferred option.

5.3 Limitations

Results demonstrate that stream filtering is a viable option for interactive ray tracing, but the algorithm is not without its limitations.

Inherent parallelism. Stream filtering effectively extracts any parallelism inherent to an input stream. However, if the stream does not exhibit such parallelism, SIMD utilization will remain low. For example, if primitive size drops below the subpixel threshold, then even a very large input stream will likely be reduced to a collection of short substreams during intersection. As discussed in Section 4, operating with a stream whose length is less than the SIMD width will necessarily result in less than 100% utilization.

This observation suggests a different parallel processing scheme for primitive intersection: instead of processing streams of rays that require intersection with the same primitive, the algorithm could be modified to combine substreams for different primitives. A memory architecture with efficient scatter/gather operations enables each stream element to process a different primitive, though merging the results would require additional care.

Potentially low SIMD utilization also suggests the use of shallow acceleration hierarchies. Instead of intersecting two small BVH leaves with three triangles each, intersecting a single larger node with six triangles reduces the number of traversal operations. Packet-based optimizations such as vertex culling already show that shallow hierarchies can be exploited to good benefit, and stream filtering does not preclude the use of such techniques.

Memory performance. A similar tension exists between stream size and performance of the memory subsystem. Effectively exploiting stream filtering depends on a reasonably large number of rays in the input stream. Although large streams increase SIMD utilization and thus rendering performance, these streams might nonetheless perform badly due to effects such as cache spilling. However, as shown in Section 4, a well-designed memory subsystem can mitigate these potentially ill effects.

Related to this observation is the necessity of a maximum stream size. Any implementation of stream filtering will likely be required to enforce a maximum stream size; for example, our simulator enforces a maximum stream size of either 1024 or 4096 rays. This upper bound may be problematic for material shaders that generate a variable number of secondary rays, for example, an ambient occlusion shader that spawns 16 rays for each intersection point. Stream filtering may be combined with explicit ray scheduling techniques and multiple output buffers to alleviate this issue.

Hardware support for key operations. Lack of support for wider-than-four SIMD processing and efficient scatter/gather operations in conventional architectures currently represents the biggest challenge facing the proposed approach. Realizing the performance benefits offered by the algorithm with current CPUs may not be trivial. Nevertheless, we believe that as processors continue to rely on increasing levels of fine-grained parallelism, hardware support for wide SIMD processing and non-sequential memory access will become commonplace. With these architectures, then, stream filtering becomes an attractive option for interactive ray tracing.

6 Conclusions and Future Work

We have introduced a new approach to coherent ray tracing that yields high utilization with wider-than-four SIMD units operating on arbitrarily-sized groups of rays. Stream filtering eliminates inactive elements during traversal, intersection, and shading to maximize coherence in subsequent operations. This approach ensures that these operations process input streams containing only active rays. The key strengths of stream filtering include:

- **Parallel processing.** The algorithm achieves high SIMD utilization by exploiting the parallelism inherent to any collection of rays.
- **Implicit reordering.** The algorithm extracts active rays with respect to scene geometry, acceleration structure, material shaders, and so forth, and does not depend on presorting operations or ray coherence heuristics.
- **Hardware requirements.** The algorithm imposes modest memory requirements and supports arbitrary SIMD widths; as such, stream filtering is amenable to a wide range of future hardware architectures.
- **Generality.** The algorithm is generally applicable to all hierarchical acceleration structures and any type of primitive, and thus supports a wide range of ray tracing applications.

As noted in Section 5, the biggest challenge facing the proposed approach is that current processors do not support wider-than-four SIMD processing nor efficient scatter/gather operations. However, the algorithm is not necessarily intended for these processors, but targets future platforms that will likely rely on increasing levels of fine-grained parallelism.

Stream filtering opens a new design space that offers many interesting implementation alternatives. We believe that the simulator presented in this work provides a compelling design for future ray-based graphics hardware, and we plan to explore both heterogeneous and homogeneous multicore designs to support renderers based on stream filtering. We also plan to explore real-time implementations of the algorithm with current processors in an attempt to eliminate the need for hardware simulation. With increasing support for SIMD parallelism expected in new generations of commodity architectures, we hope to achieve real-time performance with stream filtering for a wide variety of rendering algorithms.

