The Visual Computer (2018) 34:875–885
https://doi.org/10.1007/s00371-018-1532-8

ORIGINAL ARTICLE

A detailed study of ray tracing performance: render time and energy cost

Elena Vasiou[1] · Konstantin Shkurko[1] · Ian Mallett[1] · Erik Brunvand[1] · Cem Yuksel[1]

Published online: 30 April 2018

Abstract
Optimizations for ray tracing have typically focused on decreasing the time taken to render each frame. However, in modern computer systems it may actually be more important to minimize the energy used, or some combination of energy and render time. Understanding the time and energy costs per ray can enable the user to make conscious trade-offs between image quality and time/energy budget in a complete system. To facilitate this, in this paper we present a detailed study of per-ray time and energy costs for ray tracing. Specifically, we use path tracing, broken down into distinct kernels, to carry out an extensive study of the fine-grained contributions in time and energy for each ray over multiple bounces. As expected, we have observed that both the time and energy costs are highly correlated with data movement. Especially in large scenes that do not mostly fit in on-chip caches, accesses to DRAM not only account for the majority of the energy use, but also the corresponding stalls dominate the render time.

Keywords Ray tracing · Energy efficiency · Graphics processors · Memory timing

1 Introduction

Ray tracing [40] algorithms have evolved to be the most popular way of rendering photorealistic images. In particular, path tracing [19] is widely used in production today. Yet, despite their widespread use, ray tracing algorithms remain expensive in terms of both computation time and energy consumption. New trends arising from the need to minimize production costs in industries relying heavily on computer generated imagery, as well as the recent expansion of mobile architectures, where application energy budgets are limited, increase the importance of studying the energy demands of ray tracing in addition to the render time. A large body of work optimizes the computation cost of ray tracing by minimizing the number of instructions needed for ray traversal and intersection operations. However, on modern architectures the time and energy costs are highly correlated with data movement. High parallelism and the behavior of deep memory hierarchies, prevalent in modern architectures, make further optimizations non-trivial. Although rays contribute independently to the final image, the performance of the associated data movement is highly dependent on the overall state of the memory subsystem. As such, to measure and understand performance, one cannot merely rely on the number of instructions to be executed, but must also consider the data movement throughout the entire rendering process. In this paper, we aim to provide a detailed examination of time and energy costs for path tracing. We split the ray tracing algorithm into discrete computational kernels and measure their performance by tracking their time and energy costs while rendering a frame to completion. We investigate what affects and limits kernel performance for primary, secondary, and shadow rays. Our investigation explores the variation of time and energy costs per ray at all bounces in a path.

Electronic supplementary material The online version of this article (https://doi.org/10.1007/s00371-018-1532-8) contains supplementary material, which is available to authorized users.

✉ Elena Vasiou
elvasiou@cs.utah.edu

Konstantin Shkurko
kshkurko@cs.utah.edu

Ian Mallett
imallett@cs.utah.edu

Erik Brunvand
elb@cs.utah.edu

Cem Yuksel
cem@cemyuksel.com

[1] University of Utah , 50 Central Campus Dr, Sale Lake City, UT 84112, USA

Time and energy breakdowns are examined for both individual kernels and the entire rendering process. To extract detailed measurements of time and energy usage for different kernels and ray types, we use a cycle-accurate hardware simulator designed to simulate highly parallel architectures. Specifically, we profile TRaX [35,36], a custom architecture designed to accelerate ray tracing by combining the parallel computational power of contemporary GPUs with the execution flexibility of CPUs. Therefore, our study does not directly explore ray tracing performance on hardware that is either designed for general-purpose computation (CPUs) or rasterization (GPUs). Our experiments show that data movement is the main consumer of time and energy. As rays are traced deeper into the acceleration structure, more of the scene is accessed and must be loaded. This leads to the extensive use of the memory subsystem and DRAM, which dramatically increases the energy consumption of the whole system. Shadow ray traversal displays a similar behavior as regular ray traversal, although it is considerably less expensive, because it implements any-hit traversal optimization (as opposed to first hit). In all cases, we observe that the increase in per-ray, per-bounce energy is incremental after the first few bounces, suggesting that longer paths can be explored at a reduced proportional cost. We also examine the composition of latency per frame, identifying how much of the render time is spent on useful work versus stalling due to resource conflicts. Again, the memory system dominates the cost. Although compute time can often be improved through increases in available resources, the memory system, even when highly provisioned, may not be able to service all necessary requests without stalling.

2 Background

Some previous work focuses on understanding and improving the energy footprint of rendering on GPUs on both algorithmic and hardware levels. Yet, very little has been published on directly measuring the energy consumption and latency patterns of ray tracing and subsequently studying the implications of ray costs. In this section, we briefly discuss the related prior work and the TRaX architecture we use for our experiments.

2.1 Related work

Ray tracing performance is traditionally measured as a function of time to render a single frame. With a known upper bound on theoretical performance [2], general optimizations have been proposed to various stages of the algorithm [34] to improve performance and reduce memory traffic and data transport [5,15]. These approaches are motivated by known behavior, with bandwidth usage identified as the major bottleneck in traditional ray tracing [28,29], leading to suggested changes in ray and geometry scheduling. Although they address energy costs of ray tracing at a high level, none of those explorations examine how individual rays can affect performance, energy, and image quality, nor do they systematically analyze the performance of ray tracing as a whole. We provide a more quantifiable unit of measure for the underlying behavior by identifying the costs of rays as they relate to the entire frame generation. Aila et al. [2] evaluate the energy consumption of ray tracing on a GPU with different forms of traversal. Although the work distribution of ray traversal is identified as the major inefficiency, the analysis only goes so far as to suggest which traversal method is the quickest. Some work reduces energy consumption by minimizing the amount of data transferred from memory to compute units [3,11,31]. Others attempt to reduce memory accesses by improving ray coherence and data management [9,22,26]. More detailed studies on general rendering algorithms pinpoint power efficiency improvements [18,32], but unfortunately do not focus on ray tracing. Wang et al. [39] use a cost model to minimize power usage, while maintaining visual quality of the output image by varying rendering options in real-time frameworks. Similarly, Johnsson et al. [17] directly measure the per-frame energy of graphics applications on a smartphone. However, both methods focus on rasterization. There is a pool of work investigating architecture exploitation with much prior work addressing DRAM and its implications for graphics applications [8,38] with some particularly focusing on bandwidth [12,24,25]. Some proposed architectures also fall into a category of hardware which aims to reduce overall ray tracing energy cost by implementing packet-based approaches to increase cache hits [7,30] or by reordering work in a buffer [23]. Streaming architectures [14,37] and hardware that uses treelets to manage scene traffic [1,21,33] are also effective in reducing energy demands.

