An Analysis of Ray Tracing Bandwidth Consumption

Paul Arthur Navrátil William R. Mark*

The University of Texas at Austin
Intel Corporation

Figure 1: The six scenes used in this paper, ordered by increasing visible triangle count (visible tris, total tris): **erw6** (0.5K, 0.8K); **soda hall** (36K, 2195K); **conference** (54K, 283K); **statue** (369K, 1088K); **tri-statue** (1082K, 3263K); **dragon** (2261K, 7219K).

ABSTRACT

The trend in chip-multi-processors for the next several years is for on-chip FLOPS to grow much faster than bandwidth to off-chip DRAM. This trend is likely to emerge as a performance bottleneck for future real-time ray tracing systems. In this paper, we assess the impact of this bottleneck by measuring the DRAM bandwidth requirements for several different ray tracing algorithms, each running on simulated architectures with a variety of cache sizes. We conclude that for current packet-tracing algorithms, bandwidth will not be a bottleneck for primary rays, but that it will be a bottleneck for soft shadow rays. This bottleneck is caused primarily by dramatically lower cache hit rates, rather than by an increase in total working set, which suggests that substantial reductions in memory bandwidth requirements would be possible by designing algorithms that do a better job of scheduling ray traversals in a coherent fashion for divergent secondary rays.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing;

Keywords: ray tracing, memory bandwidth

1 INTRODUCTION

Ray-tracing based rendering algorithms can generate images that are of better visual quality than those generated easily by a Z-buffer algorithm, but ray tracing algorithms have traditionally been too slow for real-time use. Recent work demonstrates that real-time ray tracing is possible for primary rays and hard shadows[10][8][11][2], but it is still an open question as to when it will become feasible to ray trace soft shadows and other advanced effects in real time.

Performance of future ray-tracing systems for soft shadows and other secondary rays could be limited by either raw FLOPS or by DRAM bandwidth. In designing algorithms for future systems, it is important to understand which potential bottlenecks are relevant. This paper examines the potential DRAM bandwidth bottleneck by presenting measurements of the bandwidth requirements of popular packet-based ray tracing algorithms. We report results for both primary rays and soft shadow rays.

Because the rays within a soft-shadow secondary-ray packet tend to diverge more rapidly than rays within an primary-ray packet, we would expect that memory accesses for soft-shadow rays would be less coherent, and in fact we find that this is the case. We conclude that today's popular algorithms are inefficient with respect to bandwidth usage for divergent secondary rays, and that these algorithms may have to be adapted or replaced to improve their utilization of the memory hierarchy.

The remainder of the paper is organized as follows: we describe related work in Section 2. In Section 3 we describe our experimental method. We present our results in Section 4, and in Section 5 we summarize and outline future work.

2 RELATED WORK

There are several published results about the bandwidth requirements for tracing primary rays and hard shadows. Schmittler, et al.,[9][10] and Woop, et al.,[11] report primary ray and hard shadow memory bandwidth consumption on dedicated ray tracing hardware. Benthin, et al.,[2] measure memory traffic for primary rays on the IBM CELL processor. Purcell, et al.,[7] measure bandwidth between on-board RAM and processor for GPU-based ray casting. When scaled by cache size, our measured bandwidth consumption for primary rays and hard shadows are similar to these related results.

We are aware of only a few published bandwidth results for ray tracing with divergent secondary rays. Purcell, et al.,[7] report bandwidth consumption for specular reflection and for path tracing for one scene on a GPU. They find that adding secondary rays increases DRAM to core memory traffic $3\times$ to $4\times$ primary ray traffic. Pharr, et al.,[6] describe bandwidth consumption between main memory and disk for a complex scene rendered with a Monte Carlo global illumination simulation. Though this paper deals with a lower level of the memory hierarchy, they find significant excess bandwidth consumption due to thrashing data from disk to main memory. We will compare their findings with ours in Section 5.

