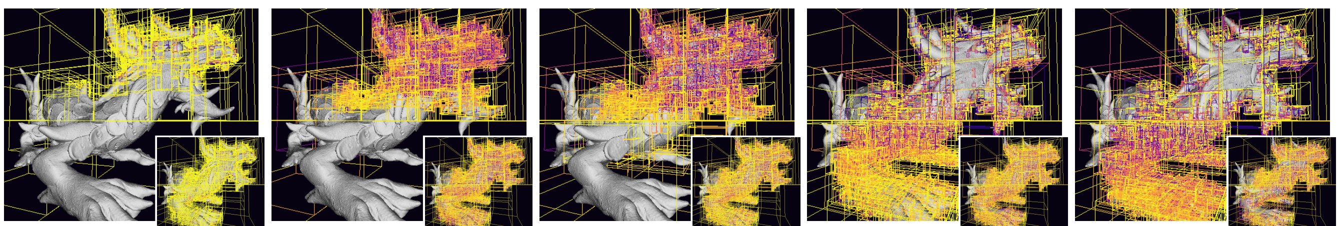
Eurographics Symposium on Parallel Graphics and Visualization (2022)
R. Bujack, J. Tierny, F. Sadlo (Editors)

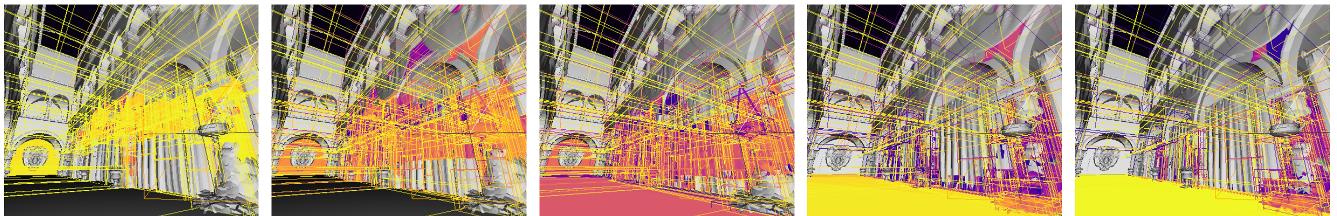
Profiling and Visualizing GPU Memory Access and Cache Behavior of Ray Tracers

Max von Buelow[1], Kai Riemann[2], Stefan Guthe[1,3] and Dieter W. Fellner[1,3,4]

[1]Technical University of Darmstadt, Germany
[2]Independent Researcher, Germany
[3]Fraunhofer IGD, Germany
[4]Graz University of Technology, Institute of Computer Graphics and Knowledge Visualization, Austria



(a) *Time series visualization for the Asian Dragon mesh using two different schedulers.*



(b) *Time series visualization for the Sponza mesh.*

Figure 1: *Frames from the scanline progression over time of the Asian Dragon (a) and Sponza (b) mesh with colors representing the cache state. Faces that are likely to produce cache hits within a frame are marked in yellow. Conversely, red is used to mark faces that are more likely to produce cache misses as they are being accessed in memory. Only five out of eight frames are shown here due to space limitations. The big frames show a global scanline progression, while the inset frames demonstrates how scheduling behaves when each SM has a disjoint scanline region distributed over the image.*

Abstract

Graphical processing units (GPUs) have gained popularity in recent years due to their efficiency in running massively parallel applications. Recent developments have also adapted ray-tracing algorithms to the GPU, where the bottleneck in the overall performance is usually given by the memory bandwidth. In this paper, we present an interactive, web-based visualization tool for GPU memory traces that provides visual insight into the memory and cache behavior of our reference ray tracer, by mapping internal GPU state back onto 3D objects. In order to visualize cache behavior, we use reuse distances on both GPU cache layers that are calculated on the basis of memory traces extracted from a real GPU using binary instrumentation. An advantage of our system is that it runs independently of the ray-tracing program. We further show visualizations of our GPU ray tracer and compare the visualizations of several ray-tracing approaches. We find our work to act as a convenient toolset to gather insights on which data structures and mesh regions can be cached efficiently, and how ray-tracing acceleration structures behave on various input meshes, bounding volume hierarchies, memory layouts, frame buffer resolutions, and work distribution techniques.

CCS Concepts

• ***Human-centered computing*** → ***Visual analytics;*** • ***Computing methodologies*** → *Graphics processors;* • ***Theory of computation*** → *Program analysis;*

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

delivered by EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

DOI: 10.2312/pgv.20221061

1. Introduction

The run-time performance of ray-tracing algorithms primarily depends on the number of rays and—when using hierarchical acceleration structures—on the logarithmic size of the scene. More realistic rendering requires an increase in either quantity. Recent developments in the field of GPGPU allow for efficient execution of ray-tracing algorithms on graphics cards that are nowadays specialized to run such computations, making ray tracing available for real-time use in computer games [PBD*10].

The main bottlenecks in ray-tracing applications are the memory latency and work-distribution [AL09]. Targeting the former, GPU vendors implemented a cache hierarchy similar to those of CPUs. Given these caches, the programmer is now faced with the challenge of keeping the structure of the cache in mind in order to make programs more efficient. Simultaneously, GPU vendors are continuously trying to optimize their caches given a set of well-known applications [JBC*15]. Due to SIMD parallelism and separation across different processors on the GPU, these cache hierarchies have become quite complex.

Therefore, we developed a two-part toolset providing visual insight into cache and memory behavior. The first tool is used to extract a list of memory accesses from the ray-tracing binary during execution by injecting assembly instructions into the run-time. Additionally, this tool simulates the cache behavior, transforming memory accesses into per-element accesses of scene elements, and lastly writes all values to a file. The second tool is a web-based dashboard that parses and visualizes said files, giving the user control over how temporal data should be displayed. The dashboard consists of a mesh viewer that shows the original mesh along with a color for each primitive representing various metrics, such as the current L1 cache state.

In summary, our contributions are:

- A toolset capable of automatically extracting memory profiles from ray-tracing programs during run-time and interactively visualizing this data in a high-performance web-based dashboard.
- A method to simulate fine-grained hardware cache rates and visualizing them onto the original application data (the mesh and BVH) using a color encoding.
- A toolset that may be used for profiling and teaching purposes to simultaneously show how caches behave on GPUs for particular memory permutations, meshes and bounding volume hierarchies (BVHs), and how they influence which parts of the mesh actually get referenced.
- A visualization of the write order with respect to the frame buffer, allowing insight into scheduling and simultaneously indicating the actual application behavior.

2. Related Work

The performance of ray tracers depends on a variety of different factors. VASIOU, SHKURKO, MALLETT, et al. [VSM*18] show that energy consumption and computational time of ray-tracing applications depend on the amount of data movement from DRAM and caches as well as memory stalls. First, we address memory and cache accesses optimizations by more efficient hardware work distribution techniques. AILA and LAINE [AL09] present such optimizations by comparing ray-tracing performance of several hardware work distribution implementations, while PHARR, KOLB, GERSHBEIN, and HANRAHAN [PKGH97] show that influencing the scheduling given the state of the cache can also improve rendering performance on CPU architectures. Similar to L1 caches on GPUs, DEMARLE, GRIBBLE, and PARKER [DGP04] introduce a distributed shared memory model for ray tracers and ensure that work is balanced across computing nodes in a way that maximizes hit rates. Besides work distribution, bounding volume hierarchies enhance ray-tracing performance by keeping triangle accesses and intersection rates small. MACDONALD and BOOTH [MB90] introduce the surface-area heuristic that optimizes BVHs, and AILA, KARRAS, and LAINE [AKL13] define further quality metrics for BVHs. Different BVH memory layouts have been evaluated by WODNIOK, SCHULZ, WIDMER, and GOESELE [WSWG13] in terms of cache performance, while WALD, MORRICAL, and ZELLMANN [WMZ22] further enhance memory access rates as well as caching by reducing the footprint of BVH leaves and faces, using simple compression techniques.