2.2 TRaX architecture

In our experiments, we use a hardware simulator to extract detailed information about time and energy consumption during rendering. We perform our experiment by simulating rendering on the TRaX architecture [35,36]. TRaX is a dedicated ray tracing hardware architecture based on a single program multiple data (SPMD) programming paradigm, as opposed to single instruction multiple data (SIMD) approach used by current GPUs. Unlike other ray tracing-specific architectures, TRaX's design is more general and programmable. Although it possesses similarities to modern GPU architectures, it is not burdened by the GPU's data processing assumptions.

Specifically, TRaX consists of thread multiprocessors (TMs), each of which has a number of thread processors (TPs), as shown in Fig. 1. Each TP contains some functional

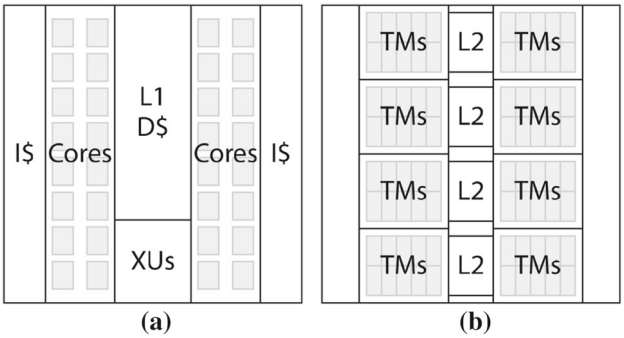

Fig. 1 Overall TRaX thread multiprocessor (TM) and multi-TM chip organization, from [36]. Abbreviations: I$—instruction cache, D$—data cache, and XU—execution unit. **a** TM architecture with 32 lightweight cores and shared cache and compute resources. **b** Potential TRaX chip organization with multiple TMs sharing L2 caches

units, a small register file, scratchpad memory, and a program counter. All TPs within a TM share access to units which are expensive in terms of area, like the L1 data cache and floating-point compute units. Several chip-wide L2 caches are each shared by a collection of TMs and are then connected to the main memory via the memory controller.

3 Experimental methodology

We run our experiments by simulating path tracing on the TRaX architecture. TRaX and its simulator are highly flexible systems, which enable testing modern architecture configurations. We have also considered other hardware simulators and decided against using them for various reasons. GPGPUSim [4] allows simulating GPUs, but only supports dated architectures and so would not provide an accurate representation of path tracing on modern hardware. Moreover, we need a system that is fast enough to run path tracing to completion, unlike other architecture simulators which are designed to feasibly simulate a few million cycles. Additionally, the profiling capabilities must separate parts of the renderer and generate detailed usage statistics for the memory system and compute, which is not easily attainable on regular CPUs. Although a comprehensive and configurable simulator for CPU architectures exists [6], it is far too detailed and thus expensive to run for the purposes of this study. As with any application, hardware dependency makes a difference in the performance evaluation. Therefore, we also run our experiments on a physical CPU, though the experiments on the CPU provide limited information, since we cannot gather statistics as detailed as those available from a cycle-accurate simulator. Yet, we can still compare the results of these tests to the simulated results and evaluate the generality of our conclusions. We augment the cycle-accurate simulator for TRaX [16] to profile each ray tracing kernel using high-fidelity statistics gathered at the instruction level. Each instruction tracks its execution time, stalls, and energy usage within hardware components, including functional units and the memory hierarchy. Additionally, the simulator relies on USIMM for high-fidelity DRAM simulation [10] enabling highly accurate measurements of main memory performance.

Table 1 Hardware configuration for the TRaX processor

	Details	Latency (cyc)
TM configuration		
TPs / TM	32	
Int Multiply	2	1
FP Add	8	2
FP Multiply	8	2
FP Divide	1	20
FP Inv Sqrt		
I. Cache	2× 4KB, 16 banks	1+
L1 Cache	1× 32KB, 8 banks	1+
Chip configuration		
Technology node	65nm CMOS	
Clock frequency	1GHz	
TMs	32, 1024 total threads	
L2 Cache	4× 512KB, 16 banks	3+
DRAM	4GB GDDR5, 8 channels	

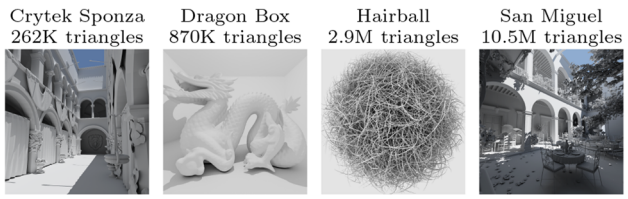

Fig. 2 Scenes used for all performance tests, arranged by their size in number of triangles

For our study, the TRaX processor comprises 32 TMs with 32 TPs each for a total of 1024 effective threads, all running at 1 GHz. This configuration resembles the performance and area of a modern GPU. Table 1 shows the energy and latency details for the hardware components. We use Cacti 6.5 [27] to estimate the areas of on-chip caches and SRAM buffers. The areas and latencies for compute units are estimated using circuits synthesized with Synopsys DesignWare/Design Compiler at a 65-nm process. The TMs share four 512KB L2 caches with 16 banks each. DRAM is set up to use 8-channel GDDR5 quad-pumped at twice the processor clock (8GHz effective) reaching a peak bandwidth of 512GB/s. We run our experiments on four scenes with different geometric complexities (Fig. 2) to expose the effects

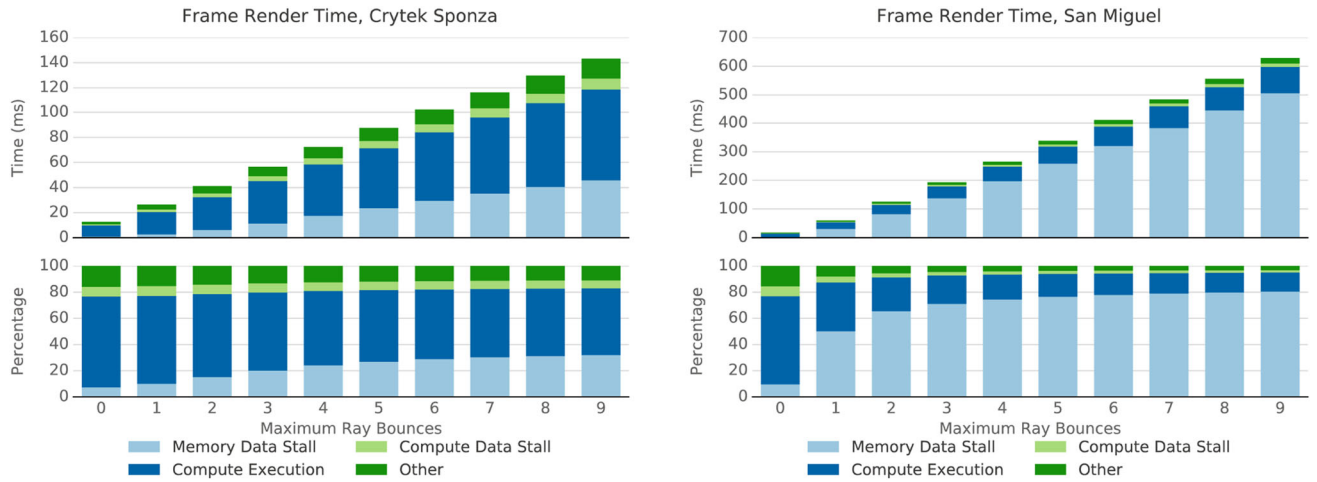

Fig. 3 Distribution of time spent between memory and compute for a single frame of the Crytek Sponza (left) and San Miguel (right) scenes rendered with different maximum ray bounces