*e-mail:[pnav | billmark]@cs.utexas.edu

Processor	L1 cache configuration	L2 cache configuration	DRAM to L2 bandwidth
Intel Prescott	16KB, 8-way, 64B lines	1MB, 8-way, 64B lines	10.7 GB/s (975x chipset)
AMD Toledo	64KB, 2-way, 64B lines	1MB, 16-way 64B lines	8.0 GB/s
Sun Niagara (constructive)	8KB, 4-way, 16B lines	4MB, 12-way, 16B lines	20 GB/s
Sun Niagara (destructive)	2KB, 4-way, 16B lines	128KB, 12-way, 16B lines	20 GB/s
IBM CELL			25.6 GB/s
GeForce 7900 GTX			51.2 GB/s
SaarCOR			1 – 2 GB/s

Table 1: System Configurations — the first four rows contain the systems we simulate: two traditional CPUs and two cases for a CMP system. We include other rendering hardware for bandwidth comparison only.

3 Experimental Method

We measure memory traffic for several different ray tracing algorithms (e.g. primary rays only vs. soft shadows); for several different scenes; and for several different cache configurations. We describe the various algorithms, scenes, and cache configurations in this section.

3.1 Cache Configurations

We use cache configurations that correspond to those of several current processors, including two traditional CPUs (AMD and Intel), and a CMP system (Sun Niagara).

Since there are various architecture choices for CPU-based ray tracers, we take a current sample from each of the major vendors: Intel and AMD. We use the configuration of an Intel Prescott core with 975x chipset[4] and an AMD Toledo core and memory controller[1]. We choose the Sun Niagara processor[5] (with floating-point processing assumed) for our CMP model, since it has hardware multi-threading, a large L2, and high DRAM to L2 bandwidth. Note that Niagara has a 3MB L2 cache, but our cache simulator forces us to use power-of-two sizes. We conservatively use a 4MB L2. The system configurations are summarized in Table 1.

We model two configurations for the Niagara: a constructive cache interference pattern, where all processing resources can share cached data; and a destructive cache interference pattern, where no processing resource shares data. For the destructive interference pattern, we partition memory-system resources evenly to each thread on each core. These two interference patterns establish upper and lower limits on memory system performance.

3.2 Scenes

We use six scenes in our survey (shown in Figure 1), each rendered at 1024×1024 resolution. These scenes represent a four-order-of-magnitude range of geometric complexity. We choose **erw6** , **conference** , **soda hall** and **dragon** since they appear in previously reported performance results for the ray tracer we use[8], which permits us to correlate our memory-system results with these previous results. We use two additional scenes containing the Stanford Buddha statue so that we have more scenes with many visible triangles. We trace each scene using four different methods: primary rays only, hard shadows, soft shadows, and ambient occlusion. These four methods represent a range of ray complexity to help judge the capacity of each memory system. We generate one frame's worth of memory traffic data for each scene. We use the single-frame data to estimate animation memory traffic at 60 fps.

During rendering, the number of visible triangles has greater effect on memory bandwidth consumption than does the total number of triangles in the scene. Even though **soda hall** has many more triangles than **conference** , **soda hall** has 20K fewer visible triangles from our selected viewpoints. Thus we expect rendering **conference** to consume more bandwidth than rendering **soda hall** .

3.3 Ray Tracing Algorithms

Our work uses a state of the art ray tracer as the basis for our architecture study. The MLRTA system[8] has achieved the best published frame rates to date on CPUs for static scenes.

Primary rays are traced in packets of 16 rays, generated through contiguous 4×4 pixel blocks. Hard shadow rays are traced in packets of 16 rays, where the shadow rays correspond directly to the hit points from a packet of primary rays. These rays are traced from point light to hit point, with last-occluder early termination testing.

Soft shadow sampling is performed in 4×4 packets that also correspond directly to hit points for a primary ray packet. Shadow ray origins are generated randomly on the surface of the area light, with a separate origin created for each ray in the packet. Nine soft shadow samples per primary-ray hit point are generated per sampling round. If the surface is too rough, only one sampling round is performed. Otherwise, soft shadow sampling continues until sample variance is under a given threshold. Surface roughness is determined by taking the cosine of the angle between surface normals at each pair of hit points from the primary ray packet and comparing the minimal cosine value against a threshold.

The ambient occlusion algorithm traces 16 primary rays per pixel in jittered 4×4 packets. Each packet covers a 4×4 pixel block, and sixteen such packets are traced per block. Nine random occlusion rays are generated per primary ray hit, for a maximum of 144 occlusion rays per pixel. The algorithm resamples until sample variance is under a given threshold (four iterations max).