In order to visually profile a ray tracer, we rely on reuse distances of memory traces. Reuse distances are a well-researched performance metric that have been shown to be an accurate locality estimation for CPU architectures by DING and ZHONG [DZ03]. The Stack Distance Cache Model (SDCM) of AGARWAL, HENNESSY, and HOROWITZ [AHH89] uses reuse distances in order to estimate the whole-program hit rate. ARAFA, CHENNUPATI, BARAI, et al. [ACB*19] apply this technique successfully to GPU architectures. Based on this, ARAFA, BADAWY, CHENNUPATI, et al. [ABC*20] use the *NVIDIA Binary Instrumentation Tool* (NVBIT) [VSNK19] in order to extract required memory traces during the execution of arbitrary programs.

ISAACS, GIMÉNEZ, JUSUFI, et al. [IGJ*14] give an extensive overview of state-of-the-art performance visualization tools that rely on extracted performance profiles. Most importantly, they mention the *HAC* model by SCHULZ, LEVINE, BREMER, et al. [SLB*11] which categorizes performance data into the hardware, application, and communication domains and explains how to map between them. As for the hardware domain, memory access visualization tools such as *VAMPIR* [NAW*96] and *TAU* [SM06] use classical visualization techniques like diagrams, charts, timelines, and matrix plots in order to illustrate memory behavior. The cache visualization tool by van der DEIJL, KANBIER, TEMAM, and GRANSTON [vdDKTG97], on the other hand, is more centered around memory layout and visualizes memory addresses as cells in a 2D grid. Similarly, ROSEN [Ros13] creates visualizations of GPU memory on the grid, block, warp, and operation level. YU, BEYLS, and D'HOLLANDER [YBD01] visualize cache hits and reuse distances as a locality metric in a similar 2D grid. The abstract visualization by CHOUDHURY and ROSEN [CR11] implements a circular layout for CPU profiles that visualizes spatial and temporal locality of the memory using shaded points whose distance to the center is smaller in higher-performance memory levels. In the application domain, bare memory accesses of ray-tracing applications are usually visualized as heat maps on bounding volumes and polygons [YLZ*16] not involving further hardware-side metrics. Recent developments use profiling data from binary instrumentation in order

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

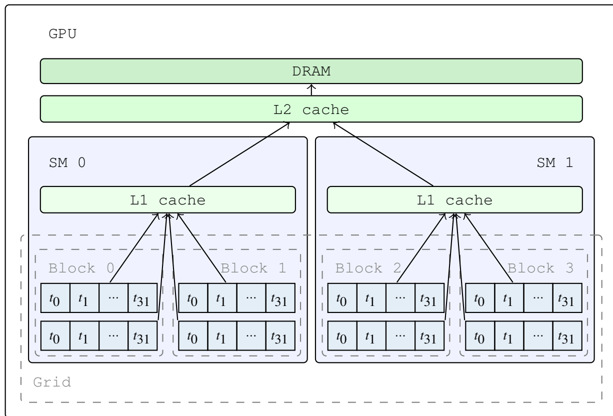


Figure 2: *A sketch of most NVIDIA architectures. Hardware components are marked with solid black borders, and memory (green) communication with arrows. Execution units are marked in blue. Software-side concepts are indicated by dashed gray borders. Each row of* 32 *threads is called a warp.*

to visualize static heat maps directly in *Nsight Graphics* [NVI21b], a popular developer tool by NVIDIA.

Although these tools show promising visualizations of general-purpose memory profiles already, none of them are capable of extracting and visualizing GPU memory profiles from ray-tracing applications dynamically, simulating their cache behavior in the hardware domain, and projecting them into the application domain.

3. Preliminaries

In this section, we briefly describe relevant parts of the pipeline that provide the data for our dashboard. Section 3.1 describes the GPU architecture and binary instrumentation on a GPU. Reuse distances are defined and introduced in section 3.2. In section 3.3, we briefly introduce the foundations of single-hit-point ray tracing.

3.1. Graphics Processing Unit

Architecture Figure 2 shows a compact sketch of most NVIDIA GPU architectures. Other vendors have similar architectures, but use a different terminology. The GPU comes with DRAM attached to an L2 cache. Their L1 cache sits on each *streaming multiprocessor* (SM) and is connected to the L2 cache. Each *warp* consists of 32 *threads* and communicates with the L1 cache, which, however, can optionally be bypassed. Blocks are used as a software-side concept for concurrent scheduling of (multiple) groups of threads to an SM. The *grid* consists of the set of *blocks*.

Binary Instrumentation Binary Instrumentation is a technique to insert a user-specified instrumentation function into the program during run-time, before or after a specific low-level assembly instruction, while still maintaining correct program execution. In order to achieve this, first, the instrumentation tool enables instruction selection by providing a detailed list of instructions and their operands. Then, the selected instruction is replaced with a jump instruction to a so-called *trampoline*. This trampoline then backs up

Time step	0	1	2	3	4
Reference	0	1	1	2	0
Reuse distance	∞	∞	0	∞	2

Table 1: *Exemplary reuse distance calculations given their access time and reference. Time step* 4 *shows that only unique memory references are taken into account. The reuse distance at time step 4 (solid border) is calculated by counting the entries with a dashed border.*

the state, calls the provided instrumentation function, and lastly restores the state. Afterwards (or beforehand, depending on the insertion point), it executes the original replaced instruction and jumps back to the original program. This technique enables tracking memory references, making debugging and profiling very prominent use cases. Recently, this technique has also been officially introduced to GPUs using the NVBIT [VSNK19] binary instrumentation tool, which we also use in this work.

3.2. Reuse Distances

The reuse distance is defined as the number of unique memory references occurring between accesses of two equal references [DZ03]. Traditionally, reuse distances are also called *LRU stack distances* because they implement an LRU stack, counting elements between its top and the position of the address that is potentially already in the stack. More efficient algorithms use splay trees in order to reduce computational complexity [DZ03]. Table 1 shows exemplary reuse distances for a small set of memory references and illustrates how to calculate them.

3.3. Ray Tracing

GPUs traditionally have been optimized for the graphics pipeline (i.e. rasterization). Despite rasterization being more efficient for simple 3D scenes, renderers become extremely complex to implement for photorealistic scenes. Ray tracing [App68; Whi79] solves this by casting rays originating from each pixel in the image plane through the camera model into the scene, collecting primitives intersecting these rays. As intersecting each primitive with each ray becomes inefficient for small input meshes already, bounding volume hierarchies (BVHs) can be used to reduce unneccessary face intersections. These BVHs are tree data structures that recursively subdivide space using bounding volumes, preventing rays from intersecting inner bounding volumes and triangles.

4. Memory Trace Extraction

Our toolset consists of two main components; a *profiler* and a *dashboard*. The profiler extracts memory accesses from the program (section 4.1) and precalculates the computationally intensive cache model (section 4.2), whereas the dashboard focuses on the presentation of this data. In this section, we will focus on the profiler whose pipeline is shown in fig. 3.