of different computational requirements and stresses to the memory hierarchy. Each scene is rendered at 1024×1024 image resolution, with up to 9 ray bounces. Our investigation aims to focus on performance related to ray traversal and the underlying acceleration structure is a bounding volume hierarchy with optimized first child traversal [20]. We use simple Lambertian shaders for all surfaces and a single point light to light each scene. Individual pixels are rendered in parallel, where each TP independently traces a separate sample to completion; therefore, different TPs can trace rays at different bounces. We track detailed, instruction-level statistics for each distinct ray tracing kernel (ray generation, traversal, and shading) for each ray bounce and type (primary, secondary, and shadow). We derive energy and latency averages per ray using these data. We run our CPU tests on an Intel Core i7-5960X processor with 20 MB L3 cache and 8 cores (16 threads) with the same implementation of path tracing used by TRaX. Only the final rendering times are available for these experiments.

4 Experimental results

Our experimental results are derived from 50 simulations across four scenes with maximum ray bounces varying between 0 (no bounce) and 9. Depending on the complexity, each simulation can require from a few hours to a few days to complete. In this section we present some of our experimental results and the conclusions we draw based on them. The full set of experimental results are included in the supplementary document.

4.1 Render time

We first consider the time to render a frame at different maximum ray bounces and track how the render time is spent. In particular, we track the average time a TP spends on the following events:

- **Compute Execution:** the time spent executing instructions,
- **Compute Data Stall:** stalls from waiting for the results of previous instructions,
- **Memory Data Stall:** stalls from waiting for data from the memory hierarchy, including all caches and DRAM, and
- **Other:** all other stalls caused by contentions on execution units and local store operations.

Figure 3 shows the distribution of time used to render the Crytek Sponza and San Miguel scenes. In Crytek Sponza, the majority of the time is spent on computation without much memory data stalling. As the maximum number of ray bounces increases, the time for all components grows approximately proportionally, since the number of rays (and thus computational requirements) increases linearly with each bounce. This is not surprising, since Crytek Sponza is a relatively small scene and most of it can fit in the L2 cache, thereby requiring relatively fewer accesses to DRAM. Once all scene data are read into the L2 cache, the majority of memory data stalls are caused by L1 cache misses. On the other hand, in the San Miguel scene, compute execution makes up the majority of the render time only for primary rays. When we have one or more ray bounces, memory data stalls quickly become the main consumer of render time, consistently taking up approximately 65% of the total time. Even though the instructions needed to handle secondary rays are comparable to the ones for the primary rays, the L1 cache hit rate drops from approximately 80% for primary rays to 60% for rays with up to two bounces or more. As a result, more memory requests escalate up the memory hierarchy to DRAM,

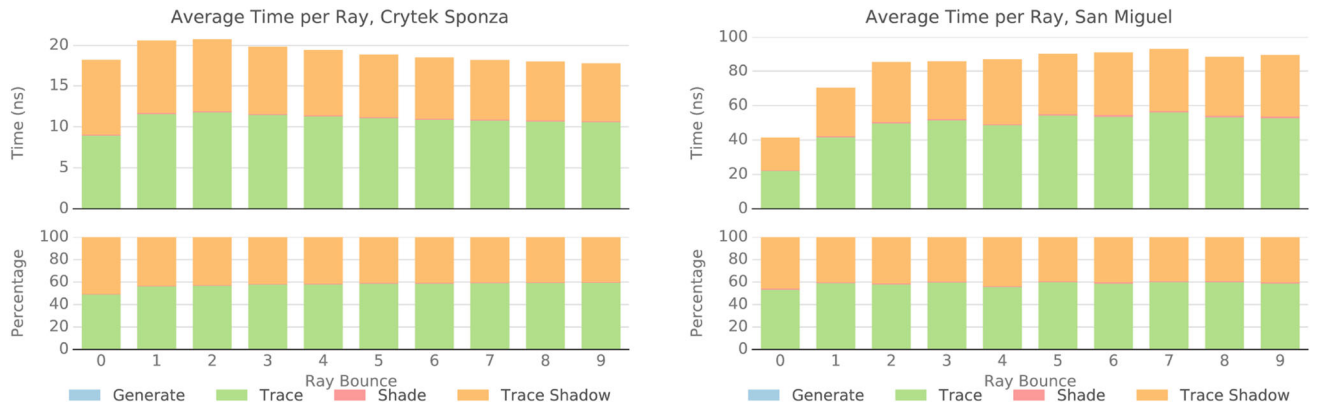

Fig. 4 Classification of time per kernel normalized by the number of rays. Crytek Sponza (left) and San Miguel (right) scenes rendered with a maximum of 9 ray bounces. Contributions from the Generate and Shade kernels are not visible, because they are negligible compared to others

putting yet more pressure on the memory banks. Besides adding latency, cache misses also incur higher energy costs.

4.2 Time per kernel

We can consider the average time spent per ray by the following individual kernels at different ray bounces:

- **Generate:** ray generation kernel,
- **Trace:** ray traversal kernel for non-shadow rays, including the acceleration structure and triangle intersections,
- **Trace Shadow:** shadow ray traversal kernel, and
- **Shade:** shading kernel.

Figure 4 shows the average computation time per ray for each bounce of path tracing up to 9 bounces. The time consumed by the ray generation and shading kernels is negligible. This is not surprising, since ray generation does not require accessing the scene data and the Lambertian shader we use for all surfaces does not use textures. Even though these two kernels are compute-intensive, the tested hardware is not compute-limited, and thus the execution units take a smaller portion of the total frame rendering time. Traversing regular rays (the Trace kernel) takes up most of the time, and traversing shadow rays (the Trace Shadow kernel) is about 20% faster for all bounces.

