Visual Insights into Memory Behavior of GPU Ray Tracers

Zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
Genehmigte Dissertation von Maximilian Alexander von Bülow aus Mainz
Tag der Einreichung: 12. März 2024, Tag der Prüfung: 30. April 2024

1. Gutachten: Prof. Dr. techn. Dr.-Ing. eh. Dieter W. Fellner
2. Gutachten: Prof. Dr. Ir. Arjan Kuijper
3. Gutachten: Prof. Dr.-Ing. Gabriel Zachmann
Darmstadt, Technische Universität Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department

Interactive Graphics Systems Group

Visual Insights into Memory Behavior of GPU Ray Tracers

Accepted doctoral thesis by Maximilian Alexander von Bülow

Date of submission: 12. März 2024
Date of thesis defense: 30. April 2024

Darmstadt, Technische Universität Darmstadt

Please cite this document as:
DOI: https://doi.org/10.2312/diss.20241000
DOI: https://doi.org/10.26083/tuprints-00027476
Year thesis published in TUprints: 2024

Erklärungen laut Promotionsordnung

§ 8 Abs. 1 lit. c PromO

Ich versichere hiermit, dass die elektronische Version meiner Dissertation mit der schriftlichen Version übereinstimmt.

§ 8 Abs. 1 lit. d PromO

Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs mitzuteilen.

§ 9 Abs. 1 PromO

Ich versichere hiermit, dass die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen verfasst wurde.

§ 9 Abs. 2 PromO

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, 12. März 2024

M. von Buelow

Abstract

Ray tracing is a fundamental rendering technique that typically projects three-dimensional representations of a scene onto a two-dimensional display. This is achieved by perspectively sampling a set of rays into the scene and computing intersections against the relevant geometry. Secondary rays may be sent out from these intersection points, allowing for physically correct global illumination on the reverse photon direction.

Real-time rendering has historically used classical rasterization pipelines, which are straightforward to implement on hardware as they form a data-parallel problem projecting the whole scene into the coordinate system of the image. In contrast, task-parallel ray tracing suffers from incoherency between rays. However, recent advances in ray tracing have led to more efficient approaches, resulting in even more efficient embedded hardware implementations. While these approaches are already capable of rendering realistic images, further improvements in run-time performance can compensate for computational time to achieve higher framerates, display resolutions, ray-tracing recursion depths, or reducing the energy footprint of ray-tracing data centers.

A fundamental technique for improving ray-tracing performance is the use of bounding-volume hierarchies (BVH), which prevent rays from intersecting the entire scene, especially in occluded or distant regions. In addition to the structural efficiency of a BVH, the primary bottlenecks of GPU ray tracing are memory latency and work distribution. These factors mainly result in more coherent memory accesses, making caching more efficient.

Creating programs with the goal of achieving higher caching rates typically requires increased programming efforts and a deep understanding of the hardware, as an additional abstraction layer is introduced, making the memory pipeline less transparent. General-purpose profilers aim to support the implementation process. However, they typically display caching rates based on kernel calls. This is because these values are measured using basic hardware counters that do not distinguish between the context of a memory access. In many cases, it would be useful to have a more detailed representation of memory-related profiling metrics, such as the number of recordings per memory allocation or projections into other domains, such as the framebuffer or the scene geometry.

This thesis presents a new method for simulating the GPU memory pipeline accurately. The method uses memory traces exported by dynamic binary instrumentation, which can be applied to any compiled GPU binaries, similar to standard profilers. The exported memory profiles can be used for performance visualization purposes in individual domains, as well as traditional memory profiling metrics that can be displayed in finer granularity than usual. A method for mapping memory metrics onto the original scene is included, allowing users to explore profiling results within the scene domain, making the profiling process more intuitive. In addition, this thesis presents a novel compressed ray-tracing implementation that optimizes its memory footprint by making assumptions about the topological properties of the scene to be rendered. The findings can be used to evaluate and optimize a wide range of ray tracing and ray marching applications in a user-friendly manner.

Zusammenfassung

Ray Tracing ist eine wichtige Rendering-Technik, bei der in der Regel dreidimensionale Repräsentationen einer Szene auf eine zweidimensionale Anzeige projiziert werden. Dies wird erreicht, indem Strahlen perspektivisch in die Szene gesampelt und Schnittpunkte mit der relevanten Geometrie berechnet werden. Von diesen Schnittpunkten aus können Sekundärstrahlen ausgesandt werden, die eine physikalisch korrekte globale Beleuchtung in der umgekehrten Photonenrichtung ermöglichen.

Beim Echtzeit-Rendering wurden in der Vergangenheit klassische Rasterisierungspipelines verwendet, die auf der Hardware einfach zu implementieren sind, da sie ein datenparalleles Problem darstellen, das die gesamte Szene in das Koordinatensystem des Bildes projiziert. Im Gegensatz dazu leidet das aufgabenparallele Ray Tracing unter der Inkohärenz zwischen den Strahlen. Jüngste Fortschritte beim Ray Tracing haben jedoch zu effizienteren Ansätzen geführt, die zu noch effizienteren Implementierungen für eingebettete Hardware führen. Während diese Ansätze bereits in der Lage sind, realistische Bilder zu rendern, können weitere Verbesserungen die Rechenzeit kompensieren, um höhere Frameraten, Bildschirmauflösungen und Ray-Tracing-Rekursionstiefen zu erreichen oder den Energiebedarf von Ray-Tracing-Datenzentren zu reduzieren.

Eine grundlegende Technik zur Verbesserung der Ray-Tracing-Leistung ist die Verwendung von Bounding-Volume-Hierarchien (BVH), die verhindern, dass alle Strahlen die gesamte Szene schneiden müssen, insbesondere in verdeckten oder weit entfernten Regionen. Neben der strukturellen Effizienz einer BVH sind die primären Flaschenhälse des GPU-Ray-Tracings die Speicherlatenz und die Arbeitsverteilung. Diese Faktoren führen vor allem zu kohärenteren Speicherzugriffen, wodurch das Caching effizienter wird.

Die Erstellung von Programmen mit dem Ziel, höhere Caching-Raten zu erreichen, erfordert in der Regel einen erhöhten Programmieraufwand und ein tiefes Verständnis der Hardware, da eine zusätzliche Abstraktionsschicht eingeführt wird, die die Speicherpipeline weniger transparent macht. Allzweck-Profiler zielen darauf ab, den Implementierungsprozess zu unterstützen. Sie zeigen jedoch in der Regel Caching-Raten auf der Granularität von Kernel-Aufrufen an. Dies liegt daran, dass diese Werte mit einfachen Hardware-Zählern gemessen werden, die nicht zwischen dem Kontext eines Speicherzugriffs unterscheiden. In vielen Fällen wäre eine detailliertere Darstellung speicherbezogener Profiling-Metriken nützlich, z. B. die Anzahl der Aufzeichnungen pro Speicherzuweisung oder Projektionen in andere Bereiche, wie den Framebuffer oder die Szenengeometrie.

In dieser Arbeit wird eine neue Methode zur genauen Simulation der GPU-Speicherpipeline vorgestellt. Die Methode verwendet Speicherprofile, die von der dynamischen Binärinstrumentierung exportiert werden, die auf bereits kompilierten GPU-Programmen angewendet werden kann, ähnlich wie bei Standard-Profilern. Die exportierten Speicherprofile können für Performanz-Visualisierungen in individuellen Domänen verwendet werden, ebenso wie für herkömmliche Speicher-Profiling-Metriken, die jedoch in feinerer Granularität als üblich angezeigt werden können. Insbesondere ist eine Methode zur Abbildung von Speichermetriken auf die ursprüngliche Szene enthalten, die es dem Benutzer ermöglicht, Profiling-Ergebnisse innerhalb der Szenendomäne zu untersuchen. Dies macht den Profiling-Prozess intuitiver. Darüber hinaus wird in dieser Arbeit eine neuartige komprimierte Ray-Tracing-Implementierung vorgestellt, die ihren Speicherbedarf optimiert, indem sie Annahmen über die topologischen Eigenschaften der zu rendernden Szene trifft. Die Ergebnisse können verwendet werden, um eine breite Palette von Ray-Tracing- und Ray-Marching-Anwendungen auf benutzerfreundliche Weise zu bewerten und zu optimieren.

Contents

List of Figures

Part I.

Synopsis

1. Introduction

This thesis focuses on visual representations of memory profiling metrics evaluated on GPU ray tracers. The motivation for this topic is presented below, followed by an outline of the thesis structure.

1.1. Motivation

Ray tracing is a rendering technique that is becoming increasingly important in the process of transforming three-dimensional scenes into images. These 3D models can result from 3D scans or can be modeled for CAD purposes or for use in video games. Ray tracing represents a foundational approach for numerous applications that utilize computer graphics. Consequently, it is a methodology with a multitude of applications in modeling, entertainment, and medicine. Ray tracing is increasingly replacing classical rasterization pipelines for rendering due to its superior capabilities in physically correct rendering [PJH17]. The primary distinction between rasterization and ray tracing is the direction in which the geometry is processed, from the scene to the image. Rasterization takes the set of all primitives of the geometry and transforms it into the coordinate system of the image. The rasterizer draws each 2D-transformed primitive into the image by filling pixels inside the primitive and then applies shading on a pixel-level. This approach can be easily accelerated through parallelization, as the transformation of the input data and shading on a pixel-level is data-parallel [HS86]. In contrast, ray tracing defines rays in the image domain, pointing towards the scene from the actual perspective. These rays are then intersected with the set of primitives independently, which makes the approach more computationally inefficient. However, tree-based spatial acceleration structures reduce the complexity of intersecting the scene from linear to logarithmic. Such an acceleration structure requires that each ray has its own control path, making ray tracing a task parallel problem. Consequently, parallelization is typically performed on each ray. Moreover, the utilization of such an acceleration structure results in incoherent memory accesses due to the random access of vertices. The total amount of available memory also serves as a hardware limitation, setting an upper bound on the amount of available scene geometry and typically restricting the scene resolution.

Graphics processing units (GPUs) are specialized processors that are optimized for data-parallel operations, allowing for the execution of a large number of threads in parallel [NVI22]. They consist of multiple independent multiprocessors, similar to those found in *central processing units* (CPUs), which are further subdivided into data-parallel units. Each device has its own *random access memory* (RAM), which is typically cached using two layers residing on the device and on each multiprocessor. The program executed on a GPU, called a *kernel*, describes the work that is executed on one thread in parallel. Parallelizing ray tracing on such devices is a non-trivial problem, due to the extensive use of data-parallelism. It is therefore important to avoid diverging code paths in order to achieve optimal performance. Moreover, incoherent memory accesses may be made sequential, which can result in inefficient cache utilization, a phenomenon that is similar to that observed in CPU architectures.

General-purpose profilers, such as *Nsight Compute* for NVIDIA architectures, may be employed to quantify inefficiencies in GPU-based computations. These profilers utilize hardware counters, specifically those recorded by the *memory management unit* (MMU), to account for the number of memory accesses or hits into the special-purpose register file of the GPU. However, this implementation has the disadvantage of accumulating various causes of inefficiency to the granularity of a kernel call. If a kernel executes multiple memory operations, such as a matrix multiplication that loads entries from two matrix objects, the memory metrics of both accesses are accumulated. This effect is manageable for smaller kernels, but larger kernels become difficult to optimize, even for experienced GPU developers. Moreover, manual analysis is usually very time-consuming.

This thesis proposes a pipeline that enables software engineers to identify the root cause of inefficiencies in their kernels by obtaining profiling values at a finer granularity. The most granular level of profiling would be the memory access itself. However, these profiling values are not natively available, and therefore require simulation of critical parts of the memory pipeline in software. This allows for the assignment of a tag to each memory access, indicating whether it was a hit or a miss in the cache. These fine-grained tags permit arbitrary accumulations, including per-kernel accumulation, which facilitates validation of the simulator against the proprietary profiler. Using such a simulation framework, several domains of profiling metrics are defined, including textual per-allocation metrics and visualized metrics for the to-be-rendered scene, rendered image, or source code. Our approach can be utilized to analyze a diverse array of GPU applications, with a particular emphasis on ray tracing implementations, in a comprehensive visual representation to facilitate program optimization.

Memory capacity issues in ray tracing can be addressed by employing compression techniques that are both simple and structured in such a way that they can be reconstructed on-the-fly on the GPU during ray tracing. The initial step typically involves the removal of unnecessary references from the BVH tree data structure [Wal+01] or the storage of multiple primitives in a leaf instead of the full storage of a deep BVH representation. Geometry can be compressed by quantizing vertices or re-arranging primitive vertices and the respective connectivity information in a locally structured way that allows for implicit vertex access. This technique, known as micro mesh rendering, has been described in recent works [BP23]. It is currently implemented in proprietary hardware, which limits its accessibility. This thesis also presents an approach to rendering of compressed scene geometry on GPUs employing a pure software implementation.

1.2. Contribution Overview

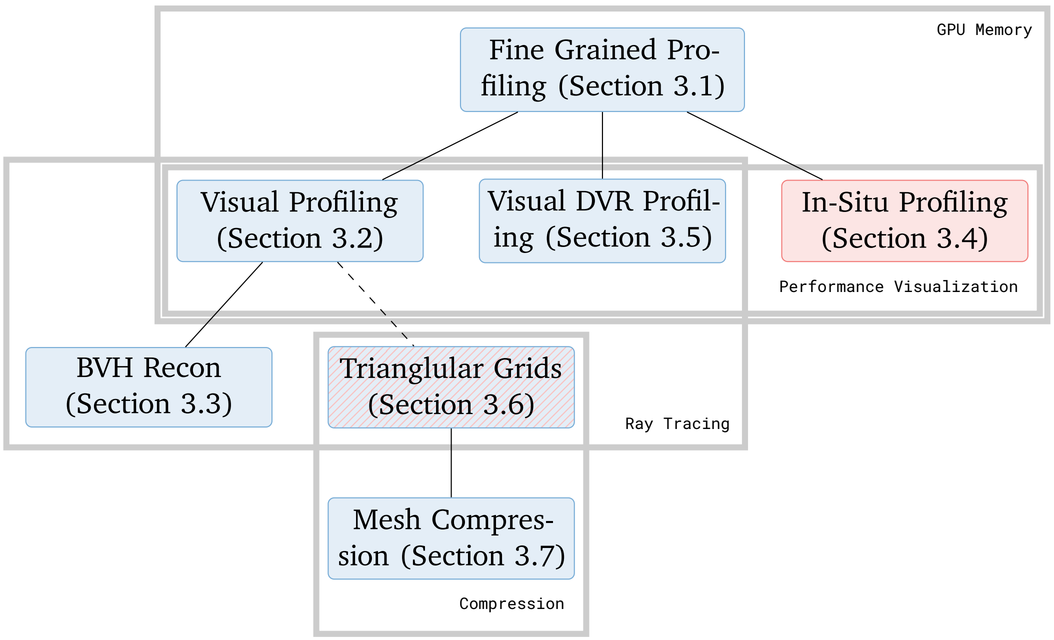

GPU Memory
Fine Grained Profiling (Section 3.1)
Visual Profiling (Section 3.2)
Visual DVR Profiling (Section 3.5)
In-Situ Profiling (Section 3.4)
Performance Visualization
BVH Recon (Section 3.3)
Trianglular Grids (Section 3.6)
Ray Tracing
Mesh Compression (Section 3.7)
Compression

Figure 1.1.: The contributions of this thesis are represented graphically as nodes with their relationships as edges. Solid lines indicate a strong relationship, while dashed lines indicate a conceptual relationship. The color red is employed to represent unpublished works, whereas blue is utilized to indicate peer-reviewed works. The boxes serve to group multiple nodes into their individual categories.

This section categorizes the contributions and summarizes their relationships. Figure 1.1 shows a diagram of these categories and contributions. The categories of *GPU memory*, *ray tracing*, *compression*, and *performance visualization* were selected to describe the motivation of this thesis, which aims to present visual insights into the GPU memory behavior of various ray-tracing configurations that incorporate on-the-fly compression techniques.

GPU Memory This thesis is based on the work on fine-grained profiling [BGF22], which is summarized in Section 3.1. The work models parts of the memory pipeline in modern GPU architectures. It introduces a technique for extracting operation-wise memory profiling values that can be accumulated to coarser representations such as per-allocation or traditional per-program metrics. This work also provides a foundation for the visual profiling tools presented in this thesis. The visual profiler for direct volume renderers [Bue+24], summarized in Section 3.5, is able to use these operation-wise annotations and accumulate them in the image domain for each pixel individually. A profiling application for surface ray tracers [Bue+22a], summarized in Section 3.2, additionally accumulates profiling values in the 3D scene domain by assigning profiling values to triangles and encoding them using color mapping. In addition to its application in ray tracing, the toolkit can also be utilized to incorporate in-situ profiling metrics directly into the source code [PBS24], as demonstrated in Section 3.4. This can assist software developers during implementation.

Ray Tracing The visual profiling systems concentrate on ray-tracing implementations, which renders the aforementioned papers pertinent to this category. Furthermore, this thesis presents a BVH reconstruction technique [Bue+22b], which is summarized in Section 3.3. This technique may be utilized as an optional extension to the visual profiler, allowing for a more abstraction representation of internal BVH data structures utilized by individual surface ray tracers. Additionally, we investigated ray tracing in a special scenario that assumes meshes of triangular grids as geometry [Bue24; BKF23], which is summarized in Section 3.6. The implementation is able to render triangularly structured grids directly on GPUs. Since such grids require fewer bytes than lists of triangles, the compression of the memory footprint of the ray tracer is possible, which allows for the rendering of higher resolution meshes.

Compression The triangular-grid ray tracer employs a compressed representation of mesh connectivity to reduce memory requirements, as previously described. A more comprehensive examination of mesh compression [BGG17], which is summarized in

Section 3.7, describes an advanced single-rate compression technique that highlights the potential of connectivity compression on polygonal meshes.

Performance Visualization The central applications described in this thesis are presented in the visual profiling paper, the in-situ code profiling paper, and the DVR profiling paper. All of these papers emply performance visualization techniques and apply domain transformations from the hardware domain to the application domain as described by Schulz, Levine, Bremer, Gamblin, and Pascucci [Sch+11]. The specialized ray tracer, presented in Fig. 1.1, which aims to optimize the memory utilization of the computing device, may be evaluated with performance visualization in future work.

Structure The sections in Chapter 3 start with the fine-grained profiler in Section 3.1 as a basis, as indicated in Fig. 1.1. Then, the visual profiling work is summarized in Section 3.2 as it can be seen as an application of the fine-grained profiling technique evaluated on a set of individual ray tracing configurations. The BVH reconstruction approach, which extends the visual profiler to handle arbitrary BVH data structures, is summarized in Section 3.3. Section 3.4 summarizes the application of the profiler for in-situ code profiling, while Section 3.5 summarizes its application to DVR. Finally, in Section 3.6, the work on triangular grid ray tracing is summarized as another ray-tracing implementation that builds on the connectivity compression techniques, which are discussed in more detail in Section 3.7.

2. Background

This chapter presents the theoretical background for this thesis. It begins with a detailed description of ray tracing (Section 2.1), followed by an explanation of compression basics in Section 2.2. It then addresses GPU architecture (Section 2.3), and finally, fundamental performance visualization techniques (Section 2.4).

2.1. Ray Tracing

Ray tracing is a rendering concept that employs a set of rays sent out into a scene in reverse direction as physical photons to render an arbitrary definition of geometry into a two-dimensional image. This approach has been proven to be physically correct, i.e., tracing in inverse photon direction, a century before and is known as the *Helmholtz Reciprocity* [Hel67]. The earliest computational approaches for ray tracing were developed by Appel [App68] and Whitted [Whi79]. Building upon these foundational principles, ray tracing has been the subject of extensive research in various domains, including the development of more accurate physical reflectance models [Nic+77], the pursuit of enhanced performance or the adaption of this technique to novel applications [Kel+13]. The following sections will first provide a theoretical overview of ray tracing, before delving into the specifics of its performance optimization.