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

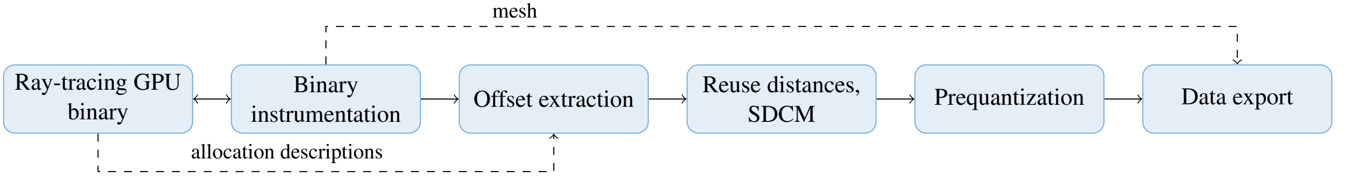


Figure 3: *The profiler's internal pipeline. The exported data can subsequently be read by our dashboard.*

4.1. Reference Tracing

Our profiler, which we intend to be used as a wrapper for ray-tracing programs, first traces memory references followed by a cache simulation. It uses NVBIT in order to inject an instrumentation function before every low-level assembly instruction that includes a memory reference operand. For each warp, our instrumentation function then extracts 32 memory references r, 32 lane predicates and the SM identifier from the device and streams it to the host system. We fetch this data warp-wise, since this is the natural granularity of low-level assembly instructions. As a thread within a warp may diverge, each instruction with side effects uses a predicate vector in order to prevent execution on the corresponding lane. Analogously, we discard memory references with a negative predicate on the host system.

In order to assign these references to their corresponding allocation and enable further processing, we introduced a programming interface for the ray tracer itself that marks allocations, the list of bounding volumes, vertices, faces, and the frame buffer with a representative name and the size of their elements a_e. This way, we can easily distinguish memory references and identify them with their actual allocation. However, if the source code of the ray-tracing program is not available, allocations can also be annotated automatically by analyzing the type of access (load, store) onto them as well as by the traffic involving them. For example, frame-buffer accesses exclusively receive write operations. Given the start address a_s of an allocation and the size of an element, we can then calculate the element-wise offset o within the allocation in eq. (1).

$$o = {}^{(r-a_s)}/_{a_e} \tag{1}$$

4.2. Cache Model

Our profiler can then compute reuse distances D_i based on these offset values for faces and bounding volumes (henceforth referred to as *scene elements*). In order to simulate the behavior of L1 and L2 caches separately, we compute an SM-wise reuse distance for the L1 cache and an SM-independent reuse distance for the L2 cache. For L1 reuse distance computation, we first allocate as many splay trees as there are SMs on the device. Then, as memory accesses occur, we update the tree that corresponds to the SM that caused the memory access in order to simulate the on-chip cache behavior. Both reuse distances are calculated at cache-line granularity, i.e. 128 B for L1 reuse distances and 32 B for L2 reuse distances. As element accesses may overlap multiple cache lines due to misalignment, we may need to compute multiple reuse distances for single-element accesses that are accumulated later in this section.

In contrast to bounding volumes, faces consist of a set of vertices forming a polygon. We assume triangular polygons and an indexed triangle list in our system. Thus, every face-loading procedure results in loading three vertex indices followed by their coordinates. In order to depict this hierarchical loading procedure, we capture reuse distances for faces and additionally their vertices, then accumulate them together.

After computing all reuse distances, we use the *stack distance cache model* (SDCM) [AHH89] in order to compute cache-hit probabilities for each cache line that has been accessed by the element, given the number of cache blocks B and its associativity A in eq. (2).

$$p_i = \sum_{a=0}^{A-1} \binom{D_i}{a} \left(\frac{A}{B}\right)^a \left(1 - \frac{A}{B}\right)^{D_i - a} \tag{2}$$

We then accumulate the SDCM metrics by averaging [ABC*20] every cache line that has been accessed by an element (referred to as *Prequantization* in fig. 3); this is done independently for either cache in eq. (3).

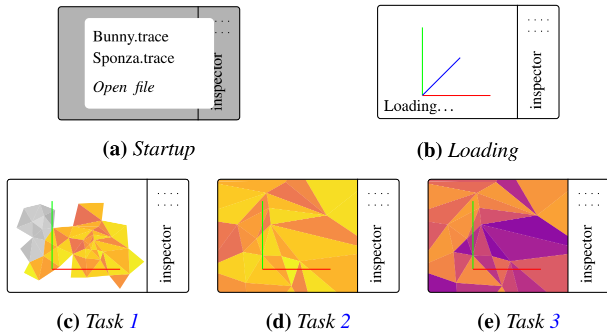
$$s = \sum_{i=1}^{N} p_i, \quad p_{\text{avg}} = \frac{s}{N} \tag{3}$$

Our memory trace extractor then stores a list of tuples $(i, o, s_{L1}, s_{L2}, N)$ for further visualization steps. Note that the extractor stores the sum and divisor separately to be able to correctly apply further accumulations in the visualization. To be able to show the mesh during visualization, we also need to store all input data prior to rendering in the resulting file. We devised a simple and compact binary file format that encodes this information and can easily be interpreted by our web dashboard.

5. Visualization Dashboard

This section describes the graphical part of our visualization. First, we conduct a requirements analysis in section 5.1 and a task analysis in section 5.2. In section 5.3, we describe the structure of the dashboard. In section 5.4, we then explain any further transformations to the profiling data and how the dashboard aggregates it. Finally, we describe the main part of our dashboard—the mesh viewer—in section 5.5, and our visualization of frame buffer accesses in section 5.6.

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.



(a) Startup (b) Loading

(c) Task 1 (d) Task 2 (e) Task 3

Figure 4: *Overview over tasks and initialization of our dashboard. The top row shows the selection and loading procedure of memory traces. The bottom row shows the tasks of our visualization. (c) shows SDCM values mapped onto mesh parts that receive accesses within the current frame. In (d), the mesh viewer points to an identified region of interest. (e) shows the same mesh region using a different ray tracer configuration, enabling comparison with (d).*

5.1. Requirements

The domain experts that our dashboard is targeting are software architects creating and optimizing ray-tracing applications. These software architects usually aim to reduce the number of memory accesses to the DRAM by generating more efficient BVHs [WSWG13], reducing memory blocking by improving the memory layout [WMZ22] or changing the work distribution [AL09; DGP04]. Relying solely on general-purpose profiling values for optimizing ray-tracing applications can be time-consuming and requires deep knowledge of possible effects caused by individual optimizations. A visualization tool should be able to support developers in finding and identifying possible memory communication bottlenecks in their ray-tracing applications in a fine-grained manner. This tool should visualize and rely on already well-established profiling metrics such as L1 and L2 hit rates that should correspond with metrics from the standard profilers by hardware vendors.

This leads to the following requirements for our dashboard:

(R1) **Visualization.** The user should be able to see a visualization of all available metrics mapped onto the mesh geometry and BVH boundaries. Especially, the SDCM values should correspond with the values of general-purpose GPU profilers under the same ray-tracing configuration.

(R2) **Exploration.** The user should be able to change the viewport orientation and the time frame in order to explore regions of interest.

(R3) **Analysis.** The user should be able to compare memory profiles with one another in order to see differences between individual optimization strategies.

5.2. Task Analysis

Given the requirements from section 5.1, we define the tasks of our visualization in the following using the typology of BREHMER and MUNZNER [BM13] for task definitions.

The user can analyze the cache behavior of a ray tracer as follows: First, the user has an option to load a memory trace (section 4) using a list of available example memory traces or load custom generated ones (fig. 4a). After the mesh has been loaded (fig. 4b), the user can define an appropriate granularity (task 1), which defaults to a single frame. Next, in task 2, the user explores regions of interest by navigating through the mesh by altering the viewport, navigating through the time series with a slider and changing the visualized metrics. Finally, the user compares two separate profiles against each other by switching between them in task 3 in order to observe differences between them.

Task 1 (Defining Granularity) In the first task, the user *produces* a reduced set of memory operations by *deriving* an *aggregation* of them as defined in eq. (3). The user defines the level of granularity by setting the total number of chunks which correspond to the number of frames in the visualization. A sketch is shown in fig. 4c, in which we exemplify this behavior as parts of the mesh that remain gray, as they do not receive any memory accesses in the initial frame, given the selected granularity.