4.3 Ray traversal kernel time

Within the ray traversal (Trace) kernel, a large portion of time is spent stalling while waiting for the memory system–either for data to be fetched or on bank conflicts which limit access requests to the memory. Figure 5 shows the breakdown of time spent for execution and stalls within the Trace kernel for handling rays at different bounces within the same rendering process up to 9 bounces. Memory access stalls, which indicate the time required for data to be fetched into registers, take a substantial percentage of time even for the first few bounces. The percentage of memory stalls is higher for larger scenes, but they amount to a sizable percentage even for a relatively small scene like Crytek Sponza. Interestingly, the percentage of memory stalls beyond the second bounce remains almost constant. This is because rays access the scene less coherently, thereby thrashing the caches. This is a significant observation, since the simulated memory system is highly provisioned in terms of both the number of banks and total storage size. This suggests that further performance improvements gained will be marginal if only simple increases in resources are made. Thus, we foresee the need to require modifications in how the memory system is structured and used.

4.4 DRAM bandwidth

Another interesting observation is the DRAM bandwidth behavior. Figure 6 shows the DRAM bandwidth for all four scenes in our tests using different maximum ray bounces. Notice that the DRAM bandwidth varies significantly between different scenes for images rendered using a few number of maximum bounces.

In our tests our smallest scene, Crytek Sponza, and largest scene, San Miguel, use a relatively small portion of the DRAM bandwidth for different reasons. Crytek Sponza uses less DRAM bandwidth, simply because it is a small scene. San Miguel, however, uses lower DRAM bandwidth because of the coherence of the first few bounces and the fact that it takes longer to render. The other two scenes, Hairball and Dragon Box, use a relatively larger portion of the DRAM bandwidth for renders up to a few bounces.

Beyond a few bounces, however, the DRAM bandwidth utilization of these four scenes tends to align with the scene sizes. Small scenes that render quickly end up using larger bandwidth, and larger scenes that require a longer time use a smaller portion of the DRAM bandwidth by spreading the

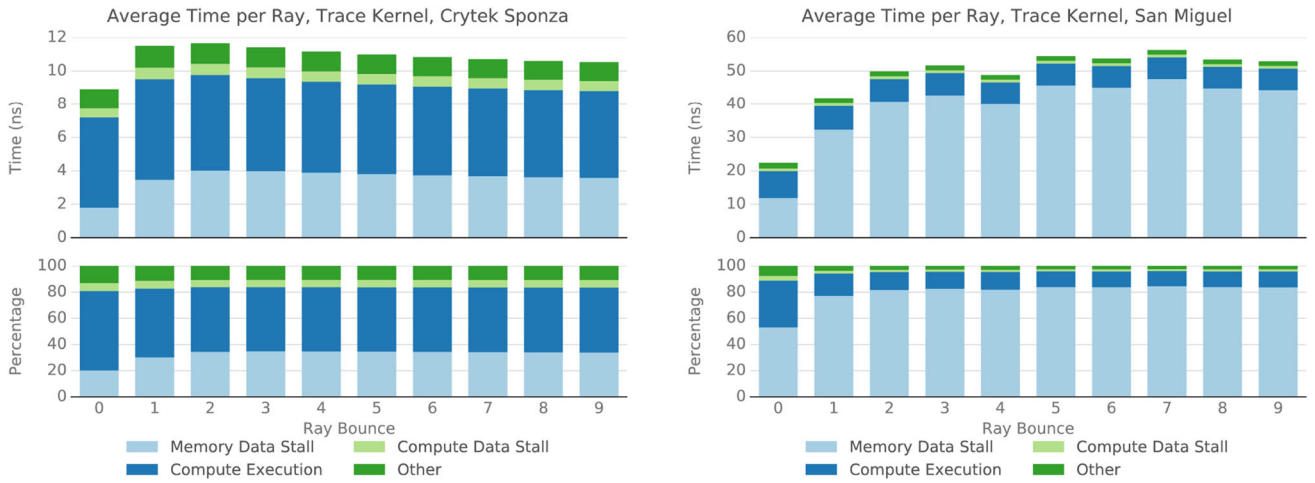

Fig. 5 Classification of time per ray spent between memory and compute for the Trace kernel. Crytek Sponza (left) and San Miguel (right) rendered with maximum of 9 ray bounces

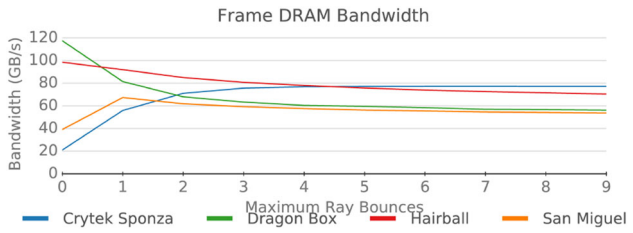

Fig. 6 DRAM bandwidth used to render each scene to different maximum ray bounces

memory requests over time. Yet, all scenes appear to converge toward a similar DRAM bandwidth utilization.

4.5 Energy use

The energy used to render the entire frame can be separated into seven distinct sources: compute, register file, local store, instruction cache, L1 data cache, L2 data cache, and DRAM. Overall, performing a floating-point arithmetic operation is both faster and three orders of magnitude less energy-expensive than fetching an operand from DRAM [13]. Figure 7 shows the total energy spent to render a frame of the Crytek Sponza and San Miguel scenes. In Crytek Sponza, a small scene which mostly fits within on-chip data caches, memory accesses still dominate the energy contributions at 80% overall, including 60% for DRAM alone, at 9 bounces. Compute, on the other hand, requires only about 1–2% of the total energy. Interestingly, a larger scene like San Miguel follows a similar behavior: The entire memory subsystem requires 95% and DRAM requires 80% of the total energy per frame at the maximum of 9 bounces. The monotonic increase in the total frame energy at higher maximum ray bounces can be attributed to the increase in the total number of rays in the system.

4.6 Energy use per kernel

We can consider energy per ray used by individual kernels at different ray bounces by investigating the average energy spent to execute the assigned kernels. Figure 8 shows the average energy use in the Crytek Sponza and San Miguel scenes. The ray generation kernel has a very small contribution (at most 2%) because it uses few instructions, mainly for floating-point computation operations. In our tests, shading also consumes a small percentage of energy, simply because we use simple Lambertian materials without textures. Other material models, especially ones that use large textures, could be substantially expensive from the energy perspective because of memory accesses. However, investigating a broad range of shading methods is beyond the scope of this work. Focusing on the traversal kernels, Fig. 9 shows the comparison between the costs to trace both shadow and non-shadow rays for all scenes. Overall, because shadow rays implement any-hit traversal optimization and consequently load less scene data, their energy cost is 15% lower than regular rays on average.