2.1.1. Intersections

In its formal definition, ray tracing is the process of identifying the point of intersection between arbitrary geometry and a ray for each pixel in the framebuffer, which represents the final rendered image. Let $\vec{r} : \mathbb{R} \to \mathbb{R}^3$ be a ray with origin position $\vec{o} \in \mathbb{R}^3$ in direction $\vec{d} \in \mathbb{R}^3$, such that $||\vec{d}||_2 \leq 1$, defined as

$$\vec{r}(t) = \vec{o} + t\vec{d}. \tag{2.1}$$

Moreover, let $T : \left(\mathbb{R}^3, \mathbb{R}^3\right) \to \mathbb{R}$ be the function for geometry intersection, defined as $T(\vec{o}, \vec{d})$, which returns the ray parameter t of the closest intersection. In the majority of cases, the geometry is a list of primitives with each primitive p having their corresponding intersection functions $T_p : \left(\mathbb{R}^3, \mathbb{R}^3\right) \to \mathbb{R}$. In this case it is defined as

$$T(\vec{o}, \vec{d}) = \min_p T_p(\vec{o}, \vec{d}). \tag{2.2}$$

In the following, the term t is used for simplicity, which should represent the function $T(\vec{o}, \vec{d})$ in any case.

Triangles In computer graphics, triangles are the most prevalent primitive shape, as they represent the most concise definition of a surface. This is because most other primitive shapes can be reduced to a set of limit triangles with minimal effort. A very typical and fast intersection has been described by MÖLLER and TRUMBORE [MT97]. Triangles are usually defined using their corner vertices $\{v_0, v_1, v_2\}$ with $v_i \in \mathbb{R}^3$. Let $u, v, w \in \mathbb{R}$, such that $u + v + w = 1$, be barycentric coordinates [Möb27] that express coordinates within a triangle as a convex linear combination

$$p(u, v) = wv_0 + uv_1 + vv_2 = v_0 + u(v_1 - v_0) + u(v_2 - v_0). \tag{2.3}$$

Intersecting the ray with the plane that is defined by the triangle corners is done by solving $r(t) = p(u, v)$ for u, v and t:

$$\vec{o} + t\vec{d} = v_0 + u(v_1 - v_0) + v(v_2 - v_0) \tag{2.4}$$
$$\Leftrightarrow \vec{o} - v_0 = t(-\vec{d}) + u(v_1 - v_0) + v(v_2 - v_0) \tag{2.5}$$
$$\Leftrightarrow \vec{o} - v_0 = \begin{bmatrix} -\vec{d} & (v_1 - v_0) & (v_2 - v_0) \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} \tag{2.6}$$

Given the definitions $A = \begin{bmatrix} -\vec{d} & (v_1 - v_0) & (v_2 - v_0) \end{bmatrix}$ and $b = \vec{o} - v_0$, the triple product, and CRAMER's rule [Cra50], we can solve $A \begin{bmatrix} t & u & v \end{bmatrix}^T = b$:

$$\begin{bmatrix} t & u & v \end{bmatrix} = \frac{\begin{bmatrix} \det \begin{bmatrix} b & A_{*,1} & A_{*,2} \end{bmatrix} & \det \begin{bmatrix} A_{*,0} & b & A_{*,2} \end{bmatrix} & \det \begin{bmatrix} A_{*,0} & A_{*,1} & b \end{bmatrix} \end{bmatrix}}{\det A} \tag{2.7}$$
$$= \frac{(\vec{o} - v_0)^T \begin{bmatrix} (v_1 - v_0) \times (v_2 - v_0) & \vec{d} \times (v_2 - v_0) & (v_1 - v_0) \times \vec{d} \end{bmatrix}}{(v_1 - v_0) \cdot \left(\vec{d} \times (v_2 - v_0)\right)} \tag{2.8}$$

If the determinant of A is zero, then the ray is parallel to the triangle and no meaningful solution exist. Additionally, u and v can be used to check if the ray intersects the triangle: $u + v \leq 1 \land u \geq 0 \land v \geq 0$.

Spheres A sphere with origin $\vec{c} \in \mathbb{R}^3$ and radius $R \in \mathbb{R}$ can be defined with the locus of all points $\vec{x} \in \mathbb{R}$, fulfilling

$$\|\vec{x} - \vec{c}\|_2^2 = R^2. \tag{2.9}$$

In order to find the parameters of an intersecting ray, we set $\vec{x} = \vec{r}(t)$, resulting in

$$\|\vec{o} + t\vec{d} - \vec{c}\|_2^2 - R^2 = 0. \tag{2.10}$$

Solving this quadratic form results in the interval of the intersection parameters $[t_0, t_1]$, with

$$t_0 = -\vec{d} \cdot (\vec{o} - \vec{c}) + \sqrt{\delta} \tag{2.11}$$
$$t_1 = -\vec{d} \cdot (\vec{o} - \vec{c}) - \sqrt{\delta}. \tag{2.12}$$

The discriminant is defined as

$$\delta = \left(\vec{d} \cdot (\vec{o} - \vec{c})\right)^2 - (\vec{o} - \vec{c})^2 + R^2. \tag{2.13}$$

If $\delta < 0$, no solution exists and the ray does not intersect the sphere.

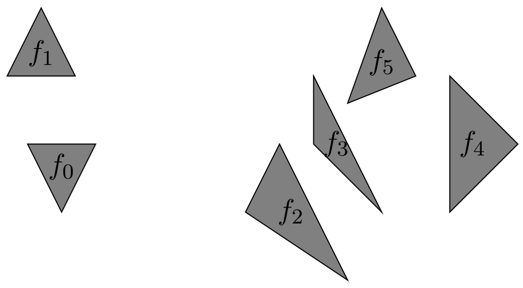
Axis-Aligned Bounding Boxes An *axis-aligned bounding box* (AABB) is typically defined as the lower bound $\vec{l}$ and upper bound $\vec{u}$ that span the axis-aligned volume. Intersections against an axis-aligned bounding box are essentially accomplished by successively intersecting against three one-dimensional bounding slabs $[\vec{l}_i, \vec{u}_i]$. Without loss of generality, let the ray and the bounding box be considered one-dimensional, where i represents an arbitrary dimension:

$$r_i(t) = o_i + td_i. \tag{2.14}$$

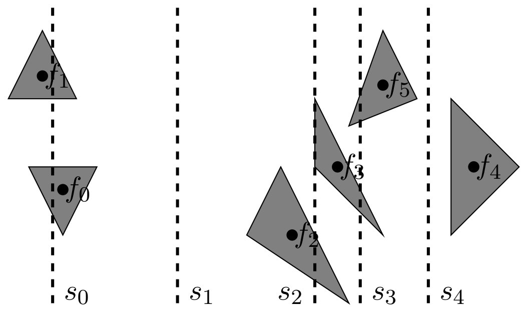
Considering the lower bound $\vec{l}_i$, Eq. (2.14) solves to $t_{l,i} = (\vec{l}_i - o_i)/d_i$ and $t_{u,i} = (\vec{u}_i - o_i)/d_i$ for the upper bound $\vec{u}_i$. For a single bounding slab, the intersection parameter interval is then $[r_i, s_i]$ with $r_i = \min\{t_{l,i}, t_{u,i}\}$ and $s_i = \max\{t_{l,i}, t_{u,i}\}$. The parameter interval of all slabs is then $[t_0, t_1] = [\min\{r_0, r_1, r_2\}, \max\{s_0, s_1, s_2\}]$. Intersecting these types of bounding boxes is essential for most bounding volume hierarchies, described in the following.

2.1.2. Bounding Volume Hierarchies

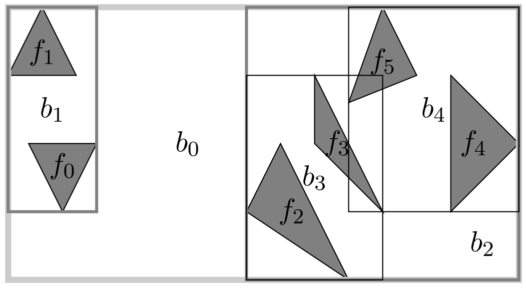
While the previous definition of a ray tracing can be implemented in a straightforward manner, it is rather inefficient, as each ray traverses the list of primitives in Eq. (2.2),



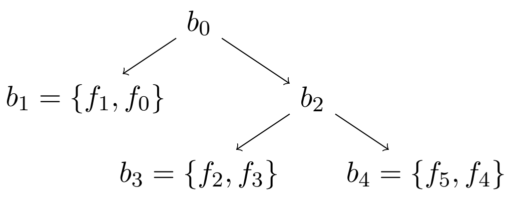
(a) Scene consisting of a set of triangles f_i.



(b) Possible splits s_i along the horizontal axis. Dots are centroids.



(c) Recursive boundaries of AABBs.



(d) Polytree of the corresponding BVH (c).

Figure 2.1.: Example BVH and its construction process over a set of triangles.

which has a computational complexity of $\mathcal{O}(n)$ for the number of primitives. In order to reduce computational complexity, so-called *bounding volume hierarchies* (BVH) have been developed. These prevent intersecting the entire geometry per ray, reducing the complexity to $\mathcal{O}(\log n)$ [Mei+21]. A BVH can be defined as an *acceleration structure*, that exploits the relationship between t and the arrangement of primitives in the scene. It is implemented as a tree data structure that recursively wraps groups of primitives into bounding volumes, where each bounding volume forms a node in the tree. The leaf nodes of the BVH contain the primitives. A bounding volume fully encloses all primitives within. A two-dimensional illustration of a BVH utilizing AABB volumes can be found in Fig. 2.1d and a comprehensive literature review of various types of BVHs was conducted by Meister, Ogaki, Benthin, Doyle, Guthe, and Bittner [Mei+21].

Let $G = (V, E)$ be a polytree with nodes V and arcs $E \subseteq \{(x, y) \in V^2 \mid x \neq y\}$. The path between two nodes a, b is defined as

$$p_{(a,b)} \Leftrightarrow \exists(v_0, v_1, ..., v_n) \in V^n : v_0 = a \wedge v_n = b \wedge (v_i, v_{i+1}) \in E. \tag{2.15}$$

Moreover, the node a is a leaf if

$$l_a \Leftrightarrow \forall(x, y) \in E : x \neq a. \tag{2.16}$$

Then, the set of leaves for the subtree based on node v is defined as

$$L_v = \{x \in V \mid p_{(v,x)} \wedge l_x\}. \tag{2.17}$$

As a last step, we define the set of all rays that intersect against a primitive p as

$$R_p = \{(\vec{o}, \vec{d}) \in (\mathbb{R}^3, \mathbb{R}^3) \mid T_p(\vec{o}, \vec{d}) \textit{ is defined}\}. \tag{2.18}$$

The previously mentioned property, that all leaves are recursively enclosed by their parent volumes can be formalized as

$$\forall v \in V : \forall l \in L_v : R_l \subseteq R_v. \tag{2.19}$$

It is typical for bounding volumes to be convex, as more sophisticated bounding volumes tend to be more computationally inefficient to intersect. However, as long as Eq. (2.19) holds, arbitrary types of bounding volumes are permitted.

Traversal Let $T_a(\vec{o}, \vec{d})$ be a recursively defined function that intersects node $a \in V$ and all its children, while $r \in V$ is the root of the BVH.

Proposition 1. *For each subtree a, the BVH traversal corresponds to the intersection by naive iteration all primitives of the subtree's leaves L_a.*

$$\min_{l \in L_a} T_l(\vec{o}, \vec{d}) = T_a(\vec{o}, \vec{d}) \tag{2.20}$$

Especially, if $a = r$, then $T(\vec{o}, \vec{d}) = T_r(\vec{o}, \vec{d})$ follows from Eqs. (2.2) *and* (2.20).

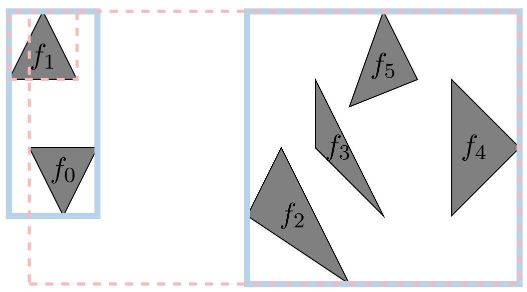
Proof. The base case is if l_a applies, then node a contains only primitives and the overall intersection corresponds to intersection of a list of primitives $T_a(\vec{o}, \vec{d}) = \min_p T_p(\vec{o}, \vec{d}) = T(\vec{o}, \vec{d})$. For an inner node a with children $(a, c_i) \in E$, the equality $T_{c_i}(\vec{o}, \vec{d}) = T(\vec{o}, \vec{d})$ is a direct implication of Eq. (2.19). □

Construction BVHs are typically constructed through recursive space splitting of ordered primitives. Considering the example scene comprising six triangles in Fig. 2.1a. Primitive sorting is a crucial initial step in BVH construction. This process involves arranging the primitives in a specific order, typically based on their centroids along each axis. This approach is exemplified in Fig. 2.1b, which depicts the sorting of the primitives in the aforementioned example. Given this sorting, there are multiple options for potential splits s_i, that subdivide the scene into two parts. One possible strategy is using median splits, which always choose the split axis at the median, s_2 in our example. However, a natural goal of BVH construction is minimizing the empty space in a BVH, making s_1 a more optimal choice here. We describe strategies that can adapt to more optimal metrics in Section 2.1.3. Given split axis s_1, two bounding volumes are created around both separated parts as illustrated in Fig. 2.1c. The construction then recurses on both parts until termination criteria are met, in our case when less than three triangles in a leaf are reached. The corresponding BVH polytree, which is actually represented in memory, can be observed in Fig. 2.1d.

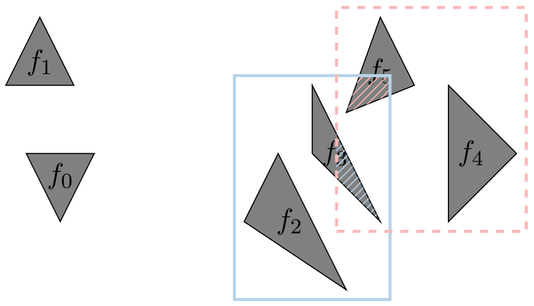
2.1.3. BVH Quality Metrics

Surface Area Heuristic The *surface area heuristic* (SAH), described by MacDonald and Booth [MB90] and Goldsmith and Salmon [GS87], is based on the idea that probability of hitting a node $n \in V$ of the BVH is proportional to the surface area of n [SK04]. According to Wald, Mark, Günther, Boulos, Ize, Hunt, Parker, and Shirley [Wal+07], given n is known to be intersected, the probability of intersecting a child c of that node $(n, c) \in E$ is based on the fraction of their surface areas A_c and A_n

$$p(c \mid n) = \frac{A_c}{A_n}. \tag{2.21}$$



(a) Visualization of the surface area evaluated for SAH for two splits from Fig. 2.1b s_0 (dashed red) and s_1 (solid blue).



(b) The End-Point Overlap for bounds b_3 and b_4 from Fig. 2.1c illustrated as the pattern within the other bounding volume.

Figure 2.2.: Two BVH quality metrics illustrated on the scene from Fig. 2.1.

Let $\mathcal{L}$ be the set of primitives within the node n that are on the left side of a split s_i, meaning the centroids of the primitives c_p are smaller than the split $c_p < s_i$, and $\mathcal{R}$ all remaining primitives within that node, as illustrated in Fig. 2.2a. Moreover, c_p is the cost of intersecting a primitive and c_v is the cost of intersecting the volume of an BVH node. Based on these cost parameters, the cost c_n of intersecting that node is defined as

$$c_n = c_v + c_p \left(|\mathcal{L}| \sum_{c \in \mathcal{L}} p(c \mid n) + |\mathcal{R}| \sum_{c \in \mathcal{R}} p(c \mid n) \right) = c_v + \frac{c_p}{A_n} \left(|\mathcal{L}| \sum_{c \in \mathcal{L}} A_c + |\mathcal{R}| \sum_{c \in \mathcal{R}} A_c \right). \tag{2.22}$$

Two distinct split configurations are illustrated in Fig. 2.2a, with the red-dashed marked configuration exhibiting a markedly inferior efficiency compared to the blue solid-line marked configuration. This is due to the greater total surface area of the former, which results in a greater cost function. Given this cost term and further incorporating the cost of intersecting primitives, we can evaluate the SAH cost C_{SAH} for the whole BVH, which roughly corresponds to the work of MacDonald and Booth [MB90]. Here, N_n corresponds to the number of primitives in a leaf $n \in L \subseteq V$ and A_r represents the surface area of the root:

$$C_{\text{SAH}} = \frac{1}{A_r} \left(c_v \sum_{n \in V} A_n + c_p \sum_{n \in L} N_n A_n \right). \tag{2.23}$$

End-Point Overlap Aila, Karras, and Laine [AKL13] describe the quality metric *end-point overlap* (EPO). It is employed to evaluate the surface area of primitives O that are geometrically within a BVH node, but not logically within the subtree of that node. The objective of this metric is to penalize overlapping regions that may result in the traversal of both BVH nodes for a single ray. Let S be the set of all primitives and A_S the total surface of these primitives $\sum_{p \in S} A_p$, then

$$C_{\text{EPO}} = \sum_{n \in V} c_n \frac{A_{O_n}}{A_S} \tag{2.24}$$

is the EPO cost that relies on the set of previously mentioned nodes $O_n = (S \setminus L_n) \cap n$, where L_n is the set of primitives within a subtree of a node n. The area of this overlap is depicted in the pattern-shaded regions in Fig. 2.2b. While the SAH allows for the iterative construction of a BVH using Eq. (2.22), the EPO metric necessitates the construction of the BVH in advance. One possible approach to integrating the EPO into BVH construction is to utilize temporary subtrees [WG17]. The correlation of a combination of SAH and EPO to the performance of a ray tracer averages to 0.979, while for SAH alone it averages to 0.933 [AKL13].

2.2. Compression in Computer Graphics

Ray tracing is based on several data structures, which store information about the scene and acceleration structures. This section describes simple formations of these structures, which already vary in entropy. It then continues with compression techniques of scene primitives and the BVH structure.

2.2.1. Polygonal Mesh Formation

A mesh consists of a set of vertices defining the geometry and their connectivity. Mathematically, a mesh can be seen as a planar graph $G = (V, E)$ and its embedding F. V, E, and F have the relationship $|V| - |E| + |F| = 2$ for bounded convex polyhedrons [Dae09]. A typical data structure that implements fast polygonal mesh connectivity queries is the *half-edge data structure* [Ber+08], illustrated in Fig. 2.3. It replaces each undirected edge $\{a, b\} \in E$ in G with two arcs $(a, b) \in E'$ and $(b, a) \in E'$ in the directed graph $G' = (V, E')$. This happens in a way that the edge adjoins two embeddings such that a directed cycle of arcs $f_i = \{\{(v_0, v_1), (v_1, v_2), ..., (v_n, v_0)\} \subseteq E'\}$ forms an embedding. On boundaries, $\{a, b\}$ is only replaced by the single arc of the corresponding embedding. A mesh is considered *two-manifold*, if each half-edge $(a, b) \in E'$ is an element of exactly one embedding f_i. If a half-edge is an element in multiple embeddings f_i, it is considered *non-manifold*. Based on G', the half-edge data structure defines the following essential

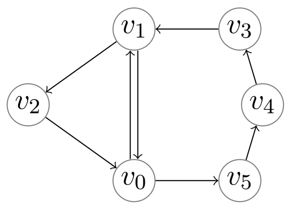


Figure 2.3.: Illustation of the half-edge data structure using a polygonal mesh consisting of a triangle $\{(v_0, v_1), (v_1, v_2), (v_2, v_0)\}$ and a pentagon $\{(v_1, v_0), (v_0, v_5), (v_5, v_4), (v_4, v_3), (v_3, v_1)\}$. The origin vertex of an edge is marked as the vertex at the origin of the arrow. The next edge can be found by following the path on a cycle of arrows one step. The only twin edge is between v_0, v_1 and highlighted by two arrows pointing in opposite directions.

functions. Common half-edge functions like the destination vertex or the previous edge can be trivially derived from the essential ones.

