Abstract

The wide availability of commodity graphics processors has made real-time graphics an intrinsic component of the human/computer interface. These graphics cores accelerate the z-buffer algorithm and provide a highly interactive experience at a relatively low cost. However, many applications in entertainment, science, medicine, etc. require higher quality lighting effects such as accurate shadows and reflections. These effects are difficult to achieve with z-buffer algorithms but are much easier to achieve using ray tracing. Ray tracing is computationally more complex, but has better scaling and parallelism properties than the z-buffer approach. However, ray tracing patterns are difficult to predict and therefore, the parallelism promise is hard to achieve. This paper highlights a novel ray tracing approach based on stream filtering, and presents StreamRay, a multi-core wide SIMD machine that employs efficient address generation mechanisms to form a stream of rays for highly parallel SIMD execution. Results demonstrate that StreamRay delivers interactive frame rates of 15-25 fps for scenes of high geometric complexity while sustaining high utilization for SIMD widths of up to 16 elements.
Stream Filtering in StreamRay: An Architecture for Coherent Ray Tracing

Abstract

The wide availability of commodity graphics processors has made real-time graphics an intrinsic component of the human/computer interface. These graphics cores accelerate the z-buffer algorithm and provide a highly interactive experience at a relatively low cost. However, many applications in entertainment, science, medicine, etc. require higher quality lighting effects such as accurate shadows and reflections. These effects are difficult to achieve with z-buffer algorithms but are much easier to achieve using ray tracing. Ray tracing is computationally more complex, but has better scaling and parallelism properties than the z-buffer approach. However, ray tracing patterns are difficult to predict and therefore, the parallelism promise is hard to achieve. This paper highlights a novel ray tracing approach based on stream filtering, and presents StreamRay, a multi-core wide SIMD machine that employs efficient address generation mechanisms to form a stream of rays for highly parallel SIMD execution. Results demonstrate that StreamRay delivers interactive frame rates of 15-25 fps for scenes of high geometric complexity while sustaining high utilization for SIMD widths of up to 16 elements.

Keywords. Graphics processors, ray tracing, interactive rendering.

1 Introduction

Graphics processing units (GPUs) have significantly enhanced human/computer interfaces and have revolutionized the use of computers for both work and entertainment. GPUs act as accelerators for the z-buffer algorithm [3], which in its most basic form consists of a loop over the objects in a scene. While this algorithm provides good interactivity, it is not well-suited to images that require high quality shadows, reflections, etc. which are important for applications in a variety of disciplines.

Ray tracing [13] consists of a loop over all of the pixels in an image and boasts several key advantages over the z-buffer algorithm. First, for preprocessed models, ray tracing is sub-linear in the number of objects, \( N \); thus, for some sufficiently large value of \( N \), ray tracing will always be faster than the z-buffer algorithm, which is linear in \( N \) [4]. Second, the computational kernel of the algorithm performs a 3D line query, and that same operation can be reused to generate global illumination effects such as shadows, reflection, and refraction [13]. Third, ray tracing is highly parallel and has been demonstrated to have over 91% efficiency on 512 processors [7]. This characteristic, combined with the advent of multi-core microprocessors and algorithmic developments make ray tracing an attractive alternative to interactive rendering if a solution can be found to mitigate problems associated with ray tracing’s less predictable memory access patterns.

2 Stream Filtering

Coherent or packet-based ray tracing [12] enables the efficient use of SIMD processing. In this approach, rays are processed in coherent groups utilizing SIMD instructions such as SSE or Altivec. While highly successful, packet-based ray tracing is efficient only for highly coherent packets. When rays begin to diverge, a large percentage of the packet’s rays do not actively participate in the same computations. As a result, unnecessary work is performed on the inactive subsets, which degrades performance and increases power consumption.

This work explores a multi-core \( N \)-wide SIMD architecture for stream filtering [5], a new approach to coherent ray tracing that processes arbitrarily large ray streams in SIMD fashion. Each of the major stages of ray tracing (traversal, intersection, and shading) can be written as a sequence of conditional statements called stream filters that are executed in SIMD fashion across groups of \( N \) rays. These filters are used to isolate only those
rays from within a stream that exhibit some property of interest. For example, at each step of the traversal process, a stream filter can be used to isolate only those rays that actually intersect an object and thus require further processing. The filter creates an active substream of these active rays, and traversal then continues. Using an appropriate sequence of stream filters, inactive rays are eliminated on-the-fly during rendering, which improves SIMD utilization and leads to improved performance. We also note that any substream created by a filter operation is optimal with respect to the original stream: all rays from the stream that require the same sequence of subsequent operations will always perform those operations together, regardless of the order in which these rays occur within the original stream or the processing that they have received prior to reaching the filter.