4.7 Ray traversal kernel energy

Figure 9 also shows the total energy cost of the ray traversal kernels at different ray bounces up to the maximum of 9. Unsurprisingly, the larger scenes and those with high depth complexity consume more energy as ray bounces increase. The energy required by rays before the first bounce is considerably lower than the secondary rays after the first bounce, since they are less coherent than primary rays and scatter toward a larger portion of the scene. This behavior translates into an increase both in the randomness of memory accesses and in the amount of data fetched. However, as the rays

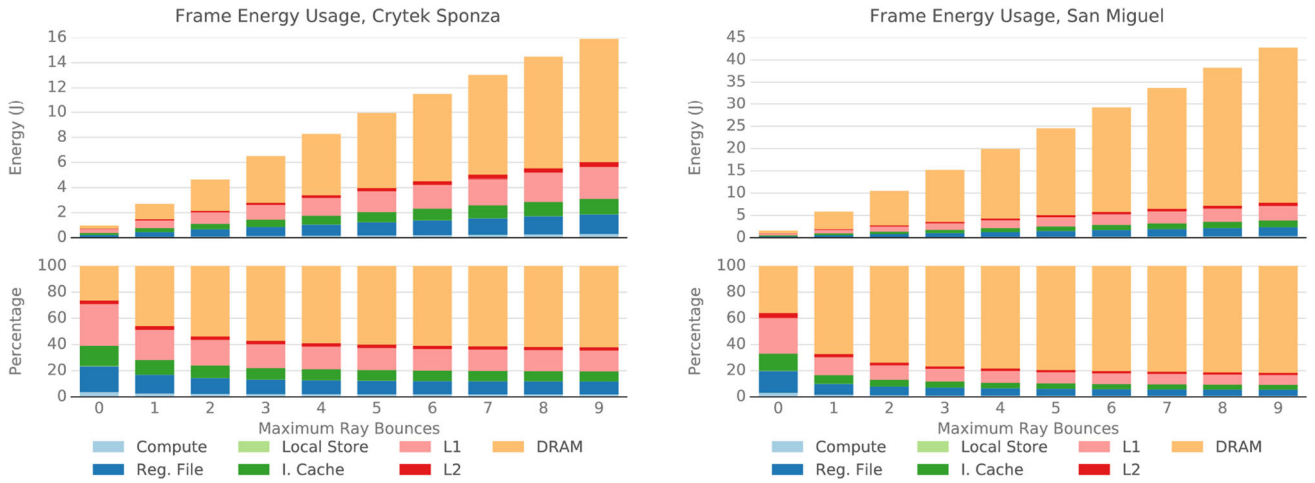

Fig. 7 Classification of energy contributions by source for a single frame of the Crytek Sponza (left) and San Miguel (right) scenes rendered with different maximum ray bounces

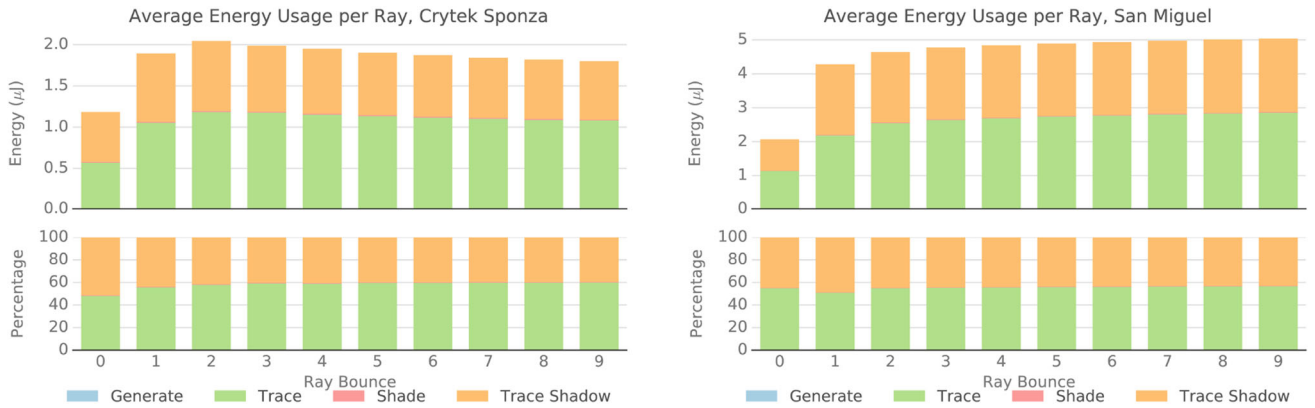

Fig. 8 Energy classification per kernel normalized by the number of rays. Crytek Sponza (left) and San Miguel (right) scenes rendered with maximum of 9 ray bounces. Contributions from the Generate and Shade kernels are not visible, because they are negligible compared to others

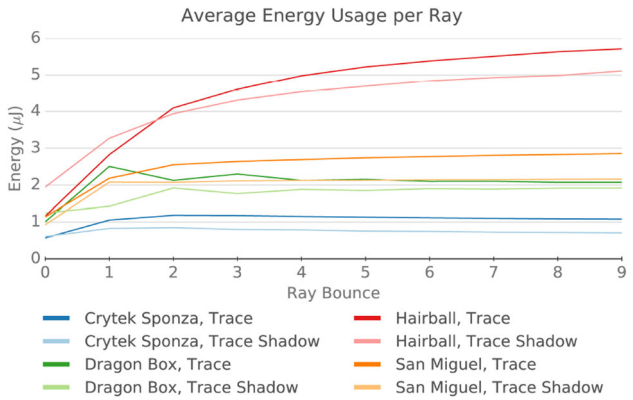

Fig. 9 Comparison of the energy contributions per ray for the Trace kernels (non- and shadow rays) for all scenes

bounce further, the cost per ray starts to level off. This pattern is more obvious for smaller scenes like Crytek Sponza. Although in the first few ray bounces the path tracer thrashes the caches and the cache hit rates drop, the hit rates become roughly constant for additional bounces. Thus, the number of requests that reach DRAM remains steady, resulting in the energy used by the memory system to be fairly consistent for ray bounces beyond three. The sources of energy usage per ray for the traversal kernels (Fig. 10) paint a picture similar to the one from the overall energy per frame. The memory system is responsible for 60–95% of the total energy, with DRAM alone taking up to 80% for higher bounces in the San Miguel scene.