Task 2 (Exploring Cache Behavior) In this task, the developer should be enabled to visually *discover* cache behavior in regions of interest of the mesh. Because these regions are not known beforehand, the user does this by *exploring* and *identifying* them. In order to visualize the cache behavior, we *encode* SDCM values using a color map. As our profiler is capable of producing different metrics (e.g. L1 or L2 cache behavior), the visualization enables the user to *change* the desired metric mapped onto the mesh. Finally, our visualization allows the user to *navigate* through the mesh using pointer controls and a slider to change the temporal component. A sketch of the altered mesh viewer is shown in fig. 4d.

Task 3 (Configuration Comparison) As individual memory optimizations result in different cache visualizations generated by task 2, the user must be able to *discover* these differences. This is done by *exploring* and *comparing* these differences. To achieve this, our dashboard is capable of *switching* between visualizations of individual memory profiles. Figures 4d and 4e show a sketch of two configurations that may be compared to each other.

5.3. Structuring

A sketch of our visualization dashboard can be seen in fig. 4, as well as a screenshot in fig. 5. As the dashboard processes large amounts of data, we decided to provide immediate visual feedback for most latency-prone user interactions, as this approach has been shown to reduce user frustration [VBNP16]. The dashboard uses the *reactive* programming paradigm to express dependencies between state variables. It permanently shows a mesh viewer in the background that provides the visualization of the extracted memory and cache trace. Additionally, an *inspector* is shown in a sidebar, providing statistical information about the mesh and offering control over the visualization. Moreover, the inspector visualizes accesses to the frame buffer and also includes two sliders to customize granularity and choose a time step in the series of frames, explained later in this section. When the user opens the dashboard, a modal dialog lists exemplary memory traces and provides the user with a means to open an arbitrary memory trace from the computer. Upon loading the mesh, it is rendered in the mesh viewer while the inspector updates its information accordingly.

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

5.4. Data Sources and Aggregation

Our dashboard quantizes the list of memory accesses into a set of *frames* that only visualize a fixed and uniformly distributed number of memory references. While our visualization dashboard defaults to a single-frame visualization, the inspector gives the user control to select both the number of frames q as well as the currently active frame f to control the visualization in the mesh viewer. Memory references are then distributed equally between frames, retaining their order. This temporal visualization enables a fine-grained analysis of caches which are are highly sensitive on the temporal axis based on realistic scheduling. The time dependency can be implicitly disabled by setting $q = 1$. To align the controls with human intuition, the active frame f can be controlled using a slider [HJM*11].

As mentioned in section 4.2, we calculate the SDCM for L1 and L2 cache simulation for each scene element. Since the memory trace potentially includes multiple accesses to the same scene element within a single frame, our dashboard needs to further aggregate this data. This step is rather simple for the SDCM, as we apply the same calculations as in eq. (3). In our setting, this effectively means that our visualization represents the conditional probability that a scene element produces hits during its intersection, given it has been accessed (i.e. the average probability). Doing this will effectively use the same calculations as for hit rate estimations [ABC*20] corresponding with those of standard GPU profilers.

Additionally, our dashboard allows the user to visualize the access rate and access order of memory references as these values are already implicitly encoded in our memory trace. They are obtained by calculating the access order $o = {}^{i_e}/_{n_f}$ from the unique ordering index i_e given by the memory trace (and actually caused by GPU-internal scheduling (section 4.1)) and the total number of memory accesses in a frame N_f. Access rates $r = {}^{N_e}/_{N_f}$ are calculated based on the number of per-element accesses N_e in the active frame.

In order to enhance run-time performance, the dashboard further implements a caching mechanism that automatically stores these metrics for later on-demand use.

5.5. Cache and Memory Behavior Visualization on the Mesh

The main window is mainly covered by an interactive mesh viewer based on the OpenGL derivative WebGL. Besides rendering the input mesh from the memory trace, it is capable of visualizing parts of the bounding volume hierarchy (BVH) and additional parameters.

The viewport of the mesh viewer can be rotated and translated arbitrarily using pointer controls, and a light source is placed at the position of the viewport in order to enable scene-independent lighting. We set the initial viewport intrinsics and extrinsics to those of the camera from the memory profile. In order to visualize camera parameters from the profile for an easy assessment of ray directions, we draw the corresponding camera frustum using lines. At the initial viewport position, the projection of these lines highlights relevant regions of the scene that caused ray-triangle intersections. In order to properly visualize the frustum, we find the triangle that is central to the viewport, yielding the distance between the viewport and the scene, which in accordance with real-world constraints, is a reasonable upper limit for the far plane.

Mesh The rendered mesh plays an important role in visualizing memory references. Our dashboard uses a mapping between per-face metrics from the memory profile to the face color. To achieve this, it uses a color map to visualize the normalized metrics from section 5.4 with light-independent shading, making sure that colors are not distorted. We elaborate on our choice of color maps later in this section. If a triangle is not accessed, it retains its flat gray shade, such that it can be clearly distinguished from accessed faces and denoting that it is *inactive*.

BVH Our dashboard currently assumes axis-aligned bounding boxes (AABBs) as the shape of the BVH volume. This is not necessarily a problem, as AABBs are a common data structure for bounding volumes. However, our dashboard implementation practically allows for later implementation of other volume shapes. We draw AABBs as the edges of their spanning cuboid. In contrast to the mesh, we decided to only show active AABBs within one frame, lest the visualization be obstructed by needless clutter. We make sure that triangles and bounding boxes share the same color scales of visualized metrics for inter-comparability. This is especially essential for SDCM visualizations, as reuse distances are computed jointly.

Finally, to give the user a better intuition of scene dimensions, we also draw color-encoded Cartesian axes that emerge from the origin. The XZ-plane also includes a grid that makes it easier to perceive distance and perspective.

Color Mapping We chose *Plasma* (see left-hand side of fig. 5) because of its good performance in terms of uniformity and intuitive ordering [BTS*18] and found its discriminative power used for comparison of visualizations (R3) to be sufficient [RS21] for our approach. Additionally, Plasma works particularly well for people with perceptual deficiencies. We decided against a divergent color map, as they usually increase from the center; however, hit rates are generally not balanced around 50 %. We also found that most divergent color maps conflict with inactive parts of the mesh or the background, as they contain shades of gray/black near the center or the poles. While rainbow color maps tend to be more discriminative, they also tend to be harder to interpret (R1) due to uniformity issues.

5.6. Frame Buffer

As our profiler automatically extracts writes to the frame buffer, we decided to include an image matrix in the inspector that visualizes them. Overall, the visualization is quite straightforward, as it receives only a single write instruction per pixel, therefore rendering an analysis of access rates or SDCM visualizations meaningless. Nevertheless, our dashboard can still visualize the access order. To accomplish this, our system uses the dimensions of the frame buffer to draw the matrix where each cell represents a pixel in the frame buffer. Each pixel represents a value from the color map, identical to those of triangles or bounding volumes in the mesh viewer, to represent the relative time when the value was written to the pixel. This visualization will further highlight which regions of images are more efficient for ray tracing, as pixels that are written to earlier involve fewer BVH accesses, which is due to the structure

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

of BVH algorithms. In contrast to the access order visualization on the mesh, which is very similar to the frame buffer visualization when navigated to the same camera position, the frame buffer visualization shows pixels *after* triangles and the BVH have been traversed. Additionally, the frame buffer visualization also encodes pixels that did not result in triangle intersections (i.e. triangles in the background).

6. Discussion

In the following, we discuss the outcome of our visualization dashboard. Section 6.1 gives an overview of our environment. In section 6.2, we discuss the effect of different ray tracer configurations in our visualization. Section 6.3 further involves the temporal functionality of our dashboard, and section 6.4 presents visualizations of the frame buffer.