Next Edge Let $n_i : f_i \to f_i$ be a function that returns the next edge of a polygon f_i with definition

$$n_i(v_0, v_1) = (v_1, v_2). \tag{2.25}$$

Twin Edge Let $t : E \to E$ be a function that returns the adjacent edge with definition

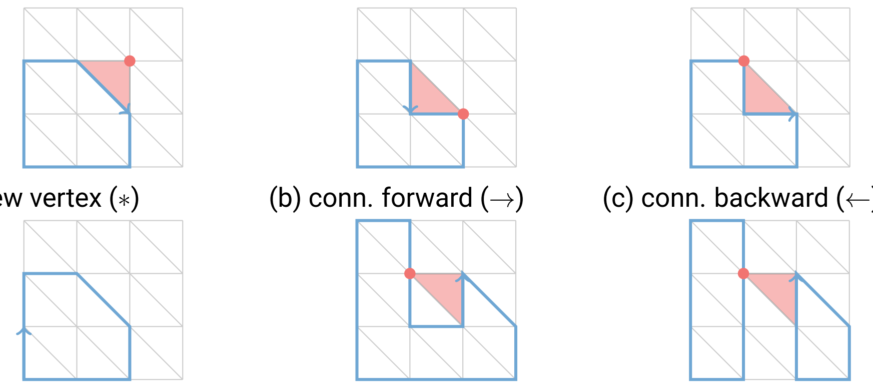
$$t_i(v_0, v_1) = (v_1, v_0). \tag{2.26}$$

Origin Vertex Let $o : E \to V$ be a function that returns the origin vertex of an edge with definition

$$o(v_0, v_1) = v_0. \tag{2.27}$$

2.2.2. Mesh Representations

One fundamental method of compression is the application of distinct standard mesh representations. The subsequent section presents three prevalent types of mesh representations.


(a) new vertex ($*$) (b) conn. forward ($\rightarrow$) (c) conn. backward ($\leftarrow$)

(d) border (_) (e) split$_i$ (∞_i) (f) union$_{p,i}$ ($\cup_i^p$)

Figure 2.4.: Illustation of cut-border operations [Gum99] on a grid-structured triangle mesh. The cut-border is highlighted as a bold blue cycle and the gate is marked as an arrow. The current triangle is shaded in light red and its corresponding adjacent vertex is represented by a red dot. It should be noted that the illustration of the union operation (f) is intended only as an example, as it does not result from a natural traversal using the cut-border machine, which requires a geometrical genus.

Triangle Array The *triangle array* encodes each triangle explicitly as the tuple (v_0, v_1, v_2) consisting of the three corner vertices v_i. Despite its simplicity, this representation is known to be storage-inefficient, as vertices are encoded repeatedly depending on their valency in a triangle fan. Furthermore, this representation lacks connectivity information, which must be inferred from geometry.

Indexed Triangle List The *indexed triangle list* encodes a triangle as the tuple (i_0, i_1, i_2), where $i \in \mathbb{N}$ is an index to a list of vertices $\{v_i \mid i \in \mathbb{N}\}$. This representation offers two significant advantages. First, each vertex is stored only once, reducing the storage requirements. Second, the connectivity can be inferred from vertex indices. From a computational perspective, the indexed triangle list introduces a secondary indexing layer to the representation, necessitating further loading of the index. However, this secondary indexing layer has a positive effect on caching due to more frequent accesses on a single vertex stored in memory.

Triangle Strips A *triangle strip* is a data structure that encodes multiple triangles that share connectivity in the form of a strip. It is defined by the tuple $(v_0, v_1, ..., v_n)$, which has the property that each combination $(v_0, v_{i+1}, v_{i+2}), i \in [0, n-2]$ forms a triangle. Triangle strips can be considered a synthesis of the strengths and weaknesses of indexed triangle lists and triangle arrays as the secondary indexing layer is omitted and redundancies are partly avoided. If the mesh consists of polygons with a higher number of edges than three, triangle strips can be employed to encode these polygons [BGG17]. A strip itself is always defined to be a two-manifold.

2.2.3. Cut-Border Machine

The *cut-border machine* (CBM) of Gumhold and Strasser [GS98] is a single-rate mesh connectivity compression approach that employs a set of symbols to unambiguously encode a depth-first traversal of a triangle mesh. It is coarsely related to the triangle strip representation from Section 2.2.2 and is therefore limited to two-manifold meshes. The internal state of the CBM is a list of *cut-border parts*, which store edges that separate between the already encoded region of the mesh and the outstanding region as shown in Fig. 2.4. A cut-border part employs a stack data structure [Lan98] along with additional non-essential operations for splitting and merging of multiple stacks. The traversal commences with an arbitrary triangle, which is encoded and initialized as a new cut-border part, along with its edges. Subsequently, the CBM iteratively examines the top edge of the current part, designated as the *gate*. This examination identifies six distinct operations given the vertex that is adjacent to the gate, which are illustrated in Fig. 2.4. Each operation is represented by a unique symbol.

New Vertex The *new vertex* operation is encoded when the adjacent vertex is not yet part of any cut-border part. To avoid traversing all cut-borders for identifying this operation, a per-vertex map can be used to flag each vertex if it was already encoded. Furthermore, the new vertex operation encodes attributes corresponding to the adjacent vertex.

Connect Forward The *connect forward* operation is encoded if the adjacent vertex is the next element of the current cut-border part. This operation does not initiate encoding of a vertex attribute, as all vertices of this triangle were already encoded.

Connect Backward The *connect backward* operation is similar to the connect forward operation, except that it checks, if the adjacent vertex is the previous element of the current cut-border part. As the gate is always located at the top of the stack, this effectively means that it checks for the last element on the stack.

Border The *border* operation was introduced to allow for encoding meshes with holes. It encodes that the gate has no adjacent vertex due to the connectivity.

Split The *split* operation is encoded if the adjacent vertex is on the current cut-border part, but neither the previous nor the next one. In order to permit the reversal of this operation during decoding, it encodes the offset within the current cut-border part's stack. This operation naturally creates a hole into the current cut-border part as edges of the adjacent vertex shares no connectivity to the cut-border. The cut-border machine handles this case by moving all edges within the hole to a new part to continues traversal on that new part.

Union The *merge* operation reverses a split. It is encoded if the adjacent vertex is on another part. Therefore, the index of the part must be encoded additionally. A merge operation gets only encoded if a mesh has a genus and the number of merge operations is always equal to the genus [GS98].

2.2.4. Structured Primitives

A recent approach to compressed ray tracing involves the use of structured primitives on simplified meshes [MMT23]. One advantage of this approach is that structured primitives have a smaller memory footprint, as their topology can be described implicitly. The concept itself is not a recent innovation; it has its roots in the definition of displacement maps [Coo84] and subdivision surfaces [CC78]. These techniques assume a fixed structure that can be refined recursively using geometry from neighboring vertices and continuity assumptions. Although this traditional representation can be traversed on the GPU [Ben+07], it tends to be very inefficient due to the explicit storage of intermediate primitives on a traversal stack. Consequently, further caching implementations are required. Displacement maps, however, have the advantage that they are not necessarily defined recursively and can therefore be implemented efficiently on a GPU [Tho+21].

More recent approaches, however, restrict to static structured grid representations that encode geometry for each vertex in the grid [SB87]. Typical grid structures include quads [BVW21] and triangular grids. Ray tracing of triangular grid primitives is also referred to as *micro mesh* ray tracing. This technique can be implemented efficiently on a

GPU [BP23; NVI23; MMT23] using specialized hardware, which limits its availability to end-users.

2.2.5. Compressed BVH

BVH compression is typically achieved by quantizing the values of the bounding volumes. This can be accomplished in a hierarchical manner, with each BVH node is quantized in relation to its parents [SE10; WMZ22]. While quantizing the BVH volumes reduces memory traffic, it naturally increases the EPO (Section 2.1.3), which in turn makes tracing less efficient. A further disadvantage of a recursively quantized BVH is that reconstructed nodes potentially need to be stored on the traversal stack, which increases incoming memory traffic by magnitudes of order. Moreover, the data structure of a BVH itself can also be compressed by squashing multiple levels of binary nodes into multi-nodes [Ben+18]. An alternative approach is to store the BVH implicitly by rearranging the geometry in memory in such a way that primitives can be used to describe the bounds of BVH nodes [Eis+12]. While hybrid representations do exist, the complete omission of all geometric information makes it impossible to use efficient heuristics like the SAH for construction [SE10]. As previously described, traversing the BVH requires a traversal stack, which can also be compressed in order to reduce traffic to the underlying memory [VWB19].

2.3. GPU Architecture

This section presents the essential elements of typical GPU architectures. While the focus is on NVIDIA architectures, the descriptions, particularly those of the logical perspective on units of execution, can be adapted to other platforms, such as those of AMD or Intel. The following employs the CUDA terminology and additionally presents the OpenCL terms where applicable.

2.3.1. Units of Execution

In OpenCL, a GPU is referred to as a *compute device*, which also encompasses accelerators such as *field-programmable gate arrays* (FPGAs). A GPU can be logically and physically subdivided into multiple layers of execution units. A program for a GPU is comprised of two distinct components: the program that is executed on the GPU, namely the kernel, and the program that is executed on the host system to manage memory initialization and program execution. Consequently, the driver API utilizes a *stream* (referred to as a *command queue* in OpenCL) to transmit kernel code to the GPU.

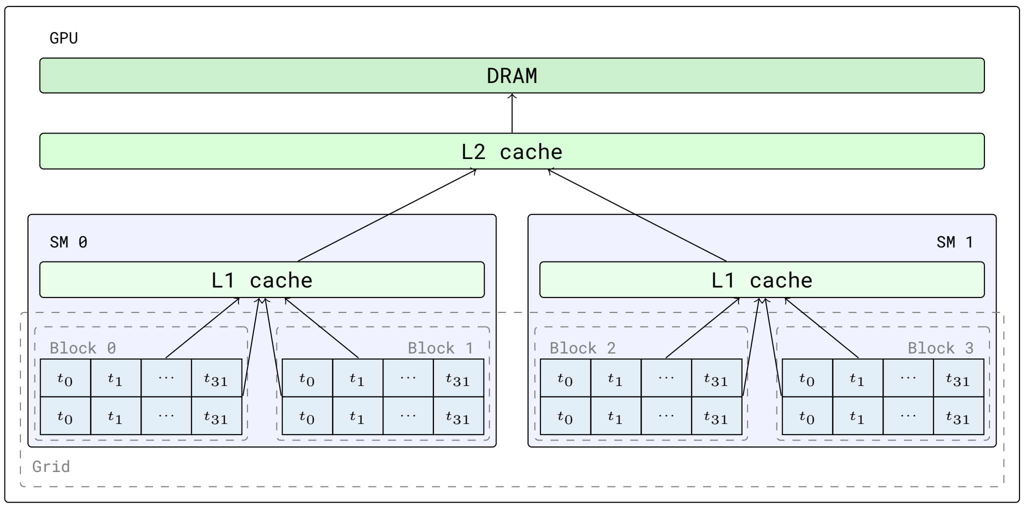


Figure 2.5.: A diagram of most NVIDIA architectures. Hardware components are indicated by solid black borders, and memory communication with arrows. The memory itself is shaded in green and execution units are marked in blue. Software-side concepts are indicated by dashed gray borders. Each row of 32 threads is referred to as a *warp*.
Adapted from the work of BUELOW, RIEMANN, GUTHE, and FELLNER [Bue+22a].

Logical Layers Logically, the kernel is executed on a *thread* (*work-item* in OpenCL), which follows the *single instruction multiple thread (SIMT)* extension of the SIMD paradigm of FLYNN's taxonomy [Fly66]. In the SIMT paradigm multiple SIMD operations get executed in lock-step [MRR12]. Multiple threads are grouped as a *block* (*work-group* in OpenCL) and multiple blocks form a *grid* (*n-d range* in OpenCL). Both groupings have to be manually configured and are exposed to the kernel using a *block index* (*global ID* in OpenCL), which describes the index of a block inside an grid and a *thread index* (*local ID* in OpenCL), which describes the index of a thread within a block. This two-level grouping is crucial for inter-thread communication, which is implemented using per-block shared memory (Section 2.3.2) and synchronization. Although the grid has no practical limitations in terms of dimensionality, the dimensionality of a block is constrained by hardware limitations, such as the availability of shared memory and hardware registers. This is due to the fact that all threads within a block are defined to be processed concurrently.

Physical Layers Physically, the bottom layer of a GPU are *warps* (*wavefronts* in OpenCL), which implement the *SIMD* model. For NVIDIA architectures, each warp holds 32 *lanes*, which are directly mapped to threads. Warps are part of a *streaming multiprocessor* (SM; *compute unit* in OpenCL) and blocks are permanently mapped onto an SM due to their requirement of concurrent processing. Because modern GPUs usually have multiple warp schedulers, they themselves operate in the MIMD model. The set of individual compute units define the compute device.

2.3.2. Memory Spaces

Device Memory and L2 Cache *Device memory* is a physical off-chip *dynamic random access memory* (DRAM) that is used to exchange data with the host, other kernels or for the usage of temporary memory. Memory transfers from the host are implemented using *direct memory access* (DMA) of the CPU DRAM via the *Peripheral Component Interconnect Express* (PCIe) bus. Similarly to CPU DRAM, the device memory is wrapped by the L2 cache that caches physical memory addresses. A cache, as illustrated in Fig. 2.6, in general, is a smaller but faster memory that maps frequently accessed addresses from the underlying memory given a replacement and write policy. A common example of a replacement policy is the *least recently used* (LRU) policy, typically implemented as a stack data structure that replaces the least recently accessed cache line when the cache becomes full. The L2 cache is the connecting point to the L1 caches on the chip. It is implemented with a write-back policy [NVI20], which means that data is initially only written to the cache. Then, the cache line is marked with a dirty bit, indicating that it must be written back to

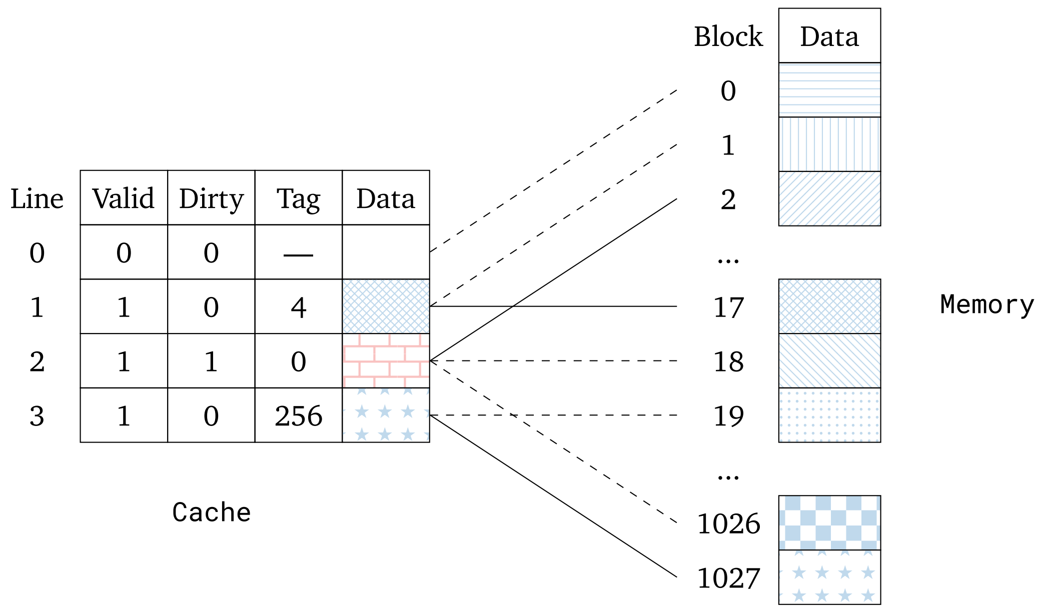


Figure 2.6.: Diagram of a direct-mapped cache with a write-back policy and four lines. The lines indicate the mapping between cache lines l maps and memory blocks b, computed as $l = {}^{b}/_{4}$. The solid lines indicate whether a block is currently in the cache. Cache line two is currently marked as dirty because the cache line received a write. The valid bit indicates whether a cache line contains data, while the tag t represents the most significant bits of the mapped address, calculated from the block as $t = b \mod 4$.

main memory upon eviction. The advantage of the write-back policy is that writes benefit from the faster cache memory. Additionally, it is possible to use the write-back policy for the L2 cache, as the L2 cache does not require synchronization with any caches on the same layer. This eliminates the need to treat dirty cache lines. In contrast to the L1 cache, the L2 cache is completely opaque to the user. The majority of parameters pertaining to the caching architecture, with the exception of the cache capacity, are typically kept secret by device manufacturers. However, some details can be revealed through the use of fine-grained microbenchmarking [MC17].

L1 Cache In newer NVIDIA architectures, the L1 cache is partially transparent to the programmer. While there is a region that can be freely addressed from the kernel (see *shared memory* below), a further region can be used as a standard cache. The L1 cache is located on each SM separately, necessitating a write-through policy [NVI20] with write masks. This ensures that data is directly written to the underlying memory layer to maintain data consistency. However, the data written to the cache is defined to be not consistent between SMs [NVI22]. A line in the L1 cache is defined to be 128 B for NVIDIA architectures. The ratio between shared memory and L1 cache can be configured on recent NVIDIA devices.

Global Memory *Global memory* represents the most generic logical abstraction of device memory. A single memory referencing instruction, a *memory request*, may encompass 1 B, 2 B, 4 B or 8 B of aligned virtual memory per thread [NVI22]. This results in 32 requests in total per warp. These 32 memory regions may be aligned in a sequence or overlap in the optimal case, although this is not a requirement. In order to minimize physical memory accesses to device memory, i.e. *memory transactions*, the memory management unit automatically coalesces per-instruction addresses into fewer sequential transactions. The maximum size of a memory transaction depends on the architecture, but is usually up to 128 B. An illustrative example of a fully coalesced memory access is that of threads accessing their respective 32 bit floating-point values in an aligned fashion, collectively forming a single memory transaction of 128 B.

Local Memory *Local memory* (*private memory* in OpenCL) is an abstraction of global memory and therefore also resides in device memory. Its purpose is to act as further temporary storage space when the amount of available registers is exhausted or the temporary memory needs to be addressed dynamically. The only difference is the logical handling, which requires separate uniform memory regions for each thread. In the context

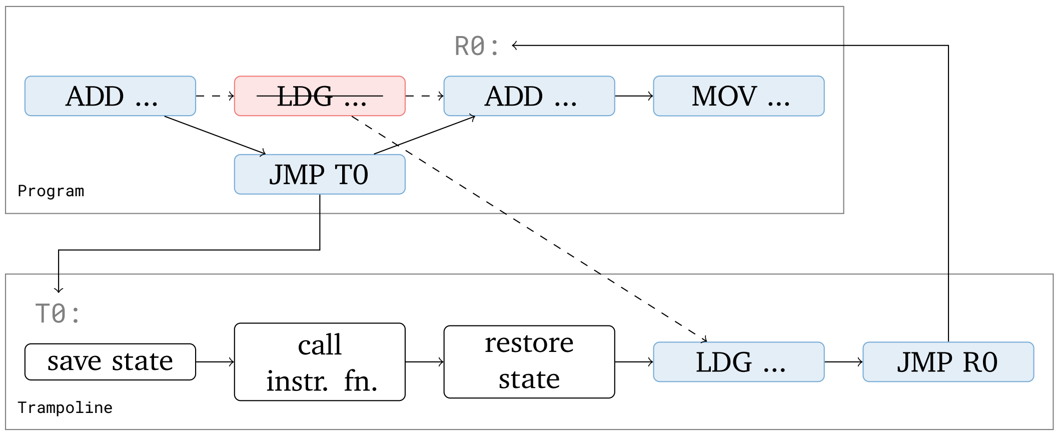


Figure 2.7.: This diagram illustrates the manner in which the trampoline interacts with the original program in dynamic binary instrumentation. The program at the top is a stream of exemplary low-level assembly instructions where the LDG is the instruction that should be instrumentated, replaced with a jump to label of the trampoline T0.

of ray tracing (Section 2.1), the dynamically addressed traversal stack, which frequently exceeds the capacity of registers, is typically stored in the local memory.