4.8 Image contributions per ray bounce

It is also important to understand how much each bounce is contributing to the final image. This information can be utilized to determine a desired performance/quality balance. In particular, we perform tests with maximum ray bounces of 9 and we consider the overall image intensity contributions of all rays up to a certain number of bounces (maximum of 9),

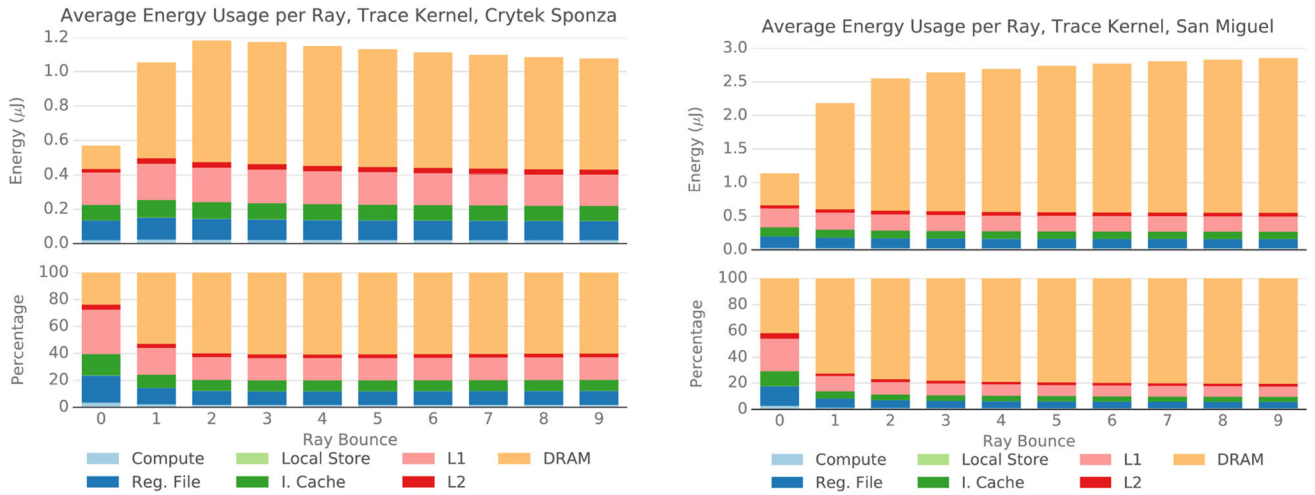

Fig. 10 Classification of energy contributions per ray by source for the Trace kernel. The Crytek Sponza (left) and San Miguel (right) scenes rendered with maximum of 9 ray bounces

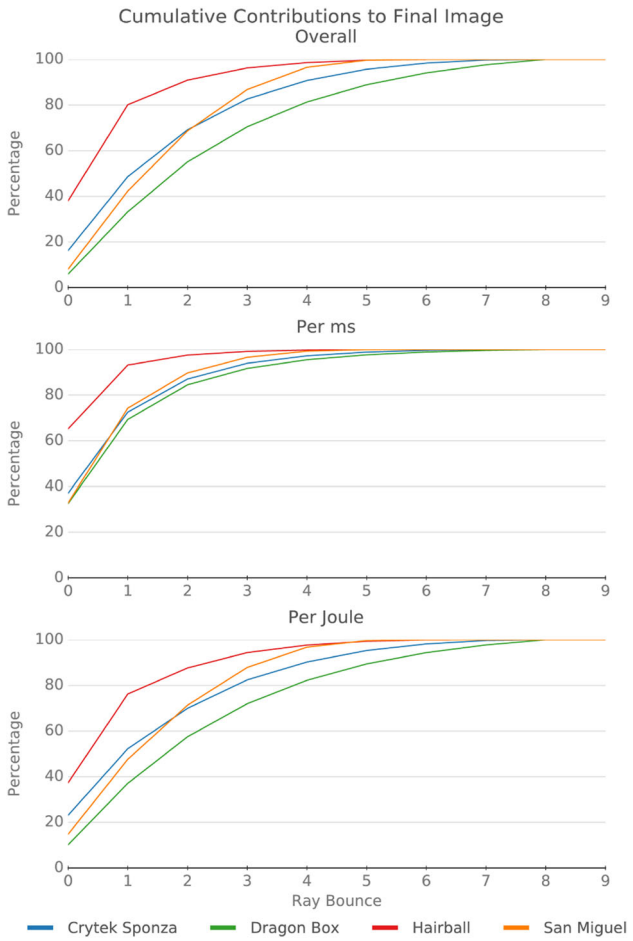

Fig. 11 Cumulative distribution of the percentage contribution to the final image using different metrics. All scenes rendered with maximum of 9 ray bounces

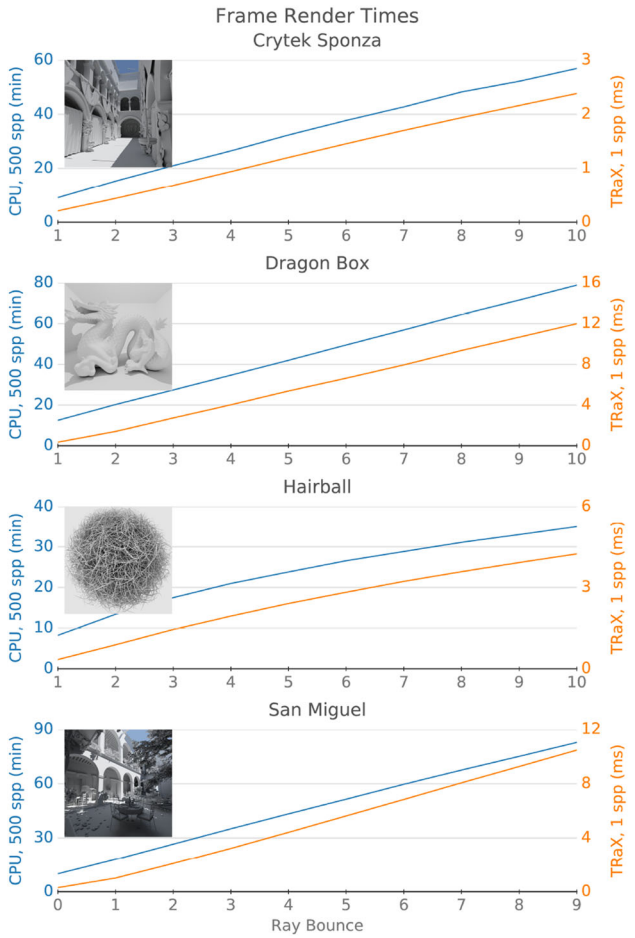

Fig. 12 Frame render times up to the maximum of 9 bounces. The CPU implementation uses 500 samples per pixel (spp), while TRaX uses 1

along with contributions per millisecond and contributions per Joule. As shown in Fig. 11, the majority of contribution to the image happens in the first few bounces. After the fourth or fifth bounce, the energy and latency costs to trace rays at that bounce become significant compared to their minimal contribution to the final image. This behavior is expected from the current analysis with scene complexity playing a minor role to the overall trend.

4.9 Comparisons to CPU experiments

The findings so far are specific to the TRaX architecture. To evaluate the generality of our findings, we also compare our results to the same path tracing application running on a CPU. For the four test scenes, we observe similar relative behavior shown in Fig. 12. Even though direct comparisons cannot be made, the behavior is similar enough to suggest that performance would be similar between the two architectures; therefore, the results of this study could be applied on implementations running on currently available CPUs.