3.4 Measurement Methodology

We create a trace of actual memory used by the MLRTA ray tracer when rendering each comination of scene and ray tracing algorithm. We record a cache read for ray and node data at each traversal step and a read for ray and geometry data at each intersection step. We never record cache writes. We use the **Dinero IV** cache simulator[3] to model each memory system. This light-weight simulator provides cache usage statistics without modeling functionality or providing timing estimates.

Our measurements conservatively estimate the memory system resources required to process the sample loads because we only model data reads during ray traversal and intersection. We do not model acceleration structure generation, instruction cache traffic, shading or writing the image. As such, we expect our measurements to serve as a lower bound for the memory-system requirements for ray tracing systems.

We report our findings both in terms of bandwidth consumed and in terms of cache efficiency. We define cache efficiency as compulsory bandwidth consumed (i.e. total size of working set) divided by total bandwidth consumed. With this measure, we can estimate how much cache utilization could be improved.

		primary rays only			primary + soft shadows		
machine	scene	L2 to L1 traffic	DRAM to L2 traffic	compulsory traffic	L2 to L1 traffic	DRAM to L2 traffic	compulsory traffic
Intel Prescott	conference	14 649 856	4 836 416	4 833 024	902 914 496	28 859 264	8 705 280
	statue	63 664 000	32 026 688	32 001 472	16 886 719 744	9 090 462 080	54 827 456
AMD Toledo	conference	11 039 616	4 835 776	4 833 024	371 575 744	29 347 904	8 705 280
	statue	59 739 840	32 011 072	32 001 472	16 121 283 648	8 795 612 672	54 827 456
Sun Niagara (constructive)	conference	13 726 672	4 214 736	4 214 736	977 741 632	8 230 672	8 220 272
	statue	49 424 016	28 732 560	28 732 560	10 673 545 664	1 929 019 776	51 919 504
Sun Niagara (destructive)	conference	40 396 432	5 420 320	4 214 736	2 785 107 632	106 545 280	8 220 272
	statue	78 355 200	41 465 696	28 732 560	12 070 645 792	9 737 228 000	51 919 504

Table 2: Memory Traffic — total data traffic in bytes from DRAM to L2 and from L2 to L1 for two representative scenes (**conference** and **statue**), each rendered with only primary rays and with primary + soft shadows. The third column under each lists compulsory traffic. Dividing compulsory traffic by total traffic provides our efficiency measurement for each cache level. Note that these traffic data are for a single frame. In Figure 2 and Figure 3, we extrapolate to 60 fps by multiplying the per-frame data by 60. Also note that compulsory traffic differs slightly between the CPUs and the CMP simulations due to their different cache line sizes.

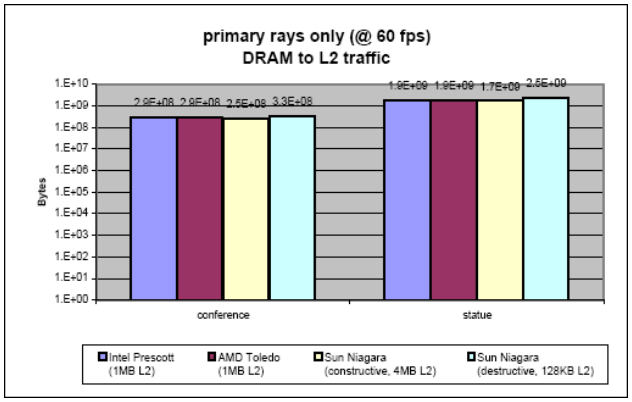

Figure 2: DRAM to L2 data traffic in bytes for primary rays, extrapolated to 60 fps. The traffic is well within current DRAM to L2 bandwidth rates.