Shared Memory The concept of *shared memory* (*local memory* in OpenCL) represents an abstraction of the underlying L1 cache memory. It is employed primarily for the purpose of facilitating data exchange between threads within a block, which is defined as a unit of execution that is permanently bound to a single SM.

2.3.3. Binary Instrumentation

The technique of *binary instrumentation* allows for the alteration of the operational behaviour of individual compiled applications. While a principal objective is the profiling of applications [Luk+05], it can also be employed for the purposes of validation, hardware analysis and educational or scientific purposes [Red+04]. In general, binary instrumentation can be distinguished into two categories. The first category is *static binary instrumentation* (SBI) [ZS15; Zha+13], which hooks between the compile- and run-time of a program. In contrast to that, the following focuses on *dynamic binary instrumentation* (DBI), which analyses and manipulates code during runtime. Two common DBI frameworks for CPU binaries are *Intel Pin* [Luk+05] or *DynamoRIO*. Recent developments include *NVBIT* [Vil+19], which brings the technique to GPU architectures. NVBIT uses a automatically generated *trampoline* code and replaces the instruction that was selected during prior code analysis with a jump instruction to said code region. This process is illustrated in Fig. 2.7. In order to retain original program behavior, the original instruction is included in the trampoline. Additionally, the trampoline safely calls the *instrumentation function* by wrapping the call instruction with procedures that back up and restore the program state. The instrumentation function is arbitrary user code that has access to the instruction operands as well as other general-purpose registers of the CUDA architecture. An illustrative example pertinent to this thesis is a memory tracer that selects all instructions with memory references in their operands during program analysis and injects a function that transfers these memory references to host memory for further processing.

2.3.4. GPU Simulation

GPU simulation represents a significant research topic, as it enables the bridging of the gap between the innovations of hardware manufacturers and open research [Kha+20]. The coarsest level of simulation is the *cycle-accurate simulation*. This approach is advantageous, as it allows for the modelling of every part of the GPU and the implementation of performance evaluation directly in the hardware design at arbitrary points. An example of a cycle-accurate GPU is the *Nyuzi* [Bus+21] GPU. It is an open source GPU that comprises fully synthesizable *hardware description language* (HDL) code. While it has basic rasterization and computation support, most advanced hardware techniques are not comparable to those of modern GPUs [Bus+16]. Its aim is to provide a general-purpose architecture that can be further specialized to more complex use cases. However, one disadvantage is that cycle-accurate simulation is very compute intensive. At a higher level, simulators like *Ocelot* [KDY10], *GPGPU-Sim* [Bak+09] and *Accel-Sim* [Kha+20] aim to simulate GPU programs at the instruction level on CPU architectures. In the case of CUDA, the *PTX instruction set architecture* (ISA) is transformed into an intermediate assembly (*LLVM* in Ocelot) that can be further transformed to numerous architectures. Moreover, these simulators separate the API from the driver by wrapping it with a reimplementation. Simulators such as *MCUDA* [SSH08] have a comparable fundamental concept but compile the CUDA code directly into CPU binaries and specify additional concepts for synchronization. The configuration parameters required for different models, which in turn depend on industry secrets, vary according to the simulation level. In this context, micro-benchmarking [MC17] represents an important technique for estimating the aforementioned configuration parameters, including those pertaining to cache parameters.

The memory is a crucial component for GPU performance, given that many applications are memory-bound, partial simulation of the GPU, i.e. the memory pipeline, may be

Time step	0	1	2	3	4
Reference	0	1	1	2	0
Reuse distance	∞	∞	0	∞	$\boxed{2}$

Table 2.1.: Exemplary reuse distance calculations given their access time and reference. The time step 4 illustrates 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.
Adapted from the work of BUELOW, RIEMANN, GUTHE, and FELLNER [Bue+22a].

a feasible approach. The simulation of the memory pipeline necessitates the actual list of *memory transactions* to the memory system. In order to derive the actual list of these transactions from the list of memory requests, address calculations must be performed [BGF22]. The list of memory requests can be extracted automatically using DBI [Ara+20] from actual compiled GPU applications during runtime. While the simulation of the actual main memory is straightforward, caches must be approximated further using reuse distances [Nug+14; TYL11; WX16; Ara+19]. An example of computing reuse distances can be seen in Table 2.1. A technique for approximating and formulating the behavior LRU cache hit rates p_i is the *stack distance cache model* (SDCM, Eq. (2.28)) of AGARWAL, HENNESSY, and HOROWITZ [AHH89] relying on reuse distances D_i and the cache associativity A and the number of cache blocks B:

$$p_i = \sum_{a=0}^{A-1} \binom{D_i}{a} \left(\frac{A}{B}\right)^a \left(1 - \frac{A}{B}\right)^{D_i - a} \tag{2.28}$$

The aforementioned simulation results can then be utilized for the purpose of performance visualization, as will be demonstrated in the following section.

2.4. Performance Visualization

Visualization can be divided into two categories: *scientific visualization* and *information visualization* [Nag06]. While physically-based ray tracing (Section 2.1) is a prominent example of scientific visualization, where natural phenomena are simulated, information visualization aims to visually prepare information for highlighting interesting phenomena. Performance visualization therefore covers both [Sch+11]. This section provides an

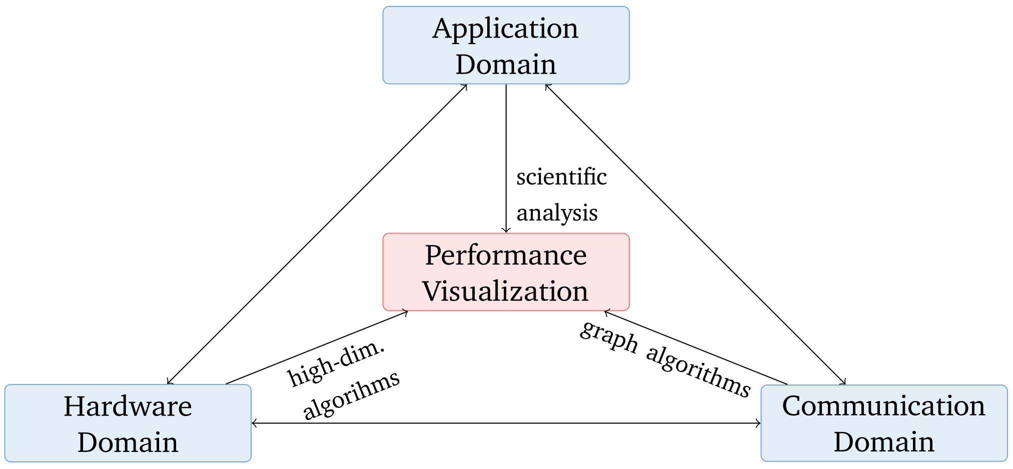


Figure 2.8.: The HAC model comprises three domains, each marked with a blue node, which can be mapped to one another using domain transformations, represented by arrows. Arrows pointing to the centre represent the information visualization technique.
Derived from the work of Schulz, Levine, Bremer, Gamblin, and Pascucci [Sch+11], © 2011 IEEE

overview of the fundamentals of information visualization and its relationship to scientific visualization in the context of performance information.

2.4.1. Information Visualization

Brehmer and Munzner [BM13] introduce a multi-level typology that separates information visualization into three parts: the *why* part, the *how* part, and the *what* part. The purpose of this typology is to characterize visualization tasks using a lexicon of smaller tasks that aim to help understand complex visualization tasks.

In practice, the why part of a visualization is the first aspect to be considered. This is because a practitioner must first decide why a visualization task is performed before asking how it is done and what is required [BM13]. This is in accordance with the typology, which divides the why part into multiple levels of specificity. The highest level of specificity is the *consume* purpose, which describes whether a visualization is consumed in the form of a *presentation*, for *discovery* or simply to let the user *enjoy* the visualization. In the next level, users should *search* within the visualization. The variable of search can be divided into two categories: if the target is known and if the position in the visualization is known. The former specifies whether a search is performed by *looking up* the target, *locating* the target, by *browsing* or *exploring*. In the final level of specificity, users found a potential target which they either *identify* or *compare* against another target or *summarize* the contents of a visualization.

The *how* part describes methods of visualization or interaction, which can be categorized into two main types: bare visual *encoding* of the initial information, the *manipulation* of the visualization, and *introduction* in the context of interactive visualization. Finally, the *what* part describes inputs and outputs of the visualization, which can be specified in a freeform textual way within the scheme.

2.4.2. The HAC Model

The *HAC model* has been described by Schulz, Levine, Bremer, Gamblin, and Pascucci [Sch+11]. It distinguishes performance data into three domains as seen in Fig. 2.8. These domains are the application domain, the hardware domain, and the communication domain. Performance data in the application domain represents the outcome of a physical model itself implemented by the domain expert. In contrast, performance data in the hardware domain is measured at the level of computing cores. The communication domain is measured in network links. Defining and understanding these domains has the advantage that inter-domain transformations can be built on top of them, allowing for existing tools to use or combine performance data from other domains. The work of Isaacs, Giménez, Jusufi, Gamblin, Bhatele, Schulz, Hamann, and Bremer [Isa+14] presents additional layer abstractions, with the software layer being of particular importance. This layer can be seen as a layer between the application and hardware layers, capturing performance information at the level of the source code.

Application Domain The application domain describes the intended behavior of a program itself. As an illustrative example, consider a physical simulation that measures stress on a mechanical object. In this context, ray tracing represents an important technique for visualization. In this case, the stress itself corresponds to the performance data. Performance data in the application domain is visualized using scientific visualization.

Hardware Domain The hardware domain describes the behavior at the layer of compute cores. In the previous example, the physical simulation must be executed on a computing device, which is a CPU or GPU core and its corresponding memory interface in practice. This hardware can provide further information, such as the temperature of the compute core, the number of intrinsic operations, or the cache hit rate. As the actual application is typically of a higher dimension than the number of hardware cores, a dimensionality reduction is required to map from the application domain to the hardware domain. However, the opposite direction is less straightforward and requires further information that allows upsampling and distributing the data back into the application domain. One potential solution to enable such transformations is to simulate the hardware in order to retain mappings from the application layer in simulated hardware operations [Bue+22b].

Communication Domain The communication domain describes the behavior at the layer of a network link. To illustrate, we extend our example to run on an *MPI* cluster, which handles multiple computing cores communicating with each other using standard network links. Each MPI instance or network switch can further measure the amount of communication between two cores. The problem decomposition is typically performed in the application code itself, where the transformation can be applied. The communication domain is of lesser importance in this thesis, as our objective is to measure memory behavior, which is largely independent of communication in our assumed single-device setup.

3. Papers and Contributions

This chapter summarizes the works and their contributions attached in Chapters 5 and 6. The works are presented in a the order corresponding to their relationships described in Section 1.2.

3.1. Fine-Grained Memory Profiling of GPGPU Kernels

This section summarizes the paper

M. von Buelow, S. Guthe, and D. W. Fellner
"Fine-Grained Memory Profiling of GPGPU Kernels"

which was published in *Computer Graphics Forum 41.7 (2022): Pacific Graphics*.

3.1.1. Motivation and Contributions

Standard general-purpose GPU profilers, such as *NVIDIA Nsight Compute*, typically evaluate hardware performance counters available at the granularity of a kernel call, displaying cache hit rates, memory transactions, and other memory-related metrics to the user. With the support of these metrics, software engineers aim to enhance inefficient parts of their software programs. This is a standard method for identifying potential bottlenecks in GPU applications. However, the exact cause of inefficiencies may not be apparent to software engineers. Finding and optimizing these inefficiencies requires a deep understanding of both the hardware and the program itself. In many cases, a finer-grained representation of memory-related profiling metrics would be useful. This representation would record memory metrics for each memory allocation separately, allowing the user to narrow down the problem to specific regions of the program where the allocation is used.

This paper introduces *profiner*, a profiling pipeline that automatically derives fine-grained memory profiles from compiled GPU applications during runtime, similar to existing profilers. The profiler extracts a list of memory references from the target application and simulates the memory flow from the processor through the caches to the DRAM. Realistic hardware parameters from NVIDIA GPU architectures, extracted using previous micro benchmarks, are relied upon and modeled in our simulator.

In summary, our contributions are:

- A profiling tool that displays the cache hit rates and coalescing behavior for each allocation individually.

- A detailed model of recent GPU architectures' memory, which outperforms state-of-the-art memory models in terms of accuracy when compared to actually measured profiling metrics.

3.1.2. Methods and Findings

The modular pipeline comprises five essential components. The method uses dynamic binary instrumentation to extract a list of memory requests. Then, it simulates coalescing behavior, resulting in a list of memory transactions (Section 2.3.2). Based on this, all L1 caches can be simulated, followed by the global L2 cache annotating each memory access as a hit or a miss. Finally, the results are accumulated to the desired granularity.

In more detail, the profiler's foundation is the extraction of memory requests using the dynamic binary instrumentation tool NVBIT [Vil+19]. The binary instrumentation tool is instructed to extract all 32 addresses of each thread within a warp, including their predicates (marking currently inactive threads), the opcode of the memory instruction used to derive the number of bytes processed by the memory instruction, and processing hardware identifiers (SM and warp) from the GPU kernel. The profiler streams the data to the host system for further processing. Accesses to global memory can be used directly for subsequent steps, except for accesses in the local memory space (Section 2.3.2). The GPU architecture strides local memory into $4\,\mathrm{B}$ segments in a manner that each $32\,\mathrm{bit}$ word is referenced by successive threads, making coalescing more efficient [NVI22]. The striding requires splitting up memory accesses greater than $4\,\mathrm{B}$. Moreover, we discovered that the extracted local memory addresses are offsets within a unique virtual memory area that does not distinguish between threads. This can give the impression that each thread is accessing the same memory area from a simplistic viewpoint. To deal with this fact, our simulator subtracts the local memory base address b from each reference r. Then, our simulator recalculates the actual memory address using the number of warps per SM n_w, the allocated local memory per thread L, the thread index within a warp i_t, the warp index within the SM i_w, and the SM index i_s:

$$r_{new} = (i_s \cdot n_w + i_w) \cdot 32 \cdot L + (\lfloor r/4 \rfloor \cdot 32 + i_t) \cdot 4 + (r \bmod 4) \tag{3.1}$$

Based on global and transformed local memory requests, we simulated coalescing as described in Section 2.3.2. Our recording of memory requests was confirmed to be correct as the number of requests only differed by approximately $0.001\,\%$ compared to accurately recorded hardware counters. We assume that this small deviation remains from synchronization barriers we experienced when issuing internal memory requests, which NVBIT does not capture. The number of transactions varies by approximately $0.28\,\%$. It is assumed that this small deviation is due to rare side effects in the hardware. As the deviations are negligible, we conclude that the general idea of our coalescing model is correct. We will use the generated list of transactions for further cache simulation.

The L1 cache plays a crucial role in cache rate simulations since it directly affects L2 cache rates. Only memory references that miss the L1 cache are forwarded to the L2 cache. Therefore, our goal is to utilize realistic models and parameters for our target architecture. Using parameters from micro-benchmarking [Jia+19], we aim to model a non-LRU replacement policy. The system consists of $L = 32\,\mathrm{B}$ lines arranged in four sets S, with a total capacity C of $57\,\mathrm{kB}$ in its default configuration (i.e., no shared memory is used). Using the given values, we can calculate the associativity A (i.e. the number of cache ways; in this case, 456) of the L1 cache as $A = {}^{C}/_{(L \cdot S)}$. As sets in the L1 cache evicts four consecutive cache lines at the same time, we refer to them as sub-lines and define our cache differently from a standard set-associative cache. We assumed a fully associative cache with 456 lines of $128\,\mathrm{B}$ each, which can be loaded partially at a granularity of $32\,\mathrm{B}$. The simulator uses a $4\,\mathrm{bit}$ tag per line to indicate the presence of these sub-lines in the cache. Multiple cache replacement policies were experimented with, and a tree-based *pseudo LRU* (PLRU) implementation was ultimately utilized for the L1 cache. This was chosen due to its likelihood of being used in real hardware and because a PLRU cache was found to result in the most accurate cache rate predictions for our set of benchmark applications.

In addition to L1 cache misses, atomic operations and global memory writes have an immediate impact on the L2 cache (Section 2.3.2). However, the local memory does not need to be consistent across SMs since it only stores per-thread data. Therefore, it uses a write-back mechanism between L1 and L2. Our study shows that a standard LRU cache model correlates best to the actual cache hit rates. Our profiler calculates reuse distances (Section 2.3.4) for each memory transaction independently for each set. If the reuse distance is smaller than the associativity, we assume a hit.

In previous steps, the profiler annotated each memory transaction as a hit in either the L1 or L2 cache. With these cache hit assignments, the profiler can perform fine-grained allocation-wise accumulations as desired. The profiler categorizes memory accesses based on their corresponding memory allocation and calculates the conditional probability $p(hit|i)$ that a memory access is a hit, given the allocation i, using the number of memory accesses n_i within the allocation and the number of hits h_i.

$$p(hit|i) = {}^{h_i}/_{n_i} \tag{3.2}$$

3.1.3. Discussion

Comparison of Coarsely Grained Hit Rates To validate the accuracy of our memory simulator, we compared cache hit rates from our simulator with those from the *PPT* model [Ara+20] and the measured rates from NVIDIA Nsight Compute on a *NVIDIA RTX 2080 Ti* GPU. We used a variety of applications from a benchmark set [Gra+12; Che+09; Che+13; Kar+19] that includes kernels of numerical algebra applications, statistical operations, graph and compression algorithms, and machine learning algorithms.

The comparison of our L1 cache simulation and PPT reveals that our implementation performs well on matrix multiplication kernels, as does PPT. However, our L1 cache simulation outperforms PPT in many other kernels. It outperforms PPT with a mean absolute percentage error (MAPE) of $3.73\,\%$, compared to PPT's $9.01\,\%$ on the benchmark applications. It is worth noting that PPT failed to execute on one benchmark due to a segmentation fault, which we did not investigate further. There are cases where our implementation and PPT lead to inferior L1 hit rate approximations, indicating that our implementation does not completely reflect actual memory behavior. Comparing absolute hit rate values reveals that PPT overestimates cache hit rates in many cases, as confirmed by inspecting their faulty approximation of the L1 cache.

The trend in approximating L2 cache hit rates, for both PPT and our implementation, indicates an increase in inaccuracy. This is expected as the L2 cache relies on the L1 cache and therefore accumulates errors. This fact is also evident in the MAPE, which is $15.81\,\%$ for our simulator and $20.10\,\%$ for PPT.

In summary, our cache hit rate approximation technique has significantly improved, particularly for the L1 cache. Our simulator also significantly improved cache hit rate approximations for our ray-tracing implementations, which were the initial reason for implementing it. Our model outperforms PPT because we use a model that is closer to the actual hardware and employs documented and previously estimated cache parameters to model them. Additionally, our model does not rely on the SDCM approximation and instead uses cache implementations to estimate whether a memory reference was a hit. In order to ensure small derivations in our results, we evaluate each step of our profiler separately, including memory reference extraction, coalescing, and address transformations.

Fine-Grained Memory Profiles Our fine-grained memory profiler was applied to individual configurations of our ray-tracing implementation. These configurations contain a large set of actively used allocations within one kernel call due to the structural nature of ray tracing. The general observation from our generated fine-grained profiles is that they confirm our expectations. Manipulating the scheduling such that each SM follows its own scanline area enhances L1 caching and decreases L2 cache rates. Changing the vertex permutation in memory from a spatial sorting to a random one decreases hit rates on vertex allocations. Additionally, median split decreases cache rates on BVH allocations compared to more optimal BVH construction strategies. A short demonstration of the *ResNet* neural network shows how our tool identifies inefficient indexing on a memory region. Improving these aspects made the kernel more efficient.