5 Discussion

We observe that even for smaller scenes, that can essentially fit into cache, memory still is the highest contributor in energy and latency, suggesting that even in a case of balanced compute workload, compute remains inexpensive. Since data movement is the highest contributor to energy use, often scene compression is the suggested solution. However, compression schemes mostly reduce, but do not eliminate, the memory bottlenecks arising from data requests associated with ray tracing. Our data suggest that render time and energy cost improvements cannot be made by simply increasing the available memory resources, which are already constrained by the on-chip area availability.

This brings up an interesting opportunity to find ways to design a new memory system that is optimized for ray tracing that would facilitate both lower energy and latency costs. For example, the recent dual streaming approach [33] that reorders the ray tracing computations and the memory access pattern is likely to have a somewhat different time and energy behavior. Exploring different ways of reordering the ray tracing execution would be an interesting avenue for future research, which can provide new algorithms and hardware architectures that can possibly separate from the trends we observe in our experiments.

6 Conclusions and future work

We have presented a detailed study of render time and energy costs of path tracing running on a custom hardware designed for accelerating ray tracing. We have identified the memory system as the main source of both time and energy consumption. We have also examined how statistics gathered per frame translate into contributions to the final image. Furthermore, we have included an evaluation of the generality of our results by comparing render times against the same application running on the CPU. Given these observations, we would like to consider more holistic performance optimizations as a function of render time, energy cost and the impact of rays on image quality.

An interesting future work direction would be a sensitivity analysis by varying the hardware specifications, such as the memory subsystem size. Also, a study targeting more expensive shading models and texture contributions could reveal how shading complexity could impact ray traversal performance. In general, detailed studies of ray tracing performance can provide much needed insight that can be used to design a function that optimizes both render time and energy under constrained budgets and a required visual fidelity.

Acknowledgements Crytek Sponza is from Frank Meinl at Crytek and Marko Dabrovic, Dragon is from the Stanford Computer Graphics Laboratory, Hairball is from Samuli Laine, and San Miguel is from Guillermo Leal Laguno.

Funding This material is supported in part by the National Science Foundation under Grant No. 1409129.

Compliance with Ethical Standards

Conflict of interest The authors, Elena Vasiou, Konstantin Shkurko, Ian Mallett, Erik Brunvand, and Cem Yuksel, declare that they have no conflict of interest relating to this work and publication.

References

1. Aila, T., Karras, T.: Architecture considerations for tracing incoherent rays. In: Proceedings of HPG (2010)
2. Aila, T., Laine, S.: Understanding the efficiency of ray traversal on GPUs. In: Proceedings of HPG (2009)
3. Arnau, J.M., Parcerisa, J.M., Xekalakis, P.: Eliminating redundant fragment shader executions on a mobile GPU via hardware memoization. In: Proceedings of ISCA (2014)
4. Bakhoda, A., Yuan, G.L., Fung, W.W.L., Wong, H., Aamodt, T.M.: Analyzing CUDA workloads using a detailed GPU simulator. In: ISPASS (2009)
5. Barringer, R., Akenine-Möller, T.: Dynamic ray stream traversal. ACM TOG **33**(4), 33 (2014)
6. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., et al.: The gem5 simulator. ACM SIGARCH Comp Arch News **39**(2), 1–7 (2011)
7. Boulos, S., Edwards, D., Lacewell, J.D., Kniss, J., Kautz, J., Shirley, P., Wald, I.: Packet-based Whitted and distribution ray tracing. In: Proceedings of Graphics Interface (2007)
8. Brunvand, E., Kopta, D., Chatterjee, N.: Why graphics programmers need to know about DRAM. In: ACM SIGGRAPH 2014 Courses (2014)

9. Budge, B., Bernardin, T., Stuart, J.A., Sengupta, S., Joy, K.I., Owens, J.D.: Out-of-core Data Management for Path Tracing on Hybrid Resources. CGF (2009)
10. Chatterjee, N., Balasubramonian, R., Shevgoor, M., Pugsley, S., Udipi, A., Shafiee, A., Sudan, K., Awasthi, M., Chishti, Z.: USIMM: the Utah SImulated Memory Module. Technical Report UUCS-12-02, U. of Utah (2012)
11. Chatterjee, N., OConnor, M., Lee, D., Johnson, D.R., Keckler, S.W., Rhu, M., Dally, W.J.: Architecting an energy-efficient DRAM system for GPUs. In: HPCA (2017)
12. Christensen, P.H., Laur, D.M., Fong, J., Wooten, W.L., Batali, D.: Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. In: Eurographics (2003)
13. Dally, B.: The challenge of future high-performance computing. Uppsala University, Uppsala, Sweden, Celsius Lecture (2013)
14. Gribble, C., Ramani, K.: Coherent ray tracing via stream filtering. In: IRT (2008)
15. Hapala, M., Davidovic, T., Wald, I., Havran, V., Slusallek, P.: Efficient stack-less BVH traversal for ray tracing. In: SCCG (2011)
16. HWRT: SimTRaX a cycle-accurate ray tracing architectural simulator and compiler. http://code.google.com/p/simtrax/ (2012). Utah Hardware Ray Tracing Group
17. Johnsson, B., Akenine-Mller, T.: Measuring per-frame energy consumption of real-time graphics applications. JCGT **3**, 60–73 (2014)
18. Johnsson, B., Ganestam, P., Doggett, M., Akenine-Möller, T.: Power efficiency for software algorithms running on graphics processors. In: HPG (2012)
19. Kajiya, J.T.: The rendering equation. In: Proceedings of SIGGRAPH (1986)
20. Karras, T., Aila, T.: Fast parallel construction of high-quality bounding, vol. hierarchies. In: Proceedings of HPG (2013)
21. Kopta, D., Shkurko, K., Spjut, J., Brunvand, E., Davis, A.: An energy and bandwidth efficient ray tracing architecture. In: Proceedings of HPG (2013)
22. Kopta, D., Shkurko, K., Spjut, J., Brunvand, E., Davis, A.: Memory considerations for low energy ray tracing. CGF **34**(1), 47–59 (2015)
23. Lee, W.J., Shin, Y., Hwang, S.J., Kang, S., Yoo, J.J., Ryu, S.: Reorder buffer: an energy-efficient multithreading architecture for hardware MIMD ray traversal. In: Proceedings of HPG (2015)
24. Liktor, G., Vaidyanathan, K.: Bandwidth-efficient BVH layout for incremental hardware traversal. In: Proceedings of HPG (2016)
25. Mansson, E., Munkberg, J., Akenine-Moller, T.: Deep coherent ray tracing. In: IRT (2007)
26. Moon, B., Byun, Y., Kim, T.J., Claudio, P., Kim, H.S., Ban, Y.J., Nam, S.W., Yoon, S.E.: Cache-oblivious ray reordering. ACM Trans. Graph. **29**(3), 28:1–28:10 (2010)
27. Muralimanohar, N., Balasubramonian, R., Jouppi, N.: Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In: MICRO (2007)
28. Navrátil, P., Fussell, D., Lin, C., Mark, W.: Dynamic ray scheduling to improve ray coherence and bandwidth utilization. In: IRT (2007)
29. Navrátil, P.A., Mark, W.R.: An analysis of ray tracing bandwidth consumption. University of Texas at Austin, Computer Science Department (2006)
30. Overbeck, R., Ramamoorthi, R., Mark, W.R.: Large ray packets for real-time Whitted ray tracing. In: IRT (2008)
31. Pool, J.: Energy-precision tradeoffs in the graphics pipeline. Ph.D. thesis, UNC, Chapel Hill (2012)
32. Pool, J., Lastra, A., Singh, M.: An energy model for graphics processing units. In: ICCD (2010)
33. Shkurko, K., Grant, T., Kopta, D., Mallett, I., Yuksel, C., Brunvand, E.: Dual streaming for hardware-accelerated ray tracing. In: Proceedings of HPG (2017)
34. Smits, B.: Efficiency issues for ray tracing. In: SIGGRAPH Courses, SIGGRAPH '05 (2005)
35. Spjut, J., Kensler, A., Kopta, D., Brunvand, E.: TRaX: A multicore hardware architecture for real-time ray tracing. IEEE Trans. CAD **28**(12), 1802–1815 (2009)
36. Spjut, J., Kopta, D., Boulos, S., Kellis, S., Brunvand, E.: TRaX: A multi-threaded architecture for real-time ray tracing. In: SASP (2008)
37. Tsakok, J.A.: Faster incoherent rays: Multi-BVH ray stream tracing. In: Proceedings of HPG (2009)
38. Vogelsang, T.: Understanding the energy consumption of dynamic random access memories. In: MICRO '43 (2010)
39. Wang, R., Yu, B., Marco, J., Hu, T., Gutierrez, D., Bao, H.: Real-time rendering on a power budget. ACM TOG **35**(4), 111:1–111:11 (2016)
40. Whitted, T.: An improved illumination model for shaded display. Com. ACM **23**(6), 343–349 (1980)