6.1. Data Sets and Environment Parameters

We evaluated our algorithm on two meshes. Both meshes are very different in their appearance and properties. The Asian Dragon captured by XYZ RGB is a 3D scanning data set with a resolution of 12 million triangles that are uniformly sized locally, but individual regions tend to have different resolutions. The Crytek Sponza mesh (262 267 triangles), however, is a man-made example data set from a game engine. This leads to very irregularly-shaped triangles, optimized to result in scenes with fewer triangles overall, making them more efficient to rasterize. The scene shows Sponza Palace, where we have placed the camera inside the atrium. We present two further datasets in section A of the supplemental material.

We use an *NVIDIA RTX 2080 Ti* GPU to record memory traces and simulate their cache behavior on the basis of realistic parameters of recent GPUs. We assume an L1 cache size of 32 kB with 128-byte lines and an associativity of 64. For the L2 cache, we assume a capacity of 6 MB with 32-byte lines and an associativity of 16. Unfortunately, few exact values are published by GPU vendors [NVI21a], and microbenchmarking experiments contradict both each other [JMSS19; ABC*20] as well as known values from the specifications. Thus, we took realistic averages on the basis of these values. Nevertheless, we do not intend to focus on specific hardware architectures in this paper, as our visualization was designed to simulate configurable target architectures.

Given the huge set of possibilities of ray-tracing implementations in terms of physical accuracy, we choose to focus solely on single-hit-point ray tracers for brevity. However, our visualization scheme generally works on all ray tracers, as it only depends on the content of the memory and not on the specific implementation. The base version of our reference ray tracer implements the *while-while* approach on persistent threads [AL09] and scanline scheduling. We construct a BVH using the surface-area heuristic on a binary tree stored in DFS layout and store leaves, similar to the work of WALD, MORRICAL, and ZELLMANN [WMZ22], implicitly. However, we also implemented less optimal configurations (e.g. median-split, the *if-if* approach, different memory permutations) in order to compare the cache rates using our dashboard.

Figure 5 shows a screenshot of our visualization dashboard and

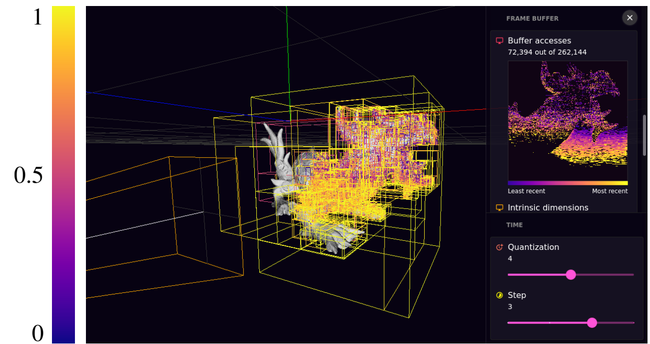


Figure 5: *Screenshot of our visualization dashboard as schematically shown in fig. 4. The background shows the mesh viewer including a visualization of accessed faces and accessed BVH nodes. Aside from various access statistics, the inspector also includes a visualization that illustrates the set of pixels that the ray tracer is writing to the frame buffer in the current time step. The color bar on the left shows the Plasma color map used for visualizing our metrics on triangles and bounding volumes.*

its controls. Through the inspector on the right, the user can set various values such as the time quantization, dividing the list of memory accesses into the number of frames configured. Setting the time step then visualizes one of these access frames and updates the inspector accordingly.

6.2. Effects of Ray Tracer Configurations

To demonstrate the strengths of our dashboard, we implemented various techniques to influence memory access patterns while using the same mesh and camera, and making sure that the output image is the same under all configurations. This allows us to review which approaches have the potential to produce higher hit rates and where on the mesh they are higher/lower. Figure 6 shows a comparison between our different ray tracer configurations which can be done directly through our dashboard (R3). We employ this comparison on all memory references, accumulated and visualized onto a single frame. For an enhanced visible distinctness between geometry hit rates as well as a comparison for an additional mesh, we would like to refer to figs. 2 and 3. in the supplemental material

General Observations It can be seen in fig. 6h that the projected size of triangles (the part of the eye on the top right, for instance) influences the hit rate. Triangles that are sampled more coarsely tend to have a better cache rate than triangles that appear smaller in the projection (i.e. slanted triangles), which also becomes visible in fig. 6g: Triangles on the sides of the dragon tend to have lower hit rates than those in the middle. Similar observations can be made from the BVH visualization: Inner BVH nodes that mainly include slanted triangles tend to produce inferior hit rates compared to BVH nodes containing camera-facing triangles. Additionally, BVH nodes closer to the root indicate higher hit rates, which can also be explained by their higher access frequency. It can also be seen that the implicit occlusion culling of BVHs indeed prevents most memory accesses on the back of the mesh geometry (fig. 6j).