3.1.4. Individual Contributions

S. Guthe and D. W. Fellner were general advisors of this work and contributed with continuous feedback during all phases of the paper writing process. All remaining work was done by the corresponding and leading author M. von Buelow (leading the overall research design, management, evaluation and writing process, e.g.).

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

This section summarizes the paper

> M. von Buelow, K. Riemann, S. Guthe, and D. W. Fellner
> "Profiling and Visualizing GPU Memory Access and Cache Behavior of Ray Tracers"

which was published in *Eurographics Symposium on Parallel Graphics and Visualization 2022*.

Awards This work received the *EG PGV 2022* best paper award.

3.2.1. Motivation and Contributions

The run-time performance of ray-tracing algorithms depends mainly on the number of rays and the size of the scene. To achieve more realistic rendering, either quantity must increase. Recent developments in the field of GPGPU have enabled efficient execution of ray-tracing algorithms on specialized graphics cards, making real-time ray tracing possible in computer games [Par+10].

The primary bottlenecks in ray-tracing applications are memory latency and work distribution [AL09]. To address the former, GPU vendors have implemented a cache hierarchy similar to that of CPUs. However, this presents a challenge for programmers who must consider the cache structure to optimize program efficiency. The cache hierarchies have become increasingly complex due to SIMD parallelism and separation across different processors on the GPU. 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.

We have developed a two-part toolset that provides visual insight into cache and memory behavior. The initial tool is designed to extract a list of memory accesses from the ray-tracing binary during execution. This is achieved by injecting assembly instructions into the runtime. This tool also simulates the cache behavior and transforms memory accesses into per-element accesses of scene elements. The second tool is a web-based dashboard that displays per-element accesses in a visual format, allowing the user to control the display of temporal data. The dashboard includes a mesh viewer that shows the original mesh, with each primitive represented by a color indicating various metrics, such as the current L1 cache state.

In summary, our contributions are:

- A toolset to simulate fine-grained hardware cache rates during run-time and interactively visualizing them onto the original application data (the mesh and BVH) using a color encoding in a high-performance web-based dashboard.

- 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 framebuffer, allowing insight into scheduling and simultaneously indicating the actual application behavior.

3.2.2. Methods and Findings

Memory Trace Extraction The profiler traces memory references and performs a basic cache simulation. It uses dynamic binary instrumentation (Section 2.3.3) to inject an instrumentation function before each low-level assembly instruction that includes a memory reference operand. For each warp, the instrumentation function extracts 32 memory references r and the SM identifier from the device and streams them to the host system.

To assign references to their corresponding allocation and enable further processing, a programming interface was introduced for the ray tracer. This interface marks allocations such as the list of bounding volumes, vertices, faces, and the framebuffer with a representative name and the size of their elements, denoted as a_e. Using the start address a_s of an allocation and the size of an element, the following formula calculates the element-wise offset o within the allocation: $o = {}^{(r-a_s)}/_{a_e}$.

Our profiler can then compute reuse distances D_i for scene elements based on these offset values for faces and bounding volumes. To simulate LRU-like behavior in L1 and L2 caches separately, we calculate an SM-wise reuse distance for the L1 cache and an SM-independent reuse distance for the L2 cache. Both reuse distances are calculated at the cache-line granularity, i.e., $128\,\mathrm{B}$ for L1 reuse distances and $32\,\mathrm{B}$ for L2 reuse distances.

After calculating all reuse distances, we utilize the *stack distance cache model* (SDCM), as presented in Section 2.3.4 to determine the cache-hit probabilities for each cache line accessed by the element. The SDCM metrics are then averaged [Ara+20] for every cache line accessed by an element. These values are subsequently provided to the visualization dashboard.

Visualization Dashboard The target audience for our dashboard are software architects who create and optimize ray-tracing applications. Their goal is to reduce the number of memory accesses to DRAM by generating more efficient BVHs [Wod+13], improving memory layout [WMZ22], or changing work distribution [AL09; DGP04]. A visualization tool should assist developers in identifying potential memory communication bottlenecks in their ray-tracing applications in a detailed manner. The tool should display profiling metrics, such as L1 and L2 hit rates, that correspond with metrics from standard hardware vendor profilers.

The dashboard quantizes memory accesses into *frames* that visualize a fixed and uniformly distributed number of memory references. The default visualization is a single frame, but the inspector allows the user to select the number of frames q and the currently active frame f to control the visualization in the mesh viewer using a slider. Memory references are distributed equally between frames while retaining their order. This temporal visualization allows for a detailed analysis of caches, which are highly sensitive to the temporal axis based on realistic scheduling.

As previously mentioned, we calculate the SDCM for L1 and L2 cache simulation for each scene element, aggregating multiple accesses to the same scene element within a single frame. In our setting, our visualization represents the conditional probability that a scene element produces hits during its intersection, given it has been accessed.

Our dashboard also allows the user to visualize the access rate and order of memory references, as these values are already implicitly encoded in our memory trace. The order of access $o = {}^{i_e}/_{N_f}$ is determined by the unique ordering index i_e provided by the memory trace, which is normalized by the total number of memory accesses in a frame (N_f). Access rates $r = {}^{N_e}/_{N_f}$ are determined by the number of per-element accesses, denoted as N_e, in the active frame.

3.2.3. Discussion

The algorithm was evaluated on five meshes: the *Asian Dragon*, the *Sponza Palace*, the *Stanford Bunny*, and the *San Miguel* scene. To maintain brevity, we focused solely on single-hit-point ray tracers, given the vast array of possibilities for ray-tracing implementations in terms of physical accuracy. However, our visualization scheme generally works on all ray tracers, as it only depends on memory access patterns and not on specific implementations.

It is observed that the size of triangles affects the hit rate, with coarser sampled triangles having a better cache rate than smaller ones in the projection. Additionally, the implicit occlusion culling of BVHs effectively prevents most memory accesses on the back of the mesh geometry. However, despite their back-facing orientation, some parts of the mesh still undergo triangle intersection tests. This can occur because the decision on which

BVH subtree to process first is based on the distance from the intersection point to the camera, which may be inefficient for certain mesh geometries. Overall, our L1 and L2 visualizations appear to align with the raw hit rates from the NVIDIA profiler.

Vertex Permutation The tracer employs indexed triangle lists and modifies caching effects by altering the permutation of vertices in memory in two defined ways. The first method involves a random shuffling operation, which is expected to have weak cache performance. The second method involves spatially sorting the vertices, which is expected to result in more coherent accesses. Our visualizations confirm these expectations, as well as those of the NVIDIA profiler. Additionally, our visualization highlights the strength of our profiler compared to standard profilers by allowing exploration of regions of interest and profiling them. Our visualizations using the sorted memory layout show a diagonal pattern with lower caching rates, while the randomly shuffled layout does not.

BVH Heuristic Different types of BVHs may have varying effects on caches. We implemented a binary BVH using the SAH and the median-split strategy. Our visualizations indicate that for the Dragon mesh, the cache performance of faces is roughly equivalent for the median-split-based implementation. This suggests that in this case, the memory coherence of face accesses is not impacted by different BVH construction heuristics. However, when visualizing more complex scenes such as San Miguel, hit rates decrease when tracing a median-split representation. This is due to an increased number of unnecessary visits to BVH nodes caused by a higher EPO.

Framebuffer Resolution The visualization demonstrates that increasing the size of the framebuffer leads to higher cache efficiency on the geometry due to the higher memory coherence resulting from a denser sampling of the mesh.

Work Distribution Scheduling typically affects cache performance. To demonstrate this in our visualization, we implemented two different scheduling techniques. 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, as confirmed by our visualization and the NVIDIA profiler. Both also demonstrate a slight decrease in the L2 hit rate. This can be attributed to the decrease in global memory coherence when scanline scheduling is tied to the SM.

Time Series We evaluated changes in the L1 visualization over time. The hit rate of a given triangle can drastically change during execution, which was one of the initial motivations for introducing a time-based exploration. The accumulation blurs out such effects, making possible misses invisible to the user. Additionally, it is evident that accessed triangles have a higher hit-rate probability in subsequent frames due to the increasing number of memory accesses. Initial frames only show bounding volume accesses. In the following frames, the visualizations demonstrate that the size of a bounding volume, which corresponds to its level, decreases during traversal.

Framebuffer Visualization Visualizations of framebuffer writes reveal an intriguing outcome. The visualization of the pixel write order highlights the shape through color variations. The silhouette of the object that emerges can be attributed to the previously mentioned 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 to optimize the surface area of BVH nodes. This phenomenon serves as a prime example of how performance data can be projected between domains, as described in Section 2.4.2.

3.2.4. Individual Contributions

As corresponding and leading author, I led the overall research question and design, related work, management, and introduction, as well as discussion of the paper. The methodology contains two major parts: The profiler, which is the backend that generates and simulates profiling values to be visualized, and the dashboard, which visualizes this data in a graphical user interface. I developed the overall concept of the publication to visualize fine-grained performance profiling values onto meshes and how to extract and simulate them from GPU programs in detail, while K. Riemann developed the concept of the dashboard in detail. During the writing process, K. Riemann improved major parts of the writing style in the paper as part of an internal review process and S. Guthe and D. W. Fellner supervised the work regarding the submission process.

3.3. Reconstructing Bounding Volume Hierarchies from Memory Traces of Ray Tracers

This section summarizes the paper

> M. von Buelow, T. Stensbeck, V. Knauthe, S. Guthe, and D. W. Fellner
> "Reconstructing Bounding Volume Hierarchies from Memory Traces of Ray Tracers"

which was published in *Pacific Graphics Short Papers, Posters, and Work-in-Progress Papers 2022*.

3.3.1. Motivation and Contributions

State-of-the-art GPU ray-tracing applications often use proprietary hardware acceleration, with implementation details frequently hidden from the scientific community. Efficient ray tracing relies heavily on BVHs that are constructed to optimize performance [AKL13]. An examination of the runtime behavior of ray tracers can provide insight into the heuristics employed in the construction of BVHs.

The objective of this work is to retrieve additional information on the construction of BVHs and reconstruct their underlying data structure. To achieve this goal, we analyze the memory accesses of a ray tracer. We extract the list of memory accesses from the ray-tracing application using a dynamic binary instrumentation tool for GPUs and export them to a memory trace. We define multiple analysis steps that are largely independent from the actual BVH layout. These steps successively reconstruct the underlying BVH structure, which can be further refined by accumulating reconstruction results from individual ray-tracing viewports. Our reconstruction steps first rely on a graph data structure that has attached weights on its edges. This structure is then refined into a polytree data structure corresponding to a BVH (Section 2.1.2).

In summary our contributions are:

- A process for reconstructing the BVH from a memory trace.
- The observation that relaxing the constraints on the connections between nodes from a binary tree to a general directed graph enables a simpler and more robust reconstruction.

3.3.2. Methods and Findings

NVBIT is used to inject an instrumentation function before each memory load instruction. This function captures the operands representing the loaded addresses and passes them to the host. The outcome of this procedure is a list of memory accesses, which is known as a *memory trace*. This list can be used for subsequent analysis.

A set of four analysis steps ia applied to a graph data structure representing the potential links between BVH nodes. These analyses annotate the edges of the graph with weights representing the probability of being an actual edge of the BVH. The analyses can be run independently, allowing for a modular application design. Their main purpose is to provide metrics about potential dependencies in memory, which are then used to build the desired graph. The first analysis step is the *linear-access analysis*, which checks for each accessed structure of an allocation whether it was accessed in a linear fashion. An example for this analysis are leaf nodes as they typically store an array of triangles for contiguous access. Moreover, the *inter-allocation link analysis* aims to capture links between two different allocations. Some allocations simply store indices (implicit or explicit) to another allocation, such as indexed face lists, making them interdependent. We capture links by storing the set of subsequently accessed indices in other allocations. The *consecutive-access analysis* identifies dependencies within a single allocation by recording successively accessed indices. This approach enables the capture of hierarchical dependencies for BVH trees that are stored in the same allocation. However, false links may occur due to irregular traversal of child nodes resulting from stack-based traversal. Thus, the *stack analysis* is specialized for such stack-based BVH traversals. Its aim is to refine the quality of the reconstructed BVH by keeping track of push and pop operations on the ray-tracing stack. Since stack operations are usually the only source of local memory usage, our reconstruction algorithm tracks local memory load and save operations.

To improve the accuracy of the reconstruction result, our BVH reconstruction pipeline can merge multiple independent partial reconstruction results. Finally, the graph data structure is refined into a polytree by defining a root and recursively selecting the two children with the highest weights.

3.3.3. Discussion

For BVH reconstructions based on a single ray-tracing viewport, the reconstruction rate is consistently below $50\,\%$. This is due to the rendered meshes, which display fully connected objects with the backside hidden from the camera. BVHs are designed to prevent access to invisible triangles, which are typically out of view, on the backside, or occluded. Our reconstruction approach relies on memory accesses, which results in certain mesh components remaining invisible. The *Bunny* model has a BVH reconstruction rate of $44\,\%$ to $49\,\%$, while *Dragon* and *Happy Budda* have slightly lower rates. The *Living Room* scene has additional mesh components that are outside the view frustum, resulting in a lower reconstruction rate. Altering the resolution of the rendered viewport affects the reconstruction rate in an S-curve until it reaches saturation. This phenomenon can be explained by triangles that are not visible to any rays due to undersampling of the mesh [Sha98].

Our reconstruction approach is able to achieve $100\,\%$ accuracy with a few uniformly chosen cameras covering the entire Bunny model. However, more complex models, such as the Dragon, only reach $85.5\,\%$ accuracy with simple camera positions, and this increases to $98\,\%$ when carefully selecting cameras that cover all parts of the mesh. This demonstrates that reconstructing the entire BVH may be impossible in certain cases, as the mesh may be modeled in a way that individual triangles cannot be seen by any natural camera configuration. However, we contend that BVHs with reconstruction rates achieved by our approach provide a solid foundation for further analysis of the underlying BVH heuristics.

3.3.4. Individual Contributions

As corresponding and leading author, M. von Buelow led the overall research design, management and writing process of the paper. All authors contributed the literature review together where M. von Buelow took most of the work. The research design and choice of the theoretical model was done by T. Stensbeck and M. von Buelow together. T. Stensbeck planned and derived the concept together with M. von Buelow, while T. Stensbeck implemented the prototype. The results and discussion were primarily written by T. Stensbeck. The central implications of this work were mainly derived by T. Stensbeck. Volker Knauthe wrote parts of the introduction and abstract. S. Guthe and D. W. Fellner were general advisors of this work and contributed with continuous feedback during all phases of the paper writing process.dissertation.

3.4. In-Situ Profiling Feedback for GPGPU Code

This section summarizes the poster

> A. Pauli, M. von Buelow, and D. Ströter
> *In-Situ Profiling Feedback for GPGPU Code*

which is currently unpublished.

3.4.1. Motivation and Contributions

Harward, Irwin, and Churcher [HIC10] implemented the idea of visualizing profiling metrics in source code using simple techniques that display the metrics directly in the code. They also added a secondary visualization layer at the beginning of each line to display an additional metric. Beck, Moseler, Diehl, and Rey [Bec+13] emphasized the psychological perspective of providing in-situ profiling metrics. The study evaluates how profiling data is cognitively processed and concludes that in-situ visualizations physically integrate different representations. This integration mitigates the split-attention effect and reduces extraneous cognitive load, resulting in enhanced cognitive resources for information processing and improved user information processing. The toolset of Cito, Leitner, Bosshard, Knecht, Mazlami, and Gall [Cit+18] offers advanced techniques for interactive visualization in an integrated development environment. While existing works provide a solid theoretical and practical foundation, they are tied to specific programming languages like Java or to CPU hardware. This poster presents an approach capable of inspecting arbitrary source code that can be executed on an NVIDIA GPU and can be easily migrated to other GPU vendors.

In summary, our contribution is an in-situ code profiling implementation focused on GPU hardware that maps memory profiling metrics to the domain of the source code by estimating per-line metrics of the program.

3.4.2. Methods and Findings

The in-situ profiler consists of two main components: the *frontend* and the *backend*. The backend relies on a framework [BGF22] that emulates certain parts of the memory of an NVIDIA GPU based on a list of memory requests obtained through *dynamic binary instrumentation*. The memory simulation also includes the simulation of caching behavior, allowing for the accumulation of per-line cache hit rates. This is in contrast to standard general-purpose profilers, which can only account for such metrics at the granularity of a kernel call.

Annotated cache hit rates are utilized in the *integrated development environment* (IDE) that defines the frontend of our profiler. Specifically, we chose *Visual Studio Code* as the IDE due to its popularity in the community and the ability to customize parts of the code editor, which is necessary for our visualization.

The user's workflow is defined by developing a program until a certain point is reached, at which point the program should be analyzed using our profiler. The user compiles the GPGPU program in debug mode to allow our profiler to extract source code line information. Then, the user can right-click on the signature of a kernel in the source code editor to start the profiling procedure for that specific kernel. After the profiling process is complete, the resulting cache hit rates are highlighted in the source code using boxes around the line, with a color coding representing the hit rate value. Hovering over the box provides more detailed information.

This representation allows the user to identify inefficient lines in the source code and optimize them. The profiling workflow can be repeated an arbitrary number of times.

3.4.3. Discussion

Our in-situ profiler was evaluated on a program that included lines expected to be inefficient in terms of cache hit rates. To achieve this, we chose a problem size large enough to produce a reasonable number of incoherent memory accesses. The profiling system confirms this behavior by marking those lines in red, visually corresponding to expected values. Our work represents an initial approach to bringing the idea of in-situ profiling into the domain of GPUs. However, in the future, a larger evaluation and further study of the interest group must be conducted.

3.4.4. Individual Contributions

The poster is based on the bachelor's thesis titled "In Situ Code Profiling IDE Feedback for CUDA Applications" by A. Pauli. I was the primary supervisor for the thesis. Although A. Pauli had the initial idea for the project, the final research question was developed iteratively with D. Ströter and me. Furthermore, I contributed to the poster's content, while D. Ströter reviewed it for an upcoming conference proceeding submission.

3.5. A Visual Profiling System for Direct Volume Rendering

This section summarizes the paper

M. von Buelow, D. Ströter, A. Rak, and D. W. Fellner
"A Visual Profiling System for Direct Volume Rendering"

which was published in *Eurographics 2024 - Short Papers 2024*.

3.5.1. Motivation and Contributions

Direct volume rendering (DVR) is used to explore simulation results of virtual prototyping, typically implemented using ray marching. Over the past few decades, many GPU-based DVR implementations have been developed to optimize various aspects for improved run-time performance. However, only a few works have focused on profiling tools for individual DVR implementations. In this paper, we present a specialized profiling tool for DVR implementations. The tool visualizes extracted profiling values in the domain of the finally rendered image, making it possible to obtain a spatial component of profiled metrics. The basis of the tool is a memory simulator that uses dynamic binary instrumentation, as described in Section 2.3.3.

In summary, our contributions are:

- An extensible visual profiling system that visualizes cache hit rates and code branching in the spatial domain of the framebuffer from arbitrary compiled GPU DVR applications.
- An analysis of several configurations of a state-of-the-art DVR implementation using our proposed profiler.

3.5.2. Methods and Findings

Our pipeline is based on a toolkit [BGF22] that includes a simulator of a typical memory pipeline of an NVIDIA GPU. The simulator is necessary because cache hit rates are only available on a per-kernel basis using standard profilers.