Stream filtering opens a new design space that offers many interesting implementation alternatives, one of which is described in this study. The StreamRay architecture consists of two major subsystems: the Ray engine that employs address computation mechanisms to form large data streams for SIMD processing, and the Filter engine that employs many energy-efficient accelerator cores to perform partitioning and wide SIMD operations. Results demonstrate that this architecture can deliver interactive frame rates of 15-25 frames/second for a variety of scenes (rrtr shown in Figure 1). StreamRay supports streaming filtering and provides an attractive alternative to the traditional packet-based methods used in interactive ray tracing; it improves SIMD utilization by exploiting the parallelism inherent to arbitrarily-sized groups of rays and is general enough to support many ray tracing applications.

3 Architecture Description

An architecture that efficiently implements stream filtering needs to provide two core capabilities: stream assembly to form sequential streams of ray data, and stream filters that evaluate a set of conditional statements to determine whether or not a ray should be placed in the active output stream. Stream assembly exhibits non-sequential memory access patterns, which in turn require address computations to support scatter/gather operations. Stream filters in an N-wide SIMD environment require two steps: (1) conditional statements are applied to groups of N elements from the input stream, creating a boolean mask; (2) the input stream is then partitioned into active and inactive subsets based on the boolean mask.

As shown in Figure 2, StreamRay employs a two-tiered approach to provide the core capabilities. The Ray engine employs efficient address generation mechanisms to support stream assembly. This engine is programmed using C++ templates and supervises data movement for stream assembly. The Filter engine consists of N program controlled kernel accelerators that implement each of the ray tracing kernels (traversal, intersection, and shading) in a N-wide SIMD environment.

**The Ray Engine.** The ray engine (Figure 3) consists of two subsystems: the address fetch unit (AFU) and the ray memory unit. To form a sequential stream of data, N non-sequential memory addresses need to be computed. The address fetch unit consists of N address generator units (AGUs) [8] that provide support for scatter/gather, strided, or sequential addressing. Each of the AGUs also contain an integer execution unit to perform address computations. The ray memory unit is a distributed system that consists of two ray buffers and a dual-ported scratch pad memory for storing texture data. The ray buffers facilitate data movement between the main memory and the filter execution engine, so StreamRay employs two such buffers for decoupling: one for current generation rays that are accessed by the filter engine, and one for next-generation rays where data is fetched to and from main memory by the address fetch unit. To provide support for efficient N-wide SIMD processing, each of the ray buffers is banked into N single-ported ways and each of the AGUs fetch data to one bank. Banking is an efficient alternative to multi-ported buffers, which are expensive in terms of area and power. Provided that requests do not collide frequently, this design efficiently provides data to the filter engine. Performance is improved as a result of minimized communication and efficient isolation between stream formation and kernel computations.
The Filter Engine. The filter engine implements the filter operations that partition the stream of rays into active and inactive subsets to exploit the coherence via wide SIMD processing. A set of $N$ accelerators implement the various kernels in ray tracing, including traversal, intersection, and shading. The architecture of the accelerator is shown in figure 2. Horizontal micro-code is generated by a compiler [8] and is stored in the kernel control block. The microcode explicitly controls instruction and data movement. Each accelerator contains two sets of register files and a set of execution units and comparators. While the execution units implement the kernel, the comparators partition the input set into active and inactive sets. The execution units may be SIMD or scalar. The execution units are backed by pipelined registers and a multiplexer-based interconnect [8] and can be configured by the program on a cycle-by-cycle basis. Thus, the result is a programmable accelerator whose energy-delay characteristics approach that of an ASIC.

Orchestration. The interconnect subsystem coordinates data movement across the two engines and is critical to the performance of the system. The StreamRay architecture consists of a simple multiplexer-based interconnect in which each of the accelerators or banks can transmit data in a single cycle to either of its neighbors (left or right). When compared to a fully-connected $N \times N$ network, performance degrades by only 4% but area is reduced by at least $3 \times$ [1], [8]. The stream control block issues the load/store operations from/to memory and supervises synchronization for the architecture. While the next-generation ray buffer is filled with data, the current generation ray buffer communicates with the filter engine. The kernel control block synchronizes the accelerators across the kernel boundaries. For ray tracing operations that are split into subkernels (for example, shading operations), synchronization occurs across the subkernel boundary. Overall, the StreamRay architecture supports the stream filtering algorithm by providing the following salient features:

- The ray engine performs stream assembly while the Filter engine implements the $N$-wide SIMD processing operations. Isolation of the two computations reduces data movement and contention for the interconnect, thereby improving performance.
- The filter engine implements on-the-fly re-ordering via the comparators to remove the inactive ray elements, thereby improving SIMD utilization.
- The architecture is general enough to support other forms of ray tracing and is applicable to other hierarchical acceleration structures and primitive types.