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

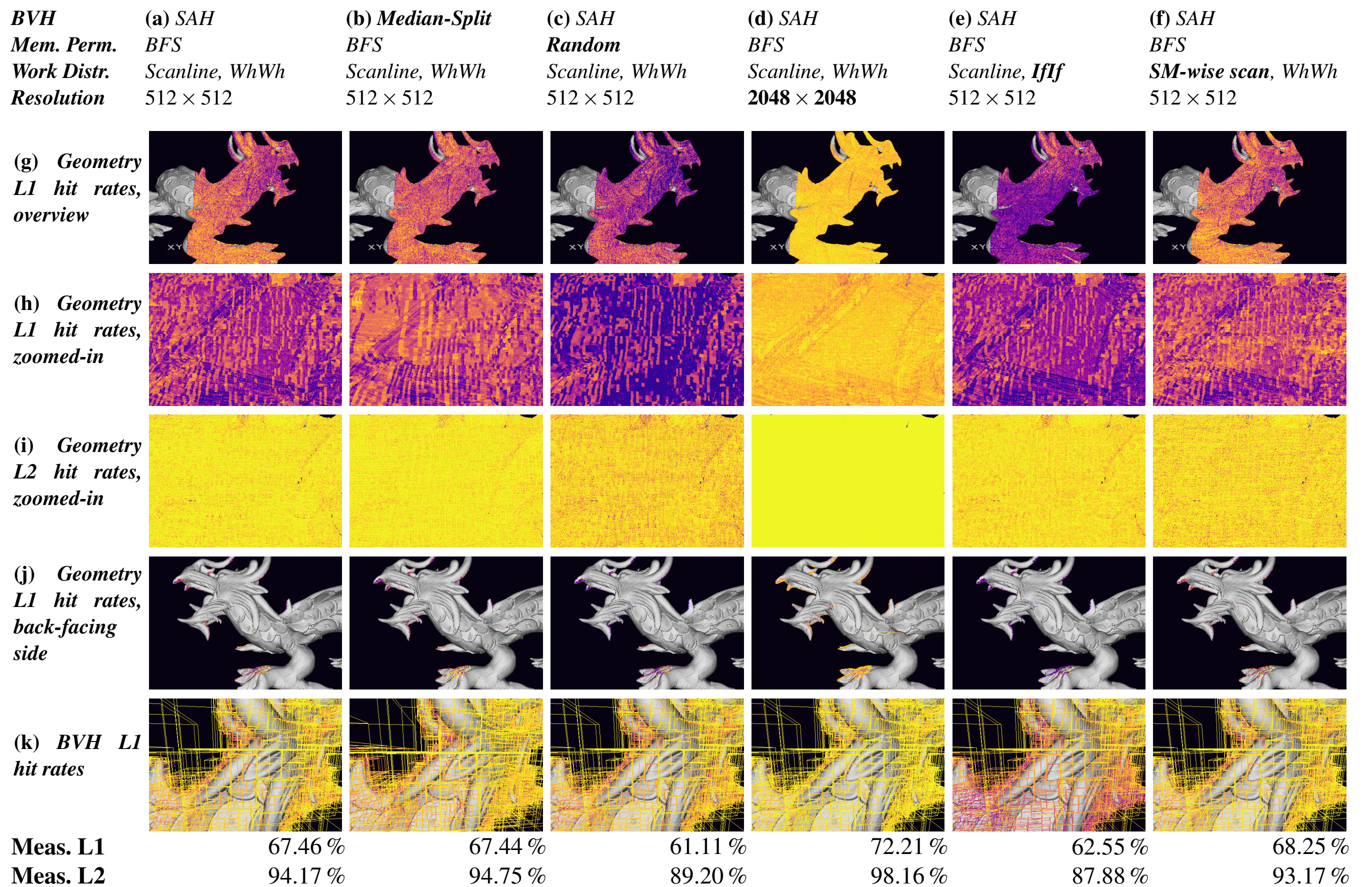


Figure 6: *Single-frame visualizations using the Asian Dragon. It shows all triangle accesses and their hit rates caused by a render call. The zoomed-in region (h) of the L1 SDCM visualization (g) near the eye of the dragon is shown for different configurations of the ray tracer. Similarly, a comparison of L2 visualizations (i) is shown. Memory accesses on back-facing regions (j), and our BVH visualization without any metric mapped to the faces (k) can be seen in the following rows. Finally, the bottom rows state the L1 and L2 hit rate that we profiled using NVIDIA Nsight Compute. With the changed parameter highlighted in bold, the columns are compared against the default configuration (a): The BVH impact (b), the vertex permutation impact (c), the resolution impact (d), and the work distribution impact (e, f).*

Nevertheless, some parts of the mesh receive triangle intersection tests despite their back-facing orientation. This can happen, since the decision as to which BVH subtree is processed first depends on the distance from its intersection point to the camera, which may be inefficient for particular mesh geometries. Generally, our L1 and L2 visualizations seem to correspond to the bare hit rates from the NVIDIA profiler (R1), which is to be expected, as the SDCM has already been used successfully on GPU memory traces [ABC*20].

Vertex Permutation One of the effects we influence is the memory blocking caused by the memory layout. Instead of changing the BVH layout [WG16], we focused on the face and vertex cache efficiency, similar to the work of DeMarle, Gribble, Boulos, and Parker [DGBP05]. As our tracer uses vertex index lists, we change the permutation of vertices in memory in two defined ways. The first order implements a random shuffling operation which is expected to have weak cache performance. The second order is locally optimized by traversing the mesh connectivity using a breadth-first search (BFS) and encoding vertices in traversal order, which is expected to result in more coherent accesses. The effects of different memory layouts can be seen in figs. 6a and 6c. Both visualizations use a BVH constructed with the SAH and use scanline scheduling, whereas fig. 6a results from a BFS-sorted memory layout, and fig. 6c from a randomly shuffled memory layout. As expected, the randomly shuffled layout results in less efficient cache utilization, which can be observed in both the NVIDIA profiler as well as our visualizations of the L1 and L2 caches. Additionally, fig. 6c highlights the strength of our visualization compared to the bare hit-rate found in standard profilers: It allows to explore the mesh and locate regions of interest that may behave differently from one another (R2). More specifically, the remaining visualizations that also use the BFS-sorted memory layout show a diagonal pattern on the SDCM metric, especially in figs. 6a and 6b, while the randomly shuffled layout does not. This effect indicates that there is still room for improvement in our presented BFS-sorted vertex permutations in terms of memory coherence, which could possibly result in a further increase in hit rates. Implementing an optimal vertex memory layout, however, is beyond the scope of this paper.

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

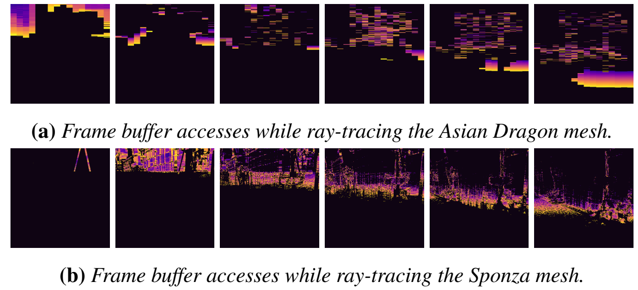
BVH Heuristic The primary motivation for BVHs is the reduction of unnecessary memory traffic. However, different types of BVHs may be more or less efficient in achieving this goal, exhibiting different effects on caches. We implemented a binary BVH using two different construction heuristics. The more efficient one is the popular surface-area heuristic. The second one is the more simplistic median-split strategy. Figures 6a and 6b show that for this particular mesh, the cache performance of faces is approximately the same for the median-split-based implementation. This shows that in this instance, the memory coherence of face accesses does not suffer from different BVH construction heuristics. Nevertheless, fig. 3 in the supplemental material demonstrates that hit rates decrease when tracing a median-split representation of the additional San Miguel scene. The effect that median-split-based BVHs tend to be less efficient can mainly be explained by an increased number of unnecessary visits of BVH nodes caused by a higher end-point overlap [AKL13], which also affects BVH cache rates (fig. 6k).

Frame Buffer Resolution As can be seen in fig. 6d, increasing the size of the frame buffer results in higher cache efficiency on the geometry. This is caused by higher memory coherence that arises from a denser sampling of the mesh. The same effect has already been observed in the eye of the dragon earlier in this section and now becomes apparent for the entire visible part of the mesh. The L2 cache in this instance reaches a hit rate of almost 100 %, making it highly probable that the visible part fully fits into the L2 cache, and mainly cold misses occur.

Work Distribution We experienced that scheduling has an impact on cache performance, regardless of the specific application. To demonstrate this in our visualization, we implemented two different scheduling techniques that we integrated directly into the ray-tracing implementation using the *persistent threads* method [AL09]. One configuration allows our ray tracer to use a simple global scanline scheduling, while in a second configuration, it performs scanline scheduling for each SM independently. By doing this, we expect to enhance the L1 hit rate, as the L1 cache is located directly on the SM. The general-purpose profiler confirms this expectation with a slight enhancement in the L1 hit rate and a slight decline in the L2 hit rate, which is further backed up by our visualization (see fig. 6f, which shows a slightly improved L1 hit rate on the geometry). The decline in the geometry's L2 hit rate can be explained by the fact that when tying scanline scheduling to the SM, the global memory coherence decreases. In the end, the SM-wise scanline approach resulted in fewer rays per second, possibly because of lower L2 performance. Additionally, we implemented the different work distribution techniques *while-while* and the less efficient *if-if* by AILA and LAINE [AL09]. We can observe that the *if-if* implementation leads to worse caching effects, especially on the BVH.

6.3. Time Series

Figure 1 shows changes in the L1 visualization over time (R2) as we process the Asian Dragon with the base version of our ray tracer and the Sponza mesh using a suboptimal random vertex permutation and the *if-if* approach. Especially on the Sponza mesh, it can be seen that the hit rate of a given triangle can drastically change during execution, which has been one of the initial motivations to introduce a time-based exploration: The averaging operation from eq. (3) blurs out such effects, making possible misses invisible to the user. The Sponza visualization also shows that large triangles are accessed on many frames, as they are included in higher-level BVH nodes due to their size. In addition, it is visually noticeable that accessed triangles exhibit higher hit-rate probabilities in subsequent frames due to the continously increasing number of memory accesses onto them. On either mesh, initial frames show bounding volume accesses exclusively. In the following frames, the visualizations then show that the size of a bounding volume, which corresponds to its level, decreases during traversal. This behavior is hardly surprising when keeping the structure of BVH acceleration structures and their traversal in mind. The traversal starts with bounding volumes until it reaches leaf nodes, which in turn contain the triangles being accessed. As a result, no triangles are accessed in the beginning. Additionally, bounding volumes tend to be highlighted in brighter shades of yellow compared to faces, denoting that the former are more likely to be cached.