Metrics To visualize the cache hit rate, we use the list of memory accesses extracted from the memory simulation pipeline, which has already undergone coalescing simulation and cache hit rate estimation. We then accumulate the cache hit rates based on the grid coordinates to transform the flat list of annotations into the spatial domain of the output image. To provide the user with a more comprehensive understanding of the program's behavior, we perform this aggregation for each memory allocation separately. Our toolkit automatically identifies and separates individual memory allocations of the profiled program and assigns each set of memory accesses $a_{(x,y,i)}$ a continuous allocation identifier i. This allows us to use a simple array of two-dimensional hit rate counters to handle the desired accumulation. Note, that we use the operator $\lceil \cdot \rceil$ to denote the number of hits in a set of memory transactions. The hit rate is calculated for each pixel individually as follows:

$$p_{(x,y,i)}(hit) = \frac{\lceil a_{(x,y,i)} \rceil}{|a_{(x,y,i)}|} \tag{3.3}$$

The methodology employed for accounting for branching in the image domain is analogous to that employed for cache hit rate accounting. On GPUs, branching is executed by sequentially executing each code path with each thread in a warp, consisting of $w = 32$ threads, being inactive in the current code path. To measure branching, a counter corresponding to the specific pixel is incremented. The number of active threads within a warp i is denoted as n_i, and the set of warp-wise memory requests per pixel is denoted as $r_{(x,y)}$. Moreover, the metric for the number of memory requests per pixel $N_{(x,y)}$ and the branching rate $b_{(x,y)} \in [0, 1]$ are given by:

$$N_{(x,y)} = \sum_{i \in r_{(x,y)}} n_i \tag{3.4}$$

$$b_{(x,y)} = \frac{N_{(x,y)}}{w \cdot |r_{(x,y)}|} \tag{3.5}$$

These extracted values are transformed into color values using standard color mapping.

Visualization System Our visualization system aims to optimize the run-time performance of ray-tracing applications for experienced software engineers in the computer graphics and ray-tracing domain. Standard profilers only provide coarse metrics at the granularity of a kernel call. However, our visual profiler provides further insight into performance data mapped to the spatial domain of the framebuffer, which can potentially help identify inefficiencies in spatial data structures.

3.5.3. Discussion

In this section, we present our profiling tools utilizing the *SimJeb122* and *SimJEB514* models. We use compression rates that are controlled by the DVR-internal parameters $\alpha \in \{0, 1, 2\}$ and sampling rates of $S \in \{2, 1, 0.5\}$.

Our visualizations clearly depict the original structure of the model, despite only visualizing performance values, which is a common phenomenon in performance visualization [Sch+11]. Additionally, we observed a general trend of good caching efficiency in vertices, which can be attributed to coherent memory accesses resulting from the relatively high amount of data loaded per vertex.

The visualizations of the SimJeb122 model indicate that branching primarily occurs within the silhouette boundary of the model. This is expected as threads diverge in these regions, resulting in either empty space or the model. The individual compression and sampling rates do not significantly affect branching visualizations. However, as α values increase and sampling rates decrease, the number of global accesses also increases. The internal DVR implementation's children offsets have slightly higher cache hit rates when the sampling rate is increased due to oversampling behavior, where individual rays intersect the same leaf nodes more frequently. The primitive index allocation used in the DVR implementation shows varying cache hit rates depending on the scalar data and model structure. At the positions of holes in the rendered model, we observe lower scalar values, resulting in a higher local sampling point density and better cache hit rates.

The visualizations of the SimJEB514 model have similar effects, except for a vertical line that becomes visible on the right side of the framebuffer. This line correlates with a higher local mesh resolution in the corresponding region, leading to a higher caching rate of the primitive indices in these regions. Increasing the compression rate α and the sampling rate positively influences the cache on thin model regions.

3.5.4. Individual Contributions

The project combines profiling, based on M. von Buelow's work, and rendering, based on D. Ströter's work. To achieve this, M. von Buelow's toolkit was extended, and D. Ströter's DVR was used for evaluation. M. von Buelow, as the corresponding and lead author, led the research question, paper design, and management. M. von Buelow wrote Section 3, parts of the introduction, related work, abstract, and conclusion. The evaluation was conducted jointly by M. von Buelow and D. Ströter. D. Ströter wrote the results section and its equivalent in the introduction, related work, abstract, and conclusion. D. Ströter implemented the DVR application for profiling. D. Ströter added important functionalities to the DVR application to distinguish between allocations. D. W. Fellner supervised the paper with respect to the submission process.

3.6. GPU Ray Tracing of Triangular Grid Primitives

This section summarizes the paper

> M. von Buelow
> *GPU Ray Tracing of Triangular Grid Primitives*

which is currently unpublished. However, parts of the paper are published in *International Conference on Artificial Reality and Telexistence and Eurographics Symposium on Virtual Environments Posters and Demos 2023* as:

> M. von Buelow, A. Kuijper, and D. W. Fellner
> "A GPU Ray Tracing Implementation for Triangular Grid Primitives"

3.6.1. Motivation and Contributions

Ray tracing is a rendering technique that transforms scene descriptions, such as triangle meshes or neural radiance fields, into images. It has practical applications in computer-aided design, virtual reality, games, and the movie industry. Ray tracing has become increasingly popular in the real-time domain, replacing classical rasterization due to its superior capabilities in physically correct global illumination approximation [PJH17]. In addition, it has been embedded in specialized GPU hardware in recent years [Par+10]. Although high quality scenes can currently be rendered in real time, memory reduction advancements can be utilized to increase the density of a mesh and compensate for more detail in the output image.

Structured primitives, like triangular grid primitives, can be used in areas where static geometry can be assumed within a coarse domain, such as subdivision surface representations or discretized displacement maps. The data can be derived from either measurements or previous simplification steps, such as in micro mesh rendering. Our focus is on rendering these structures using ray tracing, so we will use traditional subdivision surfaces or displacement maps as input data. However, conversion from micro meshes will be straightforward. Although there are recent works that implement these primitives [BP23; MMT23; KSW21], none of them focus on a pure software implementation that uses the GPU as a ray-tracing device. The downside of these implementations is that they require newer hardware, making the idea less accessible to older or less specialized devices such as smartphones or self-contained virtual environment devices.

This paper presents a data structure optimized for ray tracing triangular grid primitives on the GPU. The main idea is to reduce the memory footprint of connectivity data by exploiting the internal structure of the grid. This is accomplished by differentiating between interior and boundary vertices of a grid primitive and referencing them using a two-level static indexed triangle list, which ensures that no vertex is stored more than once. This basic structure is accelerated by a secondary level BVH using bounding spheres built around the geometry after subdivision recursion. This BVH is stored semi-implicitly in memory, so that only few additional memory is required and its construction is computationally negligible. Our data structure is very easy to implement because it exploits the structure of the subdivision recursion tree.

In summary, our contributions are:

- A joint geometry and BVH data structure for triangular grid primitives targeting software GPU ray tracers that achieves a memory footprint of $6.3\,\mathrm{B}$ per triangle for four-level subdivisions.
- Increased availability due to a simple design that is easy to implement on arbitrary GPU architectures.

3.6.2. Methods and Findings

The primary concept of our lower-level acceleration structure is to utilize semi-implicitly derived spheres from triangles as enclosing volumes and align the BVH with the recursive subdivision aspect. These spheres comprise their centers, which we calculate strictly as the centroid of the triangle, and the radius is chosen such that all triangles from recursively traversed children tightly fit within the sphere. All radii are stored in memory using a breadth-first search (BFS) traversal order. This ensures that equal levels of detail are located within the same region of memory. Additionally, since all subtrees have the same subdivision depth, the BVH is a complete tree, making implicit indexing trivial.

The geometry is based on an indexed triangle list with modifications to improve run-time performance on GPUs. Each grid has the same connectivity, allowing for the storage of vertex indices for each triangle in the GPU's constant memory. This results in fast access times, almost as fast as registers. To prevent encoding vertices of grid boundaries and corners multiple times, we have extended the simple scheme by introducing a secondary indexing layer. This layer translates from *preliminary indices* to *final indices*. Preliminary indices are referenced in the constant memory indexed triangle list and translated to final indices in the corresponding vertex buffer at runtime. The corner vertices of the grid have the first three preliminary indices, followed by all three boundaries. Boundaries and corners require additional base pointers, which are explicitly stored in the grid data structure. To reference two adjacent half-edges only once, negative pointers are allowed, enabling the adjacent triangle to index vertices in opposite directions with the same preliminary indices. The number of inner vertices is always identical between grids, and thus can be fully implicitly addressed since they are never shared with another grid.

The structure is traversed using the *while-while* approach, which is implemented differently for the two-level BVH. The first inner while loop traverses the top-level hierarchy, while the second traverses the bottom-level hierarchy and its leaves, instead of distinguishing between inner and leaf nodes. Traversal of the bottom-level hierarchy starts by loading the tuple of the grid, which includes the corner and boundary pointers, from memory. The traversal implementation computes the bounding spheres of the six preliminary vertex indices for the first subdivision level, which consist of four triangles, fetched from constant memory and translates them into the final index. This process determines whether to continue traversing a child. The algorithm works by creating intersection intervals for each intersection, which are then used to push sorted intersected spheres onto the traversal stack based on the side of the interval that represents the closer hit point. Traversal continues until the maximum traversal depth is reached, at which point standard triangle intersection tests are performed instead of sphere intersections. To keep track of the current node, the traversal state requires only one variable in addition to the stack and the current interval. The calculation of child node offsets can be done implicitly due to the complete BFS layout.

Our proposed data structure enables the treatment of all inner nodes as leaf nodes by performing triangle intersection tests directly on the intermediate subdivision. This allows for a basic level of detail functionality that can be easily implemented by reducing the maximum hierarchy depth during traversal.

3.6.3. Discussion

Memory Footprint The memory footprint has been divided into four groups: *connectivity*, which consists of vertex indices and our grid data structure; *geometry*, which consists of vertices; *BVH*, which consists of the tree data structure; and *bounding boxes* and *bounding spheres*. Our data structure significantly reduces the storage overhead of the connectivity. This fact supports the use of grid data structures to eliminate unnecessary connectivity information by assuming static connectivity within a grid. Our implementation also demonstrates that boundary vertices occupy approximately one-third of the space of the entire geometry, emphasizing the importance of not redundantly storing them. However, the memory usage of the geometry remains unchanged, confirming that our implementation has not duplicated any vertices at grid boundaries. The reduced memory consumption for the bounding volume hierarchy (BVH), including its bounding volumes, results from the implicit representation of the bottom-level BVH and its significantly reduced geometric information.

When analyzing the trend of memory usage as BVH levels increase, it becomes clear that higher BVH levels require more memory for storage. Our representation is generally more memory-efficient, except for the case of one-level subdivision with full BVH depth. This is due to the large memory footprint of our primitive compared to a standard indexed triangle list or the four corresponding triangles resulting from the subdivision.

Run-Time Performance The triangular grid data structure and reference implementation are both implemented using the *CUDA* runtime API. When examining the run-time performance of each level of subdivision, we observe a speedup of $0\,\%$ to $28\,\%$ for an one-level subdivision. For a two-level subdivision, we see speedups ranging from $10\,\%$ to $39\,\%$ impact. The trend continues for three levels with a $16\,\%$ to $68\,\%$ performance impact and for four levels with a $55\,\%$ to $85\,\%$ performance impact. The reason for this behavior is that our semi-implicitly constructed bottom-level BVH has increased the EPO [AKL13], resulting in more BVH nodes being traversed unnecessarily during rendering. Our results indicate that meshes with a higher number of visible triangles in the viewport tend to produce more computational overhead, which was expected since more BVH nodes need to be intersected. It can be observed that the *Head* and *Armadillo* meshes exhibit the highest potential among the bottom-level BVH. This can be attributed to their lower mesh complexity, resulting in a lower number of self-intersecting bounding volumes.

3.6.4. Individual Contributions (Poster Paper)

A. Kuijper and D. W. Fellner supervised the work on the submission process. All other work (research question and design, related work, management, introduction and discussion, etc.) was done by M. von Buelow.

3.6.5. Individual Contributions (Unshortened Version)

This work was completed solely by the author and does not have any co-authors.

3.7. Compression of Non-Manifold Polygonal Meshes Revisited

This section summarizes the paper

> M. von Buelow, S. Guthe, and M. Goesele
> "Compression of Non-Manifold Polygonal Meshes Revisited"

which was published in *Vision, Modeling & Visualization 2017*.

3.7.1. Motivation and Contributions

Triangle and polygonal meshes are widely used and have numerous applications. Modern meshes often consist of millions or even billions of polygons, requiring efficient storage methods. To address this, various compression techniques and standards have been proposed. Initially, meshes were stored in plain ASCII or binary format and compressed using generic methods such as *GZip*. While this approach reduces storage requirements, it does not take advantage of redundancies in the underlying mesh, resulting in sub-optimal compression rates.

Compression approaches can be classified as either progressive or single rate. Progressive approaches allow for different levels of detail and instant visualization of coarser levels while loading the remainder of the data. However, multi-rate approaches tend to offer lower compression rates than single-rate approaches. Several single-rate standards have been proposed over time, such as *OpenCTM*, *WebGL Loader*, *Open3DGC*, and *Google Draco*. However, these standards impose severe restrictions on the type of meshes (i.e., only triangle meshes) or the types of attributes they support.

This paper proposes a new mesh compression scheme based on the *cut-border machine* [GS98; Gum99] that enables the compression of generic, manifold, and non-manifold polygonal meshes.

More specifically, our contributions are:

- A general and efficient single-rate connectivity compression scheme for polygon meshes of arbitrary topology.
- A compression scheme for different types and number of attributes attached to vertices, faces, or corners.
- Compression rates that are on par or better than the state-of-the-art in current approaches for triangle or polygonal mesh compression even for meshes supported by other approaches.

3.7.2. Methods and Findings

Our approach involves splitting the input mesh into connectivity and attribute data and encoding both parts separately. The connectivity information is used to set up a data structure for querying adjacency information, which is then used to encode the connectivity using our extended cut-border machine. The compression of attributes, on the other hand, require the connectivity data to define an ordering and prediction.

While the original cut-border mashine [GS98] is capable of compressing a set a fully connected mesh components, it is not possible to encode ill-formed meshes that contain non-manifold parts. The impoved one can handle a subset of non-manifold cases, however, it still not allows for encoding arbitrary topologies. We thus propose a generalized compression scheme that works for arbitrary polygonal meshes. In the event of a failure of the cut-border machine, the vertex is identified as already encoded, yet it cannot be located on any cut-border. On two-manifold meshes this never happen, as the cut-border mashine processes each edge exactly twice: for creation and for deletion given the cut-border operations. We solve this problem by identifing this case programmatically and handle it as a new vertex operation splitting the mesh up into sub-compontents implicitly. Instead of encoding attribute data for the gate-adjacent vertex redundantly, we encode the global vertex index recovering the connectivity information similar to a indexed triangle list (Section 2.2.2).

In the improved cut-border machine [Gum99], the author notes that the compression can be improved by traversing the mesh using a depth-first search. This keeps the start vertex of the gate fixed as long as possible and a traversal around the triangle fan of this vertex happens as long as new vertex operations occur. When the last triangle of the triangle fan is about to be encoded, a connect forward operation closes the triangle fan and finally changes the start vertex of the gate to traverse a new triangle fan. Since polygons can always be split up into such an open triangle fan, we can exploit the traversal order to continuously encode a triangulation of the polygon. To recover the original polygon, we simply have to encode the number of additional vertices that belong to this polygon.

Whenever a vector of floating point values representing a certain attribute is encountered for the first time, it must be encoded in the compressed stream. If some loss of accuracy is acceptable, the floating point numbers can be transformed into fixed point numbers using quantization. To minimize the number of bits required for encoding attributes, only differences to a predicted attribute [Mag+15], that are required to be similar to each other, are encoded. We model the average of all available parallelograms [TG98] spanned by the triangles on the triangle fan of the adjacent vertex to predict a vertex. Faces and corners use their neighbors for prediction. To avoid loss of accuracy when computing differences in floating point arithmetic, we transform to a signed integer representation that maintains the relative ordering of numbers. Floating point numbers are stored (from msb to lsb) with a sign bit, followed by the exponent and mantissa. This creates the correct ordering for positive floating point numbers [LI06]. For negative floating-point numbers, the correct ordering can be obtained by simply inverting the 31 least significant bits.

3.7.3. Discussion

To evaluate our compression toolkit, we used a set of meshes that includes non-manifold and two-manifold meshes, as well as triangle and arbitrary polygonal meshes. The meshes we used were *Lucy*, *Bunny*, *Power Plant*, *Beethoven*, *Galleon*, and *Kobe*.

When comparing the compression ratio of our lossless compression scheme to that of Google Draco, OpenCTM, and GZip compression of the original file, it is evident that our approach consistently outperforms GZip compression. Our approach outperforms all other compression schemes for both the Lucy and Bunny meshes. However, OpenCTM performs best on the very regular Power Plant model, as well as on the Beethoven, Galleon, and Triceratops meshes. The likely reason for this difference is the use of dictionary-based compression in comparison to our approach and Google Draco's use of entropy coding. Our compression speed is faster than GZip, except for very small meshes, and on average requires about the same amount of time as OpenCTM. Furthermore, our approach is the only one, besides GZip, that can encode polygonal meshes without converting them to triangle meshes. Our approach outperforms all other approaches in terms of compression ratio for the triangular meshes, except for the Power Plant. For the polygonal meshes, our approach produces better compression rates than the only other applicable approach, GZip.

When comparing lossy compression approaches, Google Draco always produces the smallest output, with our approach in second place for the Lucy and Bunny models. The difference in compression performance is due to the attribute prediction scheme. Our approach stores attributes within the connectivity compression symbol stream, while Google Draco stores them after encoding connectivity. This allows multiple existing triangles to predict the position of encoded vertices, resulting in fewer bits needed to store differences from predicted values. To implement the same prediction stream, it is necessary to modify the compression scheme to encode all attributes after the entire connectivity has been encoded. However, our approach is the only one that can encode polygonal meshes without converting them to triangle meshes, making it the best choice in terms of compression ratio for these meshes.

In the proposed compression scheme, a triangular mesh requires 1 bit to 3 bit per vertex, while polygonal meshes require an increase of about 2 bit per vertex, when analyzing the allocation of bits within the bitstream for both connectivity and attributes. The attribute compression scheme achieves between 10 bit to 24 bit per vertex or corner during lossless compression of three 32 bit floating-point values. When quantizing attributes to 14 bit for position and 10 bit for other attributes, the compression scheme achieves between 3 bit to 11 bit per vertex or corner. This corresponds to a compression ratio of 3:1 to 10:1 on top of the quantization if it is enabled.

3.7.4. Individual Contributions

As corresponding and leading author, I led the overall research question and design, related work, management, and introduction and discussion of the paper. The basis of the work is derived from M. von Buelow's bachelor thesis "Connectivity and Attribute Compression of Triangle Meshes" from March 2017. My additional contributions are the extension of the lossless compression scheme to polygonal meshes, techniques for encoding additional types of attributes (face and corner attributes), and improved predictions for vertex attributes. S. Guthe contributed the initial design of the algorithm for the connectivity compression and attribute prediction as well as writing or editing of large parts of the paper. M. Goesele supervised the work regarding the submission process.

4. Discussion

4.1. Summary of Contributions

Ray tracing is an important rendering approach that typically projects 3D representations of a scene onto a 2D display. Improvements in ray tracing can be made through more efficient memory layouts, making advanced profiling techniques necessary to make implementation more accessible. This thesis presented a new method for precisely simulating the GPU memory pipeline and exporting detailed memory profiles for performance visualization in specific domains. Additionally, it provides traditional textual memory profiling metrics that can be displayed in finer granularity than usual. It also presented a compressed ray tracing implementation that optimizes memory traffic by making assumptions about the topological properties of the mesh being rendered. The findings can be used to evaluate and optimize a wide range of ray-tracing applications in a user-friendly manner.