Acknowledgments

Ingo Wald, Solomon Boulos, and Andrew Kensler were involved with an early implementation of the approach [24] and we appreciate their contributions. Andrew Kensler first recognized that stream partitioning can be done in-place, and Abe Stephens independently, though later, made the same observation. Discussions with several people have helped shape these ideas, including Erik Brunvand, Al Davis, Steve Parker, Peter Shirley, and others.

Christiaan Gribble was supported by a grant from the Swezey Scientific Instrumentation Fund. Karthik Ramani was supported by a University of Utah graduate research fellowship and by NSF grants 0541009 and 0430063.

Temple of Kalabsha model courtesy of Veronica Sundstedt, Patrick Ledda, and the University of Bristol Computer Graphics Group.

References

[1] C. Benthin. *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Saarland University, 2006.

[2] 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.

[3] S. Boulos, D. Edwards, J. D. Lacewell, J. Kniss, J. Kautz, I. Wald, and P. Shirley. Packet-based Whitted and distribution ray tracing. In *Graphics Interface 2007*, pages 177–184, May 2007.

[4] S. Boulos, I. Wald, and P. Shirley. Geometric and arithmetic culling methods for entire ray packets. Technical Report UUCS-06-10, University of Utah, 2006.

[5] D. Burger and T. M. Austin. The simplescalar toolset, version 2.0. Technical Report TR-97-1342, University of Wisconsin-Madison, June 1997.

[6] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonte, J.-H. Ahn, N. Jayasent, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck. Merrimac: Supercompute with streams. In *SC2003*, 2003.

[7] D. Horn. Stream reduction operations for GPGPU applications. In *GPU Gems 2*, page 573. Addison-Wesley.

[8] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Change, and S. Rixner. Imagine: Media processing with streams. *IEEE Micro*, 21(2):35–46, 2001.

[9] J. Mahovsky and B. Wyvill. Memory-conserving bounding volume hierarchies with coherent raytracing. *Computer Graphics Forum*, 25(2):173–182, 2006.

[10] E. Mansson, J. Munkberg, and T. Akenine-Moller. Deep coherent ray tracing. In *2007 IEEE Symposium on Interactive Ray Tracing*, pages 79–85, September 2007.

[11] B. K. Mathew, A. Davis, and M. A. Parker. A low power architecture for embedded perception. In *CASES '04: International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 46–56, September 2004.

[12] K. Nakamaru and Y. Ohno. Breadth-first ray tracing utilizing uniform spatial subdivision. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):316–328, 1997.

[13] P. Navratil, D. Fussell, C. Lin, and W. R. Mark. Dynamic ray scheduling for improved system performance. In *2007 IEEE Symposium on Interactive Ray Tracing*, pages 95–104, September 2007.

[14] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. Rendering complex scenes with memory-coherent ray tracing. *Computer Graphics*, 31(Annual Conference Series):101–108, 1997.

[15] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002.

[16] K. Ramani and A. Davis. Application driven embedded systems design: A face recognition case study. In *CASES '07: International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 103–114, 2007.

[17] A. Reshetov. Omnidirectional ray tracing traversal algorithm for kd-trees. In *2006 IEEE Symposium on Interactive Ray Tracing*, pages 57–60, September 2006.

[18] A. Reshetov. Faster ray packets-triangle intersection through vertex culling. In *2007 IEEE Symposium on Interactive Ray Tracing*, pages 105–12, September 2007.

[19] A. Reshetov, A. Soupikov, and J. Hurley. Multi-level ray tracing algorithm. *ACM Transacions on Graphics*, 24(3):1176–1185, July 2005.

[20] J. Schmittler, I. Wald, and P. Slusallek. SaarCOR: A hardware architecture for ray tracing. In *Eurographics Workshop on Graphics Hardware*, pages 27–36, September 2002.

[21] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3), 2008. To appear.

[22] I. Wald, C. Benthin, M. Wagner, and P. Slusallek. Interactive rendering with coherent ray tracing. *Computer Graphics Forum*, 20(3):153–164, September 2001.

[23] I. Wald, S. Boulos, and P. Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics*, 26(1):6, January 2007.

[24] I. Wald, C. Gribble, S. Boulos, and A. Kensler. SIMD ray stream tracing. Technical Report UUSCI-2007-012, University of Utah, 2007.

[25] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker. Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics*, 25(3):485–493, July 2006. (Proceedings of Siggraph '06).

[26] T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.

[27] S. Woop, J. Schmittler, and P. Slusallek. RPU: a programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics*, 24(3):434–444, 2005.