(a) *Frame buffer accesses while ray-tracing the Asian Dragon mesh.*

(b) *Frame buffer accesses while ray-tracing the Sponza mesh.*

Figure 7: *Frame buffer accesses from the time series of the Asian Dragon (a) and the Sponza (b) meshes. Here, we quantized the access list to eight frames and omitted the last two due to space limitations. This creates the access bands shown above that give spatial insight into how the frame buffer is accessed over time during ray tracing. Brighter colors indicate more recent accesses.*

The figure also shows two versions for each frame of the Asian Dragon mesh. The larger frames are generated during profiling using scanline scheduling. The smaller frames on the bottom right are created via an SM-wise scanline scheduling that assigns a fixed and disjoined region to each SM and performs an independent scanline progression on it. These visualizations clearly illustrate the differences between the two scheduling approaches: Regular scanline scheduling processes the mesh from top to bottom, whereas SM-wise scheduling subdivides the frame buffer into equally-sized regions, each of which are then processed by one SM each, which becomes visually apparent through the more distributed access pattern. Figure 1 in the supplemental material demonstrates the time series visualization on two further meshes and lists the complete set of frames along with the corresponding single-frame visualization.

6.4. Frame Buffer Visualizations

Visualizations of frame buffer writes show an interesting result as shown in fig. 7. The mere visualization of pixel write order emphasizes the shape through variations in color. The silhouette of

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

the object that becomes apparent can be ascribed to the aforementioned local increase in BVH intersections. The actual 3D appearance is based on the fact that faces in slanted regions tend to result in deeper BVH levels in order to optimize the surface area of BVH nodes. The ray tracer then needs to traverse more consecutive BVH nodes along the viewing direction, resulting in a longer traversal time. This behavior then accentuates pixels that are located near the edges of the mesh, highlighted in brighter colors. This phenomenon acts as a good example of the findings of the previously mentioned work by SCHULZ, LEVINE, BREMER, et al. [SLB*11]. We already presented our visualization dashboard mapping hardware performance metrics to the application domain. The visualization of the frame buffer now indicates that the performance metrics from the hardware domain can give insight into the application behavior, especially in our BVH-supported ray-tracing environment. Finally, our visualized frame buffers show an increase of color-mapped access order values in the back of the scene, indicating the scanline scheduling.

7. Conclusion

GPU memory access optimization is an important task in order to enhance the overall performance of a program. With our toolset, this can now be supported by a visual aid showing potential bottlenecks in ray-tracing applications.

Our goal was to present an interactive, web-based visualization tool for GPU memory profiles that provides visual insight into the memory and cache behavior of our reference ray tracer by mapping internal GPU state back onto the 3D object for easy comparison between different ray-tracing implementations. In order to visualize cache behavior, we successfully used the stack distance cache model based on reuse distances on both GPU cache layers. Additionally, we extracted the data to be visualized from a real device running a ray-tracing kernel without needing to modify it.

In order to evaluate our dashboard, we compared visualizations of several ray-tracing-related acceleration structures. We found our visualization to emphasize how ray-tracing implementations and mesh regions lead to different cache performance. In particular, we find that faces of a mesh can be cached less efficiently than bounding volumes due to their memory access frequency, and that caching depends on the projected triangle size and patterns in the memory layout that influence caches negatively. Moreover, our temporal visualization is capable of showing the scheduling behavior of ray tracers. We also presented that the pure visualization of access rates is not suitable for modeling cache-related information, making them a suboptimal performance metric to visually profile ray tracers. Finally, we think of our visualization dashboard as a good example of inter-domain mapping [SLB*11] of performance metrics in the field of ray tracing.

Future Work and Limitations In the future, we would like to optimize our visualization tool in order to support larger meshes. This could be done by applying lossy mesh compression techniques to parts of the mesh that are not accessed, which is true for the majority of them. Additionally, we would like to further evaluate our profiler on other ray-tracing applications with more advanced ray-tracing techniques involving secondary rays. Lastly, as our dashboard does not necessarily expect GPU memory traces, we also intend to adapt our visual profiler to CPUs in order to profile advanced CPU ray-tracing solutions. As a limitation, despite our ability to visualize hit rates with an already useful degree of detail, it may still not be possible to find the root cause of cache inefficiencies in ray tracers in all cases. It should nevertheless be noted that the same issue can be observed in general-purpose profilers to a greater extent. More research is needed to examine such cases and address them accordingly.

Source Code The source code for this paper is available at `https://github.com/maxvonbuelow/rtmemtracer` and a instance of the dashboard at `https://riemann.dev/gpu-blame`.

Acknowledgements

Part of the research in this paper was funded by DFG (Deutsche Forschungsgemeinschaft) project 407 714 161. We thank the anonymous reviewers whose comments helped improve this manuscript.

References

[ABC*20] ARAFA, YEHIA, BADAWY, ABDEL-HAMEED, CHENNUPATI, GOPINATH, et al. "Fast, Accurate, and Scalable Memory Modeling of GPGPUs Using Reuse Profiles". *Proceedings of the 34th ACM International Conference on Supercomputing*. ICS '20. Barcelona, Spain: Association for Computing Machinery, June 2020. DOI: 10.1145/3392717.3392761 2, 4, 6–8.

[ACB*19] ARAFA, YEHIA, CHENNUPATI, GOPINATH, BARAI, ATANU, et al. "GPUs Cache Performance Estimation using Reuse Distance Analysis". *2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC)*. Oct. 2019, 1–8. DOI: 10.1109/IPCCC47392.2019.8958760 2.

[AHH89] AGARWAL, A., HENNESSY, J., and HOROWITZ, M. "An Analytical Cache Model". *ACM Trans. Comput. Syst.* 7.2 (May 1989), 184–215. ISSN: 0734-2071. DOI: 10.1145/63404.63407 2, 4.

[AKL13] AILA, TIMO, KARRAS, TERO, and LAINE, SAMULI. "On Quality Metrics of Bounding Volume Hierarchies". *Proceedings of the 5th High-Performance Graphics Conference*. HPG '13. Anaheim, California: Association for Computing Machinery, July 2013, 101–107. DOI: 10.1145/2492045.2492056 2, 9.

[AL09] AILA, TIMO and LAINE, SAMULI. "Understanding the Efficiency of Ray Traversal on GPUs". *Proceedings of the Conference on High Performance Graphics 2009*. HPG '09. New Orleans, Louisiana: Association for Computing Machinery, 2009, 145–149. DOI: 10.1145/1572769.1572792 2, 5, 7, 9.

[App68] APPEL, ARTHUR. "Some Techniques for Shading Machine Renderings of Solids". *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*. AFIPS '68 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1968, 37–45. DOI: 10.1145/1468075.1468082 3.

[BM13] BREHMER, MATTHEW and MUNZNER, TAMARA. "A Multi-Level Typology of Abstract Visualization Tasks". *IEEE Transactions on Visualization and Computer Graphics* 19.12 (2013), 2376–2385. DOI: 10.1109/TVCG.2013.124 5.

[BTS*18] BUJACK, ROXANA, TURTON, TERECE L., SAMSEL, FRANCESCA, et al. "The Good, the Bad, and the Ugly: A Theoretical Framework for the Assessment of Continuous Colormaps". *IEEE Transactions on Visualization and Computer Graphics* 24.1 (Aug. 2018), 923–933. DOI: 10.1109/TVCG.2017.2743978 6.