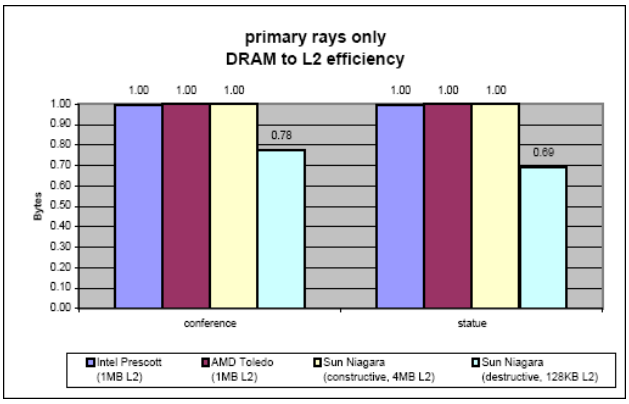

Figure 4: DRAM to L2 efficiency for primary rays only — when tracing only primary rays, bandwidth between DRAM and L2 is used efficiently, even for small L2 sizes (Sun Niagara (destructive)).

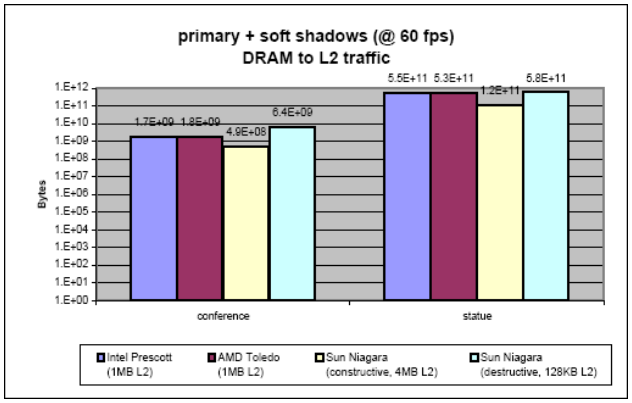

Figure 3: DRAM to L2 data traffic in bytes for primary + soft shadows, extrapolated to 60 fps. When there are few visible triangles (**conference**), traffic is within current DRAM to L2 bandwidth rates. When there are many visible triangles (**statue**), traffic exceeds current bandwidth rates by $10\times$ or more.

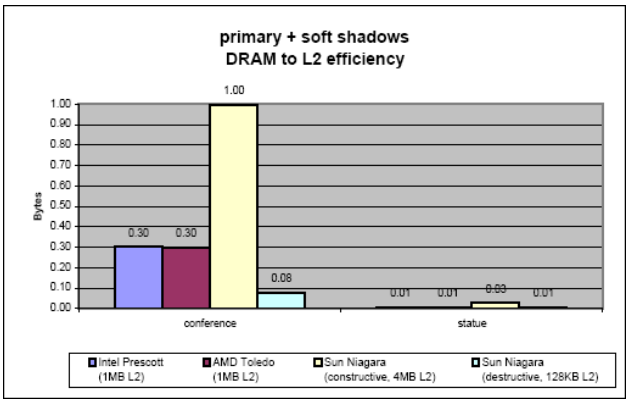

Figure 5: DRAM to L2 efficiency for primary + soft shadows — when tracing soft shadows, bandwidth efficiency is maintained when the working set can be maintained in cache (Sun Niagara (constructive)). When the working set cannot be maintained in cache, cache efficiency degrades significantly (**conference**) or catastrophically (**statue**).

4 RESULTS

We present memory traffic and memory efficiency data for the four memory systems described in Section 3.1. The data in Table 2 is for rendering a single frame at 1024×1024 resolution. The data in Figures 2–5 are extrapolated to 60 fps from single-frame data.

We gather data for four ray tracing algorithms (see Section 3.3) on each of six scenes (see Section 3.2). Since the results for primary rays and for hard shadows were similar, we report only results for primary rays. Likewise, results for soft shadows and for ambient occlusion were similar, so we report only results for soft shadows.

Memory traffic for divergent secondary rays (soft shadows, ambient occlusion) is largely determined by the number of visible triangles in the scene. There is a natural separation in our data between scenes with few visible triangles (**erw6** , **soda hall** , **conference**) and scenes with many visible triangles (**statue** , **tri-statue** , **dragon**). Results within each group are similar, so we report only the results for one scene in each group.

As the data in Table 2 shows, rendering with divergent secondary rays can increase bandwidth consumed between main memory and L2 by an order of magnitude or more. However, for scenes with many visible triangles, rendering with divergent secondary rays can increase bandwidth consumed by two orders of magnitude or more. In Figure 2 and Figure 3, we extrapolate these data to rendering at 60 fps. Bandwidth demand for scenes with few visible triangles is within current memory-to-core bandwidth rates, but bandwidth demand for scenes with many visible triangles exceeds current bandwidth rates by more than an order of magnitude.