Elena Vasiou received BS degree in mathematics from Davidson College in 2015. She is currently working toward a Ph.D. in computer graphics at the School of Computing at the University of Utah. Her research focuses on ray tracing hardware as well as rendering algorithms, real-time graphics, and energy-efficient graphics accelerators.

Konstantin Shkurko received the BA in mathematics and physics and an MS in computer graphics from Cornell University, in 2007 and 2010, respectively. He is currently working toward the Ph.D. in computer graphics from the School of Computing at the University of Utah. His research focuses mainly on ray tracing hardware, but also includes acceleration structures, rendering algorithms, and scientific visualization.

Ian Mallett received BS degrees in computer science and pure mathematics from the University of New Mexico in 2014. He is currently working toward a Ph.D. in computer graphics at the School of Computing, University of Utah. His research focuses on rendering algorithms and light transport, with excursions to ray tracing hardware.

Erik Brunvand received his Ph.D. from Carnegie Mellon University in 1990. Since then he has been a faculty member in the School of Computing at the University of Utah where his interests include the design of application-specific computers, graphics processors, physical computing, asynchronous systems, VLSI integrated circuit design, and arts/technology collaboration and integration in both research and education.

Cem Yuksel is a faculty member in the School of Computing at the University of Utah. Previously, he was a postdoctoral fellow at Cornell University, after receiving his Ph.D. in Computer Science from Texas A&M University in 2010. His research interests are in computer graphics and related fields, including physically based simulations, rendering techniques, global illumination, sampling, GPU algorithms, graphics hardware, knitted structures, and hair modeling, animation, and rendering.

Terms and Conditions

Springer Nature journal content, brought to you courtesy of Springer Nature Customer Service Center GmbH (“Springer Nature”).
Springer Nature supports a reasonable amount of sharing of research papers by authors, subscribers and authorised users (“Users”), for small-scale personal, non-commercial use provided that all copyright, trade and service marks and other proprietary notices are maintained. By accessing, sharing, receiving or otherwise using the Springer Nature journal content you agree to these terms of use (“Terms”). For these purposes, Springer Nature considers academic use (by researchers and students) to be non-commercial.
These Terms are supplementary and will apply in addition to any applicable website terms and conditions, a relevant site licence or a personal subscription. These Terms will prevail over any conflict or ambiguity with regards to the relevant terms, a site licence or a personal subscription (to the extent of the conflict or ambiguity only). For Creative Commons-licensed articles, the terms of the Creative Commons license used will apply.
We collect and use personal data to provide access to the Springer Nature journal content. We may also use these personal data internally within ResearchGate and Springer Nature and as agreed share it, in an anonymised way, for purposes of tracking, analysis and reporting. We will not otherwise disclose your personal data outside the ResearchGate or the Springer Nature group of companies unless we have your permission as detailed in the Privacy Policy.
While Users may use the Springer Nature journal content for small scale, personal non-commercial use, it is important to note that Users may not:

1. use such content for the purpose of providing other users with access on a regular or large scale basis or as a means to circumvent access control;
2. use such content where to do so would be considered a criminal or statutory offence in any jurisdiction, or gives rise to civil liability, or is otherwise unlawful;
3. falsely or misleadingly imply or suggest endorsement, approval , sponsorship, or association unless explicitly agreed to by Springer Nature in writing;
4. use bots or other automated methods to access the content or redirect messages
5. override any security feature or exclusionary protocol; or
6. share the content in order to create substitute for Springer Nature products or services or a systematic database of Springer Nature journal content.

In line with the restriction against commercial use, Springer Nature does not permit the creation of a product or service that creates revenue, royalties, rent or income from our content or its inclusion as part of a paid for service or for other commercial gain. Springer Nature journal content cannot be used for inter-library loans and librarians may not upload Springer Nature journal content on a large scale into their, or any other, institutional repository.
These terms of use are reviewed regularly and may be amended at any time. Springer Nature is not obligated to publish any information or content on this website and may remove it or features or functionality at our sole discretion, at any time with or without notice. Springer Nature may revoke this licence to you at any time and remove access to any copies of the Springer Nature journal content which have been saved.
To the fullest extent permitted by law, Springer Nature makes no warranties, representations or guarantees to Users, either express or implied with respect to the Springer nature journal content and all parties disclaim and waive any implied warranties or warranties imposed by law, including merchantability or fitness for any particular purpose.
Please note that these rights do not automatically extend to content, data or other material published by Springer Nature that may be licensed from third parties.
If you would like to use or distribute our Springer Nature journal content to a wider audience or on a regular basis or in any other manner not expressly permitted by these Terms, please contact Springer Nature at

onlineservice@springernature.com