4 Evaluation

A cycle-accurate simulator similar to the SimpleScalar tool set [2] is used to evaluate the architecture. Stalls resulting from data alignment operations are modeled accurately, as is interconnect, function-unit, and memory contention. The architectural parameters are shown in table 1. Images are generated using a Monte Carlo path tracer [6] 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 renderer also uses a thin-lens camera model to simulate depth-of-field effects. 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. This study renders three scenes of varying geometric complexity, visual effects, and shader types [5]. To evaluate the performance of StreamRay, we employ two metrics: SIMD utilization, as a measure of the number of useful operations that are performed within an $N$-wide SIMD unit; and frame rate, as a measure of rendering performance in terms of frames/second (fps). We report SIMD utilization for primary and secondary rays for all three stages of rendering and expect to render each of the scenes at interactive frame rates (above 10 fps).

We observe high utilization rates for the three scenes for primary rays. For traversal, utilization is as high as 95% for a SIMD width of 8 and marginally reduces to around 91% for a SIMD width of 16. For intersection, we observe utilization rates of around 90%. Further, as the size of the initial stream increases, we observe that utilization increases significantly because inactive rays are automatically removed from the output stream. SIMD utilization for traversal, intersection, and shading ($T / I / S$) in the path tracer for secondary rays is shown in table 2. Utilization remains reasonably high under a variety of SIMD widths, with larger initial ray streams again leading to higher utilization in all stages for all scenes. Stalls arise because of overheads introduced by address fetch and alignment in the ray engine, and by data partitioning in the Filter engine.

Frame rates, shown in Table 3, increase with the SIMD width, due to the reduced number of alignment and partitioning oper-
ations. On the other hand, wider SIMD units require more time for address computation and have higher address fetch overhead. The inherent trade-off between SIMD width and fetch overhead is the fundamental performance constraint. In particular, for \textit{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 due to reduced SIMD alignment and stream partitioning overhead. Beyond a width of 12, the overhead of address computation begins to dominate, and improvements diminish accordingly. Thus, a 12-wide SIMD machine is the optimum choice in this case-study.

The results demonstrate that StreamRay achieves interactive frame rates for complex scenes using path tracing and a variety of visual effects. 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, stream filtering architectures become a viable alternative for interactive ray tracing.

5 Related Work

The use of ray packets to exploit SIMD processing was first introduced by Wald et al. [12]. 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 \times 4 rays [1], but these fixed-size packets are neither split nor reordered. Reshetov [9] has shown that even for narrow SIMD units, perfectly specular reflection rays undergoing multiple bounces quickly lead to almost completely incoherent ray packets and \( \frac{1}{2} \) SIMD efficiency. In contrast, our evaluation demonstrates that it is possible to achieve high SIMD utilization for wide SIMD units by employing filtering to remove the inactive subsets. SaarCor [10] is a custom, hard-coded ray trace processor, while RPU [14] has a custom kd-tree traversal unit with a programmable shader. Our design is intended to support a variety of ray tracing algorithms. While commercial implementations like the G80 and the R770 provide wider-than-four SIMD capability, these machines employ the execution core for address computations and hence, interfere and compete with the actual data computations for resources, thus degrading performance. The Larrabee project [11] employs a many-core task-parallel architecture to support a variety of applications. In contrast, StreamRay delivers high performance by efficiently isolating the core concepts of stream generation and stream processing to deliver high performance.

6 Conclusion

Stream filtering is a new approach to ray tracing which creates arbitrarily-sized coherent ray groups that can efficiently utilize wider-than-four SIMD units. The Stream-Ray architecture provides hardware support for this approach and has been shown to deliver interactive frame rates for a variety of scenes. Preliminary results demonstrate that this technique delivers high performance and also opens up a new design space for ray tracing accelerators.

References


<table>
<thead>
<tr>
<th>size</th>
<th>\textit{rtrt}</th>
<th>conf</th>
<th>kala</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SIMD width</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N (= 8)</td>
<td>16.60 fps</td>
<td>8.15 fps</td>
<td>6.73 fps</td>
</tr>
<tr>
<td>32 \times 32</td>
<td>18.78 fps</td>
<td>12.78 fps</td>
<td>8.34 fps</td>
</tr>
<tr>
<td></td>
<td>SIMD width</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N (= 12)</td>
<td>21.82 fps</td>
<td>12.56 fps</td>
<td>11.78 fps</td>
</tr>
<tr>
<td>64 \times 64</td>
<td>24.52 fps</td>
<td>18.32 fps</td>
<td>13.45 fps</td>
</tr>
<tr>
<td></td>
<td>SIMD width</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N (= 16)</td>
<td>22.36 fps</td>
<td>14.35 fps</td>
<td>13.34 fps</td>
</tr>
<tr>
<td>64 \times 64</td>
<td>26.35 fps</td>
<td>20.32 fps</td>
<td>15.65 fps</td>
</tr>
</tbody>
</table>

TABLE 3. Rendering performance