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

[CR11] Choudhury, A. N. M. Imroz and Rosen, Paul. "Abstract visualization of runtime memory behavior". *2011 6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*. Sept. 2011, 1–8. DOI: 10.1109/VISSOF.2011.6069452 2.

[DGBP05] DeMarle, David E., Gribble, Christiaan P., Boulos, Solomon, and Parker, Steven G. "Memory sharing for interactive ray tracing on clusters". *Parallel Computing* 31.2 (2005). Parallel Graphics and Visualization, 221–242. DOI: 10.1016/j.parco.2005.02.007 8.

[DGP04] DeMarle, David E., Gribble, Christiaan P., and Parker, Steven G. "Memory-Savvy Distributed Interactive Ray Tracing". *Eurographics Workshop on Parallel Graphics and Visualization*. The Eurographics Association, 2004. DOI: 10.2312/EGPGV/EGPGV04/093-100 2, 5.

[DZ03] Ding, Chen and Zhong, Yutao. "Predicting Whole-Program Locality through Reuse Distance Analysis". *SIGPLAN Not.* 38.5 (May 2003), 245–257. DOI: 10.1145/780822.781159 2, 3.

[HJM*11] Hao, Ming, Janetzko, Halldor, Mittelstädt, Sebastian, et al. "A Visual Analytics Approach for Peak-Preserving Prediction of Large Seasonal Time Series". *Comput. Graph. Forum* 30 (June 2011), 691–700. DOI: 10.1111/j.1467-8659.2011.01918.x 6.

[IGJ*14] Isaacs, Katherine E., Giménez, Alfredo, Jusufi, Ilir, et al. "State of the Art of Performance Visualization". *EuroVis - STARs*. The Eurographics Association, 2014. DOI: 10.2312/eurovisstar.20141177 2.

[JBC*15] Juckeland, Guido, Brantley, William, Chandrasekaran, Sunita, et al. "SPEC ACCEL: A Standard Application Suite for Measuring Hardware Accelerator Performance". *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*. Springer International Publishing, 2015, 46–67. DOI: 10.1007/978-3-319-17248-4_3 2.

[JMSS19] Jia, Zhe, Maggioni, Marco, Smith, Jeffrey, and Scarpazza, Daniele Paolo. "Dissecting the NVidia Turing T4 GPU via Microbenchmarking". *CoRR* (Mar. 2019). arXiv: 1903.07486 7.

[MB90] MacDonald, David J. and Booth, Kellogg S. "Heuristics for Ray Tracing Using Space Subdivision". *Vis. Comput.* 6.3 (May 1990), 153–166. DOI: 10.1007/BF01911006 2.

[NAW*96] Nagel, Wolfgang E., Arnold, Alfred, Weber, Michael, et al. "VAMPIR: Visualization and Analysis of MPI Resources". *Supercomputer* 12 (1996), 69–80. DOI: 2128/11887 2.

[NVI21a] NVIDIA Corporation. "CUDA Programming Guide". (2021). URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide 7.

[NVI21b] NVIDIA Corporation. "Nsight Graphics". (2021). URL: https://developer.nvidia.com/nsight-graphics 3.

[PBD*10] Parker, Steven G., Bigler, James, Dietrich, Andreas, et al. "OptiX: A General Purpose Ray Tracing Engine". *ACM Trans. Graph.* 29.4 (July 2010). DOI: 10.1145/1778765.1778803 2.

[PKGH97] Pharr, Matt, Kolb, Craig, Gershbein, Reid, and Hanrahan, Pat. "Rendering Complex Scenes with Memory-Coherent Ray Tracing". *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '97. USA: ACM Press/Addison-Wesley Publishing Co., 1997, 101–108. DOI: 10.1145/258734.258791 2.

[Ros13] Rosen, Paul. "A Visual Approach to Investigating Shared and Global Memory Behavior of CUDA Kernels". EuroVis '13. Leipzig, Germany: The Eurographs Association & John Wiley & Sons, Ltd., 2013, 161–170. DOI: 10.1111/cgf.12103 2.

[RS21] Reda, Khairi and Szafir, Danielle Albers. "Rainbows Revisited: Modeling Effective Colormap Design for Graphical Inference". *IEEE Transactions on Visualization and Computer Graphics* 27.2 (2021), 1032–1042. DOI: 10.1109/TVCG.2020.3030439 6.

[SLB*11] Schulz, Martin, Levine, Joshua A., Bremer, Peer-Timo, et al. "Interpreting Performance Data across Intuitive Domains". *2011 International Conference on Parallel Processing*. 2011, 206–215. DOI: 10.1109/ICPP.2011.60 2, 10.

[SM06] Shende, Sameer S. and Malony, Allen D. "The Tau Parallel Performance System". *Int. J. High Perform. Comput. Appl.* 20.2 (May 2006), 287–311. DOI: 10.1177/1094342006064482 2.

[VBNP16] Varvello, Matteo, Blackburn, Jeremy, Naylor, David, and Papagiannaki, Konstantina. "EYEORG: A Platform For Crowdsourcing Web Quality Of Experience Measurements". *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '16. Irvine, California, USA: Association for Computing Machinery, 2016, 399–412. DOI: 10.1145/2999572.2999590 5.

[vdDKTG97] Van der Deijl, Eric, Kanbier, Gerco, Temam, Olivier, and Granston, Elena D. "A Cache Visualization Tool". *Computer* 30.7 (July 1997), 71–78. DOI: 10.1109/2.596631 2.

[VSM*18] Vasiou, Elena, Shkurko, Konstantin, Mallett, Ian, et al. "A Detailed Study of Ray Tracing Performance: Render Time and Energy Cost". *Vis. Comput.* 34.6–8 (June 2018), 875–885. DOI: 10.1007/s00371-018-1532-8 2.

[VSNK19] Villa, Oreste, Stephenson, Mark, Nellans, David, and Keckler, Stephen W. "NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs". *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '52. Columbus, OH, USA: Association for Computing Machinery, Oct. 2019, 372–383. DOI: 10.1145/3352460.3358307 2, 3.

[WG16] Wodniok, Dominik and Goesele, Michael. "Recursive SAH-Based Bounding Volume Hierarchy Construction". *Proceedings of the 42nd Graphics Interface Conference*. GI '16. Victoria, British Columbia, Canada: Canadian Human-Computer Communications Society, June 2016, 101–107 8.

[Whi79] Whitted, Turner. "An Improved Illumination Model for Shaded Display". *SIGGRAPH Comput. Graph.* 13.2 (Aug. 1979), 14. DOI: 10.1145/965103.807419 3.

[WMZ22] Wald, Ingo, Morrical, Nate, and Zellmann, Stefan. "A Memory Efficient Encoding for Ray Tracing Large Unstructured Data". *IEEE Transactions on Visualization and Computer Graphics* 28.1 (2022), 583–592. DOI: 10.1109/TVCG.2021.3114869 2, 5, 7.

[WSWG13] Wodniok, Dominik, Schulz, Andre, Widmer, Sven, and Goesele, Michael. "Analysis of Cache Behavior and Performance of Different BVH Memory Layouts for Tracing Incoherent Rays". *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2013. DOI: 10.2312/EGPGV/EGPGV13/057-064 2, 5.

[YBD01] Yu, Y., Beyls, K., and D'Hollander, E.H. "Visualizing the impact of the cache on program execution". *Proceedings Fifth International Conference on Information Visualisation*. July 2001, 336–341. DOI: 10.1109/IV.2001.942079 2.

[YLZ*16] Yang, Xin, Liu, Qi, Zhang, Pengfei, et al. "DKD: A Fast k-d Tree Update Design for Dynamic Scenes". *Comput. Animat. Virtual Worlds* 27.3–4 (May 2016), 340–350. DOI: 10.1002/cav.1717 2.

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.