In detail, a memory model of recent GPU architectures that is more accurate than state-of-the-art memory simulation pipelines was developed. This model can be used to display profiling data such as cache hit rates and memory access coalescing behavior. The memory simulator can simulate the behavior of arbitrary compiled GPU applications by extracting the list of accessed memory using dynamic binary instrumentation techniques. It then simulates parts of the memory pipeline and projects profiling data onto the original scene using color mapping, which corresponds to actual measured memory behavior. The visualization system can spatially and temporally subdivide performance metrics, enabling fine-grained analysis across multiple domains. Moreover, the visualization of write operations to the framebuffer provides valuable insights into GPU scheduling and confirmed fundamental aspects of performance visualization. This toolset for visualization can be utilized for profiling and educational purposes, demonstrating how caches function and supporting their efficient utilization. In order to enhance the usability of the profiler, initial steps were taken towards analyzing the internally used BVH structures. This was achieved by utilizing BVH memory access patterns extracted through dynamic binary instrumentation that were accumulated on a graph data structure. This thesis also presented visual profiling techniques that are framebuffer oriented for the use case of direct volume rendering. Additionally, they incorporated the branching behavior of GPU implementations and analyzed their behavior using the profiling toolset. The concept of projecting profiling metrics into other domains was further developed by outlining a method for projecting profiling data back to its initial point of origin: the source code. Based on knowledges in GPU memory modelling, this thesis presented a ray-tracing implementation that uses a joint compressed geometry and BVH representation using triangular grid primitives achieving a storage efficiency of $6.3\,\mathrm{B}$ per triangle, which was previously only possible by proprietary hardware implementations. The approach can be implemented on any GPU architecture suffering from such technologies due to its simple design, making the idea of ray tracing on triangular grid primitives more accessible. Additionally, a single-rate mesh compression technique that can process arbitrary mesh topologies has been developed. Its generality enables the encoding of a wide range of attributes and structures, making it more efficient than other compression techniques.

4.2. Limitations

Although profiling metrics can display fine-grained information in individual domains, they are still just indicators and may not reveal the actual inefficiencies. This is because the model may be inaccurate in specialized situations or the aggregations may be too coarse for a specific scenario. General-purpose profilers also suffer from this problem, as they only display metrics per kernel. Moreover, the implementation of ray tracing on grid structures is limited in performance, which is a trade-off for more efficient storage. However, this approach allows for larger and more detailed scene geometry. Finally, the web-based visual profiling system has limited memory to represent profiling metrics on meshes. Further improvements or migration to other platforms could address this limitation.

4.3. Future Work

In the future, there are plans to implement detailed visual profiling metrics using the profiling pipeline for the triangular grid ray-tracing approach. Although the presented memory model is more accurate than state-of-the-art models, its error values on L2 cache hit rate approximations suggest that there is still room for improvement. Moreover, the ray tracing implementation has high potential for lossy compression techniques, as it introduces additional spatial dependencies that can be used to predict parts of the geometry. It is possible that the level of detail functionality could compensate for run-time performance. The evaluation of further DVR implementations is desired for the profiling tool on direct volume rendering.

Bibliography

[AHH89] A. Agarwal, J. Hennessy, and M. Horowitz. “An analytical cache model”. In: *ACM Transactions on Computer Systems* 7.2 (May 1989), pp. 184–215. doi: `10.1145/63404.63407` 28.

[AKL13] T. Aila, T. Karras, and S. Laine. “On quality metrics of bounding volume hierarchies”. In: *Proceedings of the 5th High-Performance Graphics Conference*. ACM, July 2013. doi: `10.1145/2492045.2492056` 15, 16, 44, 55.

[AL09] T. Aila and S. Laine. “Understanding the efficiency of ray traversal on GPUs”. In: *Proceedings of the Conference on High Performance Graphics 2009*. ACM, Aug. 2009. doi: `10.1145/1572769.1572792` 39, 41.

[App68] A. Appel. “Some techniques for shading machine renderings of solids”. In: *Proceedings of the April 30–May 2, 1968, spring joint computer conference on - AFIPS ’68 (Spring)*. ACM Press, 1968. doi: `10.1145/1468075.1468082` 9.

[Ara+19] Y. Arafa, G. Chennupati, A. Barai, A.-H. A. Badawy, N. Santhi, and S. Eidenbenz. “GPUs Cache Performance Estimation using Reuse Distance Analysis”. In: *2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC)*. IEEE, Oct. 2019, pp. 1–8. doi: `10.1109/ipccc47392.2019.8958760` 28.

[Ara+20] Y. Arafa, A.-H. Badawy, G. Chennupati, A. Barai, N. Santhi, and S. Eidenbenz. “Fast, accurate, and scalable memory modeling of GPGPUs using reuse profiles”. In: *Proceedings of the 34th ACM International Conference on Supercomputing*. ACM, June 2020. doi: `10.1145/3392717.3392761` 28, 36, 40.

[Bak+09] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. “Analyzing CUDA workloads using a detailed GPU simulator”. In: *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, Apr. 2009, pp. 163–174. doi: `10.1109/ispass.2009.4919648` 27.

[Bec+13] F. Beck, O. Moseler, S. Diehl, and G. D. Rey. “In situ understanding of performance bottlenecks through visually augmented code”. In: *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, May 2013. doi: 10.1109/icpc.2013.6613834 47.

[Ben+07] C. Benthin, S. Boulos, D. Lacewell, and I. Wald. *Packet-based Ray Tracing of Catmull-Clark Subdivision Surfaces*. 2007 20.

[Ben+18] C. Benthin, I. Wald, S. Woop, and A. T. Áfra. “Compressed-leaf bounding volume hierarchies”. In: *Proceedings of the Conference on High-Performance Graphics*. ACM, Aug. 2018. doi: 10.1145/3231578.3231581 21.

[Ber+08] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry. Algorithms and Applications*. Springer Berlin Heidelberg, 2008. doi: 10.1007/978-3-540-77974-2 16.

[BGF22] M. von Buelow, S. Guthe, and D. W. Fellner. “Fine-Grained Memory Profiling of GPGPU Kernels”. In: *Computer Graphics Forum* 41.7 (Oct. 2022): *Pacific Graphics*. doi: 10.1111/cgf.14671 6, 28, 34, 47, 49, 77.

[BGG17] M. von Buelow, S. Guthe, and M. Goesele. “Compression of Non-Manifold Polygonal Meshes Revisited”. In: *Vision, Modeling & Visualization*. The Eurographics Association, Sept. 2017. doi: 10.2312/vmv.20171266 6, 19, 56, 87.

[BKF23] M. von Buelow, A. Kuijper, and D. W. Fellner. “A GPU Ray Tracing Implementation for Triangular Grid Primitives”. In: *International Conference on Artificial Reality and Telexistence and Eurographics Symposium on Virtual Environments Posters and Demos*. The Eurographics Association, Dec. 2023. doi: 10.2312/egve.20231341 6, 52, 85.

[BM13] M. Brehmer and T. Munzner. “A Multi-Level Typology of Abstract Visualization Tasks”. In: *IEEE Transactions on Visualization and Computer Graphics* 19.12 (Dec. 2013), pp. 2376–2385. doi: 10.1109/tvcg.2013.124 29.

[BP23] C. Benthin and C. Peters. “Real-Time Ray Tracing of Micro-Poly Geometry with Hierarchical Level of Detail”. In: *Computer Graphics Forum* 42.8 (Aug. 2023). doi: 10.1111/cgf.14868 5, 21, 52.

[Bue+22a] M. von Buelow, K. Riemann, S. Guthe, and D. W. Fellner. “Profiling and Visualizing GPU Memory Access and Cache Behavior of Ray Tracers”. In: *Eurographics Symposium on Parallel Graphics and Visualization*. best paper award. The Eurographics Association, June 2022. doi: 10.2312/pgv.20221061 6, 22, 28, 39, 79.

[Bue+22b] M. von Buelow, T. Stensbeck, V. Knauthe, S. Guthe, and D. W. Fellner. “Reconstructing Bounding Volume Hierarchies from Memory Traces of Ray Tracers”. In: *Pacific Graphics Short Papers, Posters, and Work-in-Progress Papers*. The Eurographics Association, Oct. 2022. doi: `10.2312/pg.20221243` 6, 31, 44, 81.

[Bue+24] M. von Buelow, D. Ströter, A. Rak, and D. W. Fellner. “A Visual Profiling System for Direct Volume Rendering”. In: *Eurographics 2024 - Short Papers*. Apr. 2024. doi: `10.2312/egs.20241030` 6, 49, 83.

[Bue24] M. von Buelow. *GPU Ray Tracing of Triangular Grid Primitives*. Tech. rep. Technical University of Darmstadt, May 2024. doi: `10.26083/tuprints-00027343` 6, 52, 93.

[Bus+16] J. Bush, M. A. Khasawneh, K. Z. Mahmoud, and T. N. Miller. “NyuziRaster. Optimizing rasterizer performance and energy in the Nyuzi open source GPU”. In: *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, Apr. 2016, pp. 204–213. doi: `10.1109/ispass.2016.7482095` 27.

[Bus+21] J. Bush, N. Taherinejad, E. Willegger, M. Wojcik, M. Kessler, J. Blatnik, I. Daktylidis, J. Ferdigg, and D. Haslauer. “Nyuzi. An Open Source GPGPU for Graphics, Enhanced with OpenCL Compiler for Calculations”. In: *IEEE Design, Automation & Test in Europe*. IEEE. 2021, p. 1 27.

[BVW21] C. Benthin, K. Vaidyanathan, and S. Woop. “Ray Tracing Lossy Compressed Grid Primitives”. In: *Eurographics 2021 - Short Papers*. The Eurographics Association, 2021. doi: `10.2312/EGS.20211009` 20.

[CC78] E. Catmull and J. Clark. “Recursively generated B-spline surfaces on arbitrary topological meshes”. In: *Computer-Aided Design* 10.6 (Nov. 1978), pp. 350–355. doi: `10.1016/0010-4485(78)90110-0` 20.

[Che+09] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. “Rodinia. A benchmark suite for heterogeneous computing”. In: *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Oct. 2009, pp. 44–54. doi: `10.1109/iiswc.2009.5306797` 36.

[Che+13] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron. “Pannotia. Understanding irregular GPGPU graph applications”. In: *2013 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Sept. 2013, pp. 185–195. doi: `10.1109/iiswc.2013.6704684` 36.

[Cit+18] J. Cito, P. Leitner, C. Bosshard, M. Knecht, G. Mazlami, and H. C. Gall. “PerformanceHat. augmenting source code with runtime performance traces in the IDE”. In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, May 2018. doi: 10.1145/3183440.3183481 47.

[Coo84] R. L. Cook. “Shade trees”. In: *ACM SIGGRAPH Computer Graphics* 18.3 (Jan. 1984), pp. 223–231. doi: 10.1145/964965.808602 20.

[Cra50] G. Cramer. *Introduction à l'analyse des lignes courbes algébriques*. chez les Frères Cramer and Cl. Philibert, 1750. doi: 10.3931/E-RARA-4048 10.

[Dae09] J. Daems. “Euler's Gem. The Polyhedron Formula and the Birth of Topology by David S. Richeson”. In: *The Mathematical Intelligencer* 32.3 (Dec. 2009), pp. 56–57. doi: 10.1007/s00283-009-9116-0 16.

[DGP04] D. E. DeMarle, C. P. Gribble, and S. G. Parker. “Memory-Savvy Distributed Interactive Ray Tracing”. In: *Eurographics Workshop on Parallel Graphics and Visualization*. The Eurographics Association, 2004. doi: 10.2312/EGPGV/EGPGV04/093-100 41.

[Eis+12] M. Eisemann, P. Bauszat, S. Guthe, and M. Magnor. “Geometry Presorting for Implicit Object Space Partitioning”. In: *Computer Graphics Forum* 31.4 (June 2012), pp. 1445–1454. doi: 10.1111/j.1467-8659.2012.03140.x 21.

[Fly66] M. J. Flynn. “Very high-speed computing systems”. In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909. doi: 10.1109/proc.1966.5273 23.

[Gra+12] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. “Auto-tuning a high-level language targeted to GPU codes”. In: *2012 Innovative Parallel Computing (InPar)*. IEEE, May 2012, pp. 1–10. doi: 10.1109/inpar.2012.6339595 36.

[GS87] J. Goldsmith and J. Salmon. “Automatic Creation of Object Hierarchies for Ray Tracing”. In: *IEEE Computer Graphics and Applications* 7.5 (May 1987), pp. 14–20. doi: 10.1109/mcg.1987.276983 14.

[GS98] S. Gumhold and W. Strasser. “Real time compression of triangle mesh connectivity”. In: *Proceedings of the 25th annual conference on Computer graphics and interactive techniques - SIGGRAPH '98*. ACM Press, 1998. doi: 10.1145/280814.280836 19, 20, 56, 57.

[Gum99] S. Gumhold. “Improved cut-border machine for triangle mesh compression”. In: *Erlangen Workshop*. 1999 18, 56, 57.

[Hel67] H. von Helmholtz. *Allgemeine Encyklopädie der Physik / 9 Handbuch der physiologischen Optik*. Vol. 9. Leopold Voss, 1867. doi: 10.3931/E-RARA-21259 9.

[HIC10] M. Harward, W. Irwin, and N. Churcher. “In Situ Software Visualisation”. In: *2010 21st Australian Software Engineering Conference*. IEEE, 2010. doi: 10.1109/aswec.2010.18 47.

[HS86] W. D. Hillis and G. L. Steele. “Data parallel algorithms”. In: *Communications of the ACM* 29.12 (Dec. 1986), pp. 1170–1183. doi: 10.1145/7902.7903 3.

[Isa+14] K. E. Isaacs, A. Giménez, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, B. Hamann, and P.-T. Bremer. “State of the Art of Performance Visualization”. In: *EuroVis - STARs*. The Eurographics Association, 2014. doi: 10.2312/EUROVISSTAR.20141177 30.

[Jia+19] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza. “Dissecting the NVidia Turing T4 GPU via Microbenchmarking”. In: (2019). doi: 10.48550/ARXIV.1903.07486. Pre-published 36.

[Kar+19] A. Karki, C. P. Keshava, S. M. Shivakumar, J. Skow, G. M. Hegde, and H. Jeon. “Detailed Characterization of Deep Neural Networks on GPUs and FPGAs”. In: *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs*. ACM, Apr. 2019. doi: 10.1145/3300053.3319418 36.

[KDY10] A. Kerr, G. Diamos, and S. Yalamanchili. “Modeling GPU-CPU workloads and systems”. In: *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, Mar. 2010. doi: 10.1145/1735688.1735696 27.

[Kel+13] A. Keller, T. Karras, I. Wald, T. Aila, S. Laine, J. Bikker, C. Gribble, W.-J. Lee, and J. McCombe. “Ray tracing is the future and ever will be...” In: *ACM SIGGRAPH 2013 Courses*. ACM, July 2013. doi: 10.1145/2504435.2504444 9.

[Kha+20] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers. “Accel-Sim. An Extensible Simulation Framework for Validated GPU Modeling”. In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, May 2020, pp. 473–486. doi: 10.1109/isca45697.2020.00047 27.

[KSW21] B. Karis, R. Stubbe, and G. Wihlidal. “A Deep Dive into Nanite Virtualized Geometry”. In: *Advances in Real-Time Rendering in Games: Part I (proc. SIGGRAPH courses)*. 2021. url: `https://advances.realtimerendering.com/s2021/Karis_Nanite_SIGGRAPH_Advances_2021_final.pdf` 52.

[Lan98] W. B. Langdon. *Genetic Programming and Data Structures*. Springer US, 1998. doi: `10.1007/978-1-4615-5731-9` 19.

[LI06] P. Lindstrom and M. Isenburg. “Fast and Efficient Compression of Floating-Point Data”. In: *IEEE Transactions on Visualization and Computer Graphics* 12.5 (Sept. 2006), pp. 1245–1250. doi: `10.1109/tvcg.2006.143` 58.

[Luk+05] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. “Pin. building customized program analysis tools with dynamic instrumentation”. In: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. ACM, June 2005. doi: `10.1145/1065010.1065034` 26.

[Mag+15] A. Maglo, G. Lavoué, F. Dupont, and C. Hudelot. “3D Mesh Compression. Survey, Comparisons, and Emerging Trends”. In: *ACM Computing Surveys* 47.3 (Feb. 2015), pp. 1–41. doi: `10.1145/2693443` 57.

[MB90] J. D. MacDonald and K. S. Booth. “Heuristics for ray tracing using space subdivision”. In: *The Visual Computer* 6.3 (May 1990), pp. 153–166. doi: `10.1007/bf01911006` 14, 15.

[MC17] X. Mei and X. Chu. “Dissecting GPU Memory Hierarchy Through Microbenchmarking”. In: *IEEE Transactions on Parallel and Distributed Systems* 28.1 (Jan. 2017), pp. 72–86. doi: `10.1109/tpds.2016.2549523` 25, 27.

[Mei+21] D. Meister, S. Ogaki, C. Benthin, M. J. Doyle, M. Guthe, and J. Bittner. “A Survey on Bounding Volume Hierarchies for Ray Tracing”. In: *Computer Graphics Forum* 40.2 (May 2021), pp. 683–712. doi: `10.1111/cgf.142662` 13.

[MMT23] A. Maggiordomo, H. Moreton, and M. Tarini. “Micro-Mesh Construction”. In: *ACM Transactions on Graphics* 42.4 (July 2023), pp. 1–18. doi: `10.1145/3592440` 20, 21, 52.

[Möb27] A. F. Möbius. *Der barycentrische Calcul*. Barth, 1827. doi: `10.3931/E-RARA-14538` 10.

[MRR12] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming. Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., 2012. isbn: 9780123914439 23.

[MT97] T. Möller and B. Trumbore. “Fast, Minimum Storage Ray-Triangle Intersection”. In: *Journal of Graphics Tools* 2.1 (Jan. 1997), pp. 21–28. doi: 10.1080/10867651.1997.10487468 10.

[Nag06] H. R. Nagel. “Scientific visualization versus information visualization”. In: *Workshop on state-of-the-art in scientific and parallel computing, Sweden*. 2006, pp. 8–9 28.

[Nic+77] F. E. Nicodemus, J. C. Richmond, J. J. Hsia, I. W. Ginsberg, and T. Limperis. “Geometrical considerations and nomenclature for reflectance”. In: *Radiometry*. National Bureau of Standards, 1977. doi: 10.6028/nbs.mono.160 9.

[Nug+14] C. Nugteren, G.-J. van den Braak, H. Corporaal, and H. Bal. “A detailed GPU cache model based on reuse distance theory”. In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Feb. 2014, pp. 37–48. doi: 10.1109/hpca.2014.6835955 28.

[NVI20] NVIDIA Corporation. *Optimizing CUDA Applications for NVIDIA A100 GPU*. 2020. url: https://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/s21819-optimizing-applications-for-nvidia-ampere-gpu-architecture.pdf 23, 25.

[NVI22] NVIDIA Corporation. *CUDA Programming Guide*. 2022. url: https://docs.nvidia.com/cuda/cuda-c-programming-guide 4, 25, 35.

[NVI23] NVIDIA Corporation. *Micro-Mesh Graphics Primitive For Micro Triangles*. 2023. url: https://developer.nvidia.com/rtx/ray-tracing/micro-mesh 21.

[Par+10] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich. “OptiX. a general purpose ray tracing engine”. In: *ACM Transactions on Graphics* 29.4 (July 2010), pp. 1–13. doi: 10.1145/1778765.1778803 39, 52.

[PBS24] A. Pauli, M. von Buelow, and D. Ströter. *In-Situ Profiling Feedback for GPGPU Code*. Tech. rep. Technical University of Darmstadt, May 2024. doi: 10.26083/tuprints-00027344 6, 47, 91.

[PJH17] M. Pharr, W. Jakob, and G. Humphreys. *Physically Based Rendering*. Elsevier, 2017. doi: 10.1016/c2013-0-15557-2 3, 52.

[Red+04] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn. "PIN. a binary instrumentation tool for computer architecture research and education". In: *Proceedings of the 2004 workshop on Computer architecture education held in conjunction with the 31st International Symposium on Computer Architecture - WCAE '04*. ACM Press, 2004. doi: 10.1145/1275571.1275600 26.

[SB87] J. M. Snyder and A. H. Barr. "Ray tracing complex models containing surface tessellations". In: *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*. ACM, Aug. 1987. doi: 10.1145/37401.37417 20.