By comparing the total bytes loaded to the compulsory byte loads (Figure 4 and Figure 5), we see that there is a large difference in how efficiently the L2 cache is used. Primary rays produce high L2 hit rates, whereas divergent secondary rays produce much lower hit rates. The Niagara results are particularly evocative. First, consider the results for few visible triangles. In the constructive interference case, where the entire 4MB of cache is available for scene data, the L2 hit rate is high; in the destructive interference case, where only a fraction of the L2 is available, the L2 hit rate is considerably lower. When we consider the results for many visible triangles, we see that the hit rate is miserable for both L2 cases.

5 SUMMARY AND FUTURE WORK

We have examined the DRAM bandwidth bottleneck for ray tracing systems, and we conclude that current traversal algorithms are inefficient with respect to divergent secondary rays. This inefficiency is a result of poor cache management rather than a dramatic increase in the working set. Future ray tracing systems must either maintain better ray coherence or increase available cache to maintain the entire working set. Since new processor designs have less cache per-core, ray tracing algorithms for such systems may need to be modified or replaced to achieve better utilization of the memory hierarchy.

A ray reordering algorithm similar to Pharr's[6] may provide sufficient secondary ray coherency. They render a 9.6M triangle scene with Monte Carlo path tracing on a system with 325MB RAM. This generates 120MB of compulsory geometry traffic and 2.1GB of additional traffic from incoherent geometry accesses (0.05 efficiency, by our measure). With the ray reordering technique they propose, geometry traffic can be reduced or eliminated while also reducing the total memory required. With ray reordering and only 50MB RAM, they cut geometry traffic to 938MB plus 70MB ray traffic overhead. With ray reordering and 120MB of RAM, they eliminate geometry traffic and consume only 70MB of ray traffic. For such a technique to be viable, it should be compatible with modern surface shaders and SIMD-based instruction optimizations. The additional memory traffic generated by maintaining ray state explicitly must be factored against any savings in geometry memory traffic.

An analytic model of ray traversal and intersection would deepen our understanding of the coherence problem and would provide insight for possible solutions. A well-constructed model will qualify the inefficiencies in current ray tracing algorithms and will provide a quantifiable measure of improvement for new approaches.

ACKNOWLEDGMENTS

We would like to thank to Alexander Reshetov for allowing us to use the MLRTA codebase and for orienting us to its use. Thanks to Igor Sevastianov for his instructions on how to use ambient occlusion in MLRTA. This research was performed with support from Intel Corporation.

REFERENCES

[1] Advanced Micro Devices, Inc. AMD Opteron(TM) Product Data Sheet (http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/23932.pdf).

[2] Carsten Benthin, Ingo Wald, Michael Scherbaum, and Heiko Friedrich. Ray tracing on the cell processor. In *submitted to the 2006 IEEE Symposium on Interactive Ray Tracing*, 2006.

[3] Jan Edler and Mark D. Hill. Dinero IV cache simulator (http://www.cs.wisc.edu/m̃arkhill/DineroIV/).

[4] Intel Corporation. Intel(R) 975x Express Chipset (http://www.intel.com/products/chipsets/975x/).

[5] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: a 32-way multithreaded SPARC processor. In *IEEE MICRO 2005*, volume 25, pages 21–29, 2005.

[6] 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, 1997.

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

[8] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. *ACM Trans. Graph.*, 24(3):1176–1185, 2005.

[9] Jörg Schmittler, Ingo Wald, and Philipp Slusallek. Saarcor - a hardware achitecture for ray tracing. In *Proceedings of Eurographics Workshop on Graphics Hardware*, pages 27–36. European Association for Computer Graphics, September 2002.

[10] Jörg Schmittler, Sven Woop, Daniel Wagner, Wolfgang J. Paul, and Philipp Slusallek. Realtime ray tracing of dynamic scenes on an FPGA chip. In *Graphics Hardware 2004*, 2004.

[11] Sven Woop, Jorg Schmittler, and Philipp Slusallek. RPU: a programmable ray processing engine. In *SIGGRAPH '05: Proceedings of the 32nd annual conference on Computer graphics and interactive techniques*, New York, NY, USA, 2005. ACM Press.