[Sch+11] M. Schulz, J. A. Levine, P.-T. Bremer, T. Gamblin, and V. Pascucci. "Interpreting Performance Data across Intuitive Domains". In: *2011 International Conference on Parallel Processing*. IEEE, Sept. 2011, pp. 206–215. doi: 10.1109/icpp.2011.60 7, 28–30, 51.

[SE10] B. Segovia and M. Ernst. "Memory Efficient Ray Tracing with Hierarchical Mesh Quantization". In: *Proceedings of Graphics Interface 2010*. Canadian Information Processing Society, 2010. isbn: 9781568817125 21.

[Sha98] C. Shannon. "Communication In The Presence Of Noise". In: *Proceedings of the IEEE* 86.2 (Feb. 1998), pp. 447–457. doi: 10.1109/jproc.1998.659497 46.

[SK04] L. A. Santaló and M. Kac. *Integral Geometry and Geometric Probability*. Cambridge University Press, Oct. 2004. doi: 10.1017/cbo9780511617331 14.

[SSH08] J. A. Stratton, S. S. Stone, and W.-m. W. Hwu. "MCUDA. An Efficient Implementation of CUDA Kernels for Multi-core CPUs". In: *Languages and Compilers for Parallel Computing*. Springer Berlin Heidelberg, 2008, pp. 16–30. doi: 10.1007/978-3-540-89740-8_2 27.

[TG98] C. Touma and C. Gotsman. "Triangle Mesh Compression". In: *Proceedings of the Graphics Interface 1998 Conference*. Canadian Human-Computer Communications Society, 1998, pp. 26–34. doi: 10.20380/GI1998.04 57.

[Tho+21] T. Thonat, F. Beaune, X. Sun, N. Carr, and T. Boubekeur. "Tessellation-free displacement mapping for ray tracing". In: *ACM Transactions on Graphics* 40.6 (Dec. 2021), pp. 1–16. doi: 10.1145/3478513.3480535 20.

[TYL11] T. Tang, X. Yang, and Y. Lin. “Cache Miss Analysis for GPU Programs Based on Stack Distance Profile”. In: *2011 31st International Conference on Distributed Computing Systems*. IEEE, June 2011, pp. 623–634. doi: 10.1109/icdcs.2011.16 28.

[Vil+19] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler. “NVBit. A Dynamic Binary Instrumentation Framework for NVIDIA GPUs”. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, Oct. 2019. doi: 10.1145/3352460.3358307 27, 35.

[VWB19] K. Vaidyanathan, S. Woop, and C. Benthin. “Wide BVH Traversal with a Short Stack”. In: *High-Performance Graphics - Short Papers*. The Eurographics Association, 2019. doi: 10.2312/HPG.20191190 21.

[Wal+01] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. “Interactive Rendering with Coherent Ray Tracing”. In: *Computer Graphics Forum* 20.3 (Sept. 2001), pp. 153–165. doi: 10.1111/1467-8659.00508 5.

[Wal+07] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley. “State of the Art in Ray Tracing Animated Scenes”. In: *Eurographics 2007 - State of the Art Reports*. The Eurographics Association, 2007. doi: 10.2312/EGST.20071056 14.

[WG17] D. Wodniok and M. Goesele. “Construction of bounding volume hierarchies with SAH cost approximation on temporary subtrees”. In: *Computers & Graphics* 62 (Feb. 2017), pp. 41–52. doi: 10.1016/j.cag.2016.12.003 16.

[Whi79] T. Whitted. “An improved illumination model for shaded display”. In: *ACM SIGGRAPH Computer Graphics* 13.2 (Aug. 1979), p. 14. doi: 10.1145/965103.807419 9.

[WMZ22] I. Wald, N. Morrical, and S. Zellmann. “A Memory Efficient Encoding for Ray Tracing Large Unstructured Data”. In: *IEEE Transactions on Visualization and Computer Graphics* 28.1 (Jan. 2022), pp. 583–592. doi: 10.1109/tvcg.2021.3114869 21, 41.

[Wod+13] D. Wodniok, A. Schulz, S. Widmer, and M. Goesele. “Analysis of Cache Behavior and Performance of Different BVH Memory Layouts for Tracing Incoherent Rays”. In: *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2013. doi: 10.2312/EGPGV/EGPGV13/057-064 41.

[WX16] D. Wang and W. Xiao. “A reuse distance based performance analysis on GPU L1 data cache”. In: *2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC)*. IEEE, Dec. 2016, pp. 1–8. doi: 10.1109/pccc.2016.7820638 28.

[Zha+13] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. “Practical Control Flow Integrity and Randomization for Binary Executables”. In: *2013 IEEE Symposium on Security and Privacy*. IEEE, May 2013, pp. 559–573. doi: 10.1109/sp.2013.44 26.

[ZS15] M. Zhang and R. Sekar. “Control Flow and Code Integrity for COTS binaries. An Effective Defense Against Real-World ROP Attacks”. In: *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, Dec. 2015. doi: 10.1145/2818000.2818016 26.

Part II.

Publications

5. Peer-Reviewed Publications

This chapter includes all peer-reviewed works that establish the foundations of this thesis.

5.1. Fine-Grained Memory Profiling of GPGPU Kernels

M. von Buelow, S. Guthe, and D. W. Fellner

Computer Graphics Forum 41.7 (2022): Pacific Graphics
DOI: 10.1111/cgf.14671

Abstract

Memory performance is a crucial bottleneck in many GPGPU applications, making optimizations for hardware and software mandatory. While hardware vendors already use highly efficient caching architectures, software engineers usually have to organize their data accordingly in order to efficiently make use of these, requiring deep knowledge of the actual hardware. In this paper we present a novel technique for fine-grained memory profiling that simulates the whole pipeline of memory flow and finally accumulates profiling values in a way that the user retains information about the potential region in the GPU program by showing these values separately for each allocation. Our memory simulator turns out to outperform state-of-the-art memory models of NVIDIA architectures by a magnitude of 2.4 for the L1 cache and 1.3 for the L2 cache, in terms of accuracy. Additionally, we find our technique of fine grained memory profiling a useful tool for memory optimizations, which we successfully show in case of ray tracing and machine learning applications.

Copyright notice

License information

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

M. von Buelow, K. Riemann, S. Guthe, and D. W. Fellner
Eurographics Symposium on Parallel Graphics and Visualization 2022
DOI: 10.2312/pgv.20221061

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.

Copyright notice

License information

Awards

This work received the *EG PGV 2022* best paper award.

5.3. Reconstructing Bounding Volume Hierarchies from Memory Traces of Ray Tracers

M. von Buelow, T. Stensbeck, V. Knauthe, S. Guthe, and D. W. Fellner

Pacific Graphics Short Papers, Posters, and Work-in-Progress Papers 2022
DOI: 10.2312/pg.20221243

Abstract

The ongoing race to improve computer graphics leads to more complex GPU hardware and ray tracing techniques whose internal functionality is sometimes hidden to the user. Bounding volume hierarchies and their construction are an important performance aspect of such ray tracing implementations. We propose a novel approach that utilizes binary instrumentation to collect memory traces and then uses them to extract the bounding volume hierarchy (BVH) by analyzing access patters. Our reconstruction allows combining memory traces captured from multiple ray tracing views independently, increasing the reconstruction result. It reaches accuracies of 30 % to 45 % when comparing against the ground-truth BVH used for ray tracing a single view on a simple scene with one object. With multiple views it is even possible to reconstruct the whole BVH, while we already achieve 98 % with just seven views. Because our approach is largely independent of the data structures used internally, these accurate reconstructions serve as a first step into estimation of unknown construction techniques of ray tracing implementations.

Copyright notice

License information

5.4. A Visual Profiling System for Direct Volume Rendering

M. von Buelow, D. Ströter, A. Rak, and D. W. Fellner

Eurographics 2024 - Short Papers 2024
DOI: 10.2312/egs.20241030

Abstract

Direct Volume Rendering (DVR) is a crucial technique that enables interactive exploration of results from scientific computing or computer graphics. Its applications range from virtual prototyping for product design to computer-aided diagnosis in medicine. Although there are many existing DVR optimizations, they do not provide a thorough analysis of memory-specific hardware behavior. This paper introduces a profiling toolkit that enables the extraction of performance metrics, such as cache hit rates and branching, from a compiled GPU-based DVR application. The metrics are visualized in the image domain to facilitate spatial visual analysis. This paper presents a pipeline that automatically extracts memory traces using binary instrumentation, simulates the GPU memory subsystem, and models DVR-specific functionality within it. The profiler is demonstrated using the Octree-Linear Bounding Volume Hierarchy (OLBVH), and the visualized profiling metrics are explained based on the OLBVH implementation. Our discussion demonstrates that optimizing ray traversal for adaptive sampling, cache usage, branching, and global memory access has the potential to improve performance.

Copyright notice

License information

5.5. A GPU Ray Tracing Implementation for Triangular Grid Primitives

M. von Buelow, A. Kuijper, and D. W. Fellner

International Conference on Artificial Reality and Telexistence and Eurographics Symposium on Virtual Environments Posters and Demos 2023
DOI: 10.2312/egve.20231341

Abstract

Triangular grid primitives are a new technique for more efficient handling of memory intensive meshes, also called micro meshes in recent proprietary hardware implementations. This makes it a technique with high potential in the area of virtual environments where hardware capabilities are typically limited. In this poster, we focus on software ray tracing on GPUs and present a novel, easy-to-implement approach that uses a two-level bounding volume hierarchy (BVH) to accelerate these grids. The primary goal of our work is to make the technology more accessible by focusing on standard GPU devices without hardware ray tracing units. With our approach, we are able to encode geometry and BVH with approximately 7.5 bytes per triangle, reducing standard representations by a factor of 3.73 while reducing BVH construction time. Our data structure achieves a peak performance impact of 16 % for a three-level subdivision.

Copyright notice

License information

5.6. Compression of Non-Manifold Polygonal Meshes Revisited

M. von Buelow, S. Guthe, and M. Goesele
Vision, Modeling & Visualization 2017
DOI: 10.2312/vmv.20171266

Abstract

Polygonal meshes are used in various fields ranging from CAD to gaming and web based applications. Reducing the size required for storing and transmitting these meshes by taking advantage of redundancies is an important aspect in all of these cases. In this paper, we present a connectivity based compression approach that predicts attributes and stores differences to the predictions together with minimal connectivity information. It is an extension to the Cut-Border Machine and applicable to arbitrary manifold and non-manifold polygonal meshes containing multiple attributes of different types. It compresses both the connectivity and attributes without loss outside of re-ordering vertices and polygons. In addition, an optional quantization step can be used to further reduce the data if a certain loss of accuracy is acceptable. Our method outperforms state-of-the-art compression techniques, including specialized triangle mesh compression approaches when applicable. Typical compression rates for our approach range from 2:1 to 6:1 for lossless compression and up to 25:1 when quantizing to 14 bit accuracy.

Copyright notice

License information

6. Unpublished Works

In addition to peer-reviewed works, there are also unpublished or works currently under review that provide additional valuable contributions to this thesis. These works are presented below.

6.1. In-Situ Profiling Feedback for GPGPU Code

A. PAULI, M. VON BUELOW, and D. STRÖTER
DOI: 10.26083/tuprints-00027344

Abstract

The evolving landscape of software development increasingly prioritizes functionality, maintainability, and developer productivity. This typically comes hand in hand with the shortcoming that less focus is invested on optimizing for runtime performance of programs. However, optimizing for performance is an important task in time-critical domains. Additionally, optimizing for performance can be an important way of reducing actual hardware requirements and achieving a better ecological footprint. So, why not bringing program optimization closer to the software engineer and reducing the disconnect between profiling results and their interpretability? This poster presents a GPU-focused in-situ profiling approach that visualizes memory profiling metrics directly inside the source code and gives the software engineer an direct hint for identifying inefficient parts during development. Performance metrics evaluated on each line are highlighted in the source code.

Copyright notice

License information

6.2. GPU Ray Tracing of Triangular Grid Primitives

M. von Buelow
DOI: 10.26083/tuprints-00027343

Abstract

Triangular grid primitives are a technique used to handle memory-intensive meshes more efficiently. They are also referred to as micro meshes in recent proprietary hardware implementations. This representation can reduce the memory footprint during ray tracing of subdivision surfaces or displacement maps that may result from mesh simplification. This paper presents a novel approach to accelerate GPU software ray tracing using a two-level bounding volume hierarchy (BVH) to store vertices in a non-redundant manner. The primary goal is to make the technology more accessible by focusing on standard GPU devices. The bottom-level BVH strictly follows the subdivision recursion, allowing for the side effect of rendering intermediate recursion depths. Our approach enables us to encode geometry and BVH using approximately 6.3 bytes per triangle, reducing standard representations by a factor of 4.5. Additionally, the construction time of the BVH is reduced. Our data structure achieves a peak performance impact of 16 % for a three-level subdivision.

Copyright notice

License information

7. Complete Publication List

In addition to the publications listed in Section 7.1 of this thesis, I have co-authored several papers in the field of visual computing. The following sections group these papers into three topics.

7.1. Core Publications

[BGF22] M. von Buelow, S. Guthe, and D. W. Fellner. “Fine-Grained Memory Profiling of GPGPU Kernels”. In: *Computer Graphics Forum* 41.7 (Oct. 2022): *Pacific Graphics*. doi: 10.1111/cgf.14671.

[BGG17] M. von Buelow, S. Guthe, and M. Goesele. “Compression of Non-Manifold Polygonal Meshes Revisited”. In: *Vision, Modeling & Visualization*. The Eurographics Association, Sept. 2017. doi: 10.2312/vmv.20171266.

[BKF23] M. von Buelow, A. Kuijper, and D. W. Fellner. “A GPU Ray Tracing Implementation for Triangular Grid Primitives”. In: *International Conference on Artificial Reality and Telexistence and Eurographics Symposium on Virtual Environments Posters and Demos*. The Eurographics Association, Dec. 2023. doi: 10.2312/egve.20231341.

[Bue+22a] M. von Buelow, K. Riemann, S. Guthe, and D. W. Fellner. “Profiling and Visualizing GPU Memory Access and Cache Behavior of Ray Tracers”. In: *Eurographics Symposium on Parallel Graphics and Visualization*. best paper award. The Eurographics Association, June 2022. doi: 10.2312/pgv.20221061.

[Bue+22b] M. von Buelow, T. Stensbeck, V. Knauthe, S. Guthe, and D. W. Fellner. “Reconstructing Bounding Volume Hierarchies from Memory Traces of Ray Tracers”. In: *Pacific Graphics Short Papers, Posters, and Work-in-Progress Papers*. The Eurographics Association, Oct. 2022. doi: 10.2312/pg.20221243.

[Bue+24] M. von Buelow, D. Ströter, A. Rak, and D. W. Fellner. “A Visual Profiling System for Direct Volume Rendering”. In: *Eurographics 2024 - Short Papers*. Apr. 2024. doi: `10.2312/egs.20241030`.

[Bue24] M. von Buelow. *GPU Ray Tracing of Triangular Grid Primitives*. Tech. rep. Technical University of Darmstadt, May 2024. doi: `10.26083/tuprints-00027343`.

[PBS24] A. Pauli, M. von Buelow, and D. Ströter. *In-Situ Profiling Feedback for GPGPU Code*. Tech. rep. Technical University of Darmstadt, May 2024. doi: `10.26083/tuprints-00027344`.

7.2. Archiving of Cultural Heritage Images

The following works are the result of a joint project with Fraunhofer IGD aimed at archiving images from cultural heritage datasets. We have developed a novel technique that uses depth-of-field segmentations to encode the foreground and background of these images independently using lossy and lossless image compression algorithms. Our proposed algorithm relies on well-documented standard image compression algorithms and is therefore suitable for long-term archiving.

[Bue+19] M. von Buelow, S. Guthe, M. Ritz, P. Santos, and D. W. Fellner. "Lossless Compression of Multi-View Cultural Heritage Image Data". In: *Eurographics Workshop on Graphics and Cultural Heritage*. The Eurographics Association, Nov. 2019. doi: `10.2312/gch.20191343`.

[Bue+20] M. von Buelow, R. Tausch, V. Knauthe, T. Wirth, S. Guthe, P. Santos, and D. W. Fellner. "Segmentation-Based Near-Lossless Compression of Multi-View Cultural Heritage Image Data". In: *Eurographics Workshop on Graphics and Cultural Heritage*. best paper award. The Eurographics Association, Nov. 2020. doi: `10.2312/gch.20201294`.

[Bue+22] M. von Buelow, R. Tausch, M. Schurig, V. Knauthe, T. Wirth, S. Guthe, P. Santos, and D. W. Fellner. "Depth of Field Segmentation for Near-Lossless Image Compression and 3D Reconstruction". In: *Journal on Computing and Cultural Heritage* 15.3 (Sept. 2022). invited paper. doi: `10.1145/3500924`.

7.3. Scanning System for Tensile Testing Specimens

The following work deals with tensile testing in materials science. We developed an automated 3D scanning system, which outperforms state-of-the-art manual measurement approaches. The system comprises the scanning setup, data acquisition, calibration, and evaluation.

[Kna+22] V. Knauthe, M. Kraus, M. von Buelow, T. Wirth, A. Rak, L. Merth, A. Erbe, C. Kontermann, S. Guthe, A. Kuijper, and D. W. Fellner. "Alignment and Reassembling of Broken Specimens for Creep Ductility Measurements". In: *Vision, Modeling & Visualization*. The Eurographics Association, Sept. 2022. doi: `10.2312/vmv.20221201`.

[Kon+23] C. Kontermann, A. Erbe, V. Knauthe, M. von Buelow, T.-U. Kern, and M. Oechsner. "Uniform Elongation Measurements on Creep Specimens by a Novel 3D-Scanning System". In: *Materials at High Temperatures* 40 (May 2023): *6th International ECCC Creep & Fracture Conference*. doi: `10.1080/09603409.2023.2261783`.

[Neu+22] K. A. Neumann, P. P. Hoffmann, M. von Buelow, V. Knauthe, T. Wirth, C. Kontermann, A. Kuijper, S. Guthe, and D. W. Fellner. "A Structure from Motion Pipeline for Orthographic Multi-View Images". In: *International Conference on Image Processing*. Institute of Electrical and Electronics Engineers (IEEE), Oct. 2022. doi: `10.1109/ICIP46576.2022.9897368`.

7.4. Miscellaneous

[Jou+21] N. Jourdan, T. Biegel, V. Knauthe, M. von Buelow, S. Guthe, and J. Metternich. “A computer vision system for saw blade condition monitoring”. In: *Procedia CIRP* 104 (Nov. 2021), pp. 1107–1112. doi: 10.1016/j.procir.2021.11.186.

[Kna+23] V. Knauthe, T. Pöllabauer, K. Faller, M. Kraus, T. Wirth, M. von Buelow, A. Kuijper, and D. W. Fellner. “Distortion-Based Transparency Detection using Deep Learning on a Novel Synthetic Image Dataset”. In: *Scandinavian Conference on Image Analysis*. Springer Nature Switzerland, Apr. 2023. doi: 10.1007/978-3-031-31435-3_17.

[Thu+16] D. Thuerck, M. Waechter, S. Widmer, M. von Buelow, P. Seemann, M. E. Pfetsch, and M. Goesele. “A Fast, Massively Parallel Solver for Large, Irregular Pairwise Markov Random Fields”. In: *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*. The Eurographics Association, June 2016. doi: 10.2312/hpg.20161203.

[Wir+22] T. Wirth, A. Jamili, M. v. Buelow, V. Knauthe, and S. Guthe. “Fitness of General-Purpose Monocular Depth Estimation Architectures for Transparent Structures”. In: *Eurographics Short Papers*. The Eurographics Association, Apr. 2022. doi: 10.2312/egs.20221020.

[Wir+24] T. Wirth, A. Rak, M. von Buelow, V. Knauthe, A. Kuijper, and D. W. Fellner. “NeRF-FF. A Plug-In Method to Mitigate Defocus-Blur for Run-Time Optimized Neural Radiance Fields”. In: *The Visual Computer* 40 (July 2024): *CGI’2024 Conference*. doi: 10.1007/s00371-024-03507-y.