High-Performance Ray Tracing on Modern Parallel Processors

Summary of the PhD Thesis

Author:
ATTILA T. ÁFRA

Supervisor:
PROF. DR. HORIA F. POP

CLUJ-NAPOCA
2013
List of Publications

ISI Journal Papers


International Conference Papers


Technical Reports


Research Grants

Grant OTKA K-104476 of the Hungarian Scientific Research Fund: *Physics Simulation and Inverse Problem Solution on Massively Parallel Systems.*
Contents

List of Publications 3

Contents 5

Contents of the Thesis 7

Summary 9

1 Introduction ................................................................. 9
2 Coherent Ray Packet Traversal Using AVX .......................... 11
3 Incoherent Ray Traversal Using AVX ................................. 13
4 Stackless Multi-BVH Traversal for CPU, MIC, and GPU ........ 14
5 Interactive Ray Tracing of Large Models ............................ 17
6 Incoherent Ray Tracing Without Acceleration Structures ........ 19
7 Conclusions ................................................................. 21

Bibliography 23
# Contents of the Thesis

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>3</td>
</tr>
<tr>
<td>List of Publications</td>
<td>5</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>7</td>
</tr>
<tr>
<td>Contents</td>
<td>9</td>
</tr>
<tr>
<td>List of Figures</td>
<td>13</td>
</tr>
<tr>
<td>List of Tables</td>
<td>15</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td></td>
</tr>
<tr>
<td>1.1 Ray Tracing</td>
<td>18</td>
</tr>
<tr>
<td>1.2 Modern Parallel Processors</td>
<td>22</td>
</tr>
<tr>
<td>1.3 Contributions of This Thesis</td>
<td>23</td>
</tr>
<tr>
<td>1.4 Outline of This Thesis</td>
<td>24</td>
</tr>
<tr>
<td><strong>2 Coherent Ray Packet Traversal Using AVX</strong></td>
<td>27</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>27</td>
</tr>
<tr>
<td>2.2 AVX Ray Packet Tracing</td>
<td>28</td>
</tr>
<tr>
<td>2.3 Results</td>
<td>29</td>
</tr>
<tr>
<td>2.4 Conclusions</td>
<td>30</td>
</tr>
<tr>
<td><strong>3 Incoherent Ray Traversal Using AVX</strong></td>
<td>31</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>31</td>
</tr>
<tr>
<td>3.2 Previous Work</td>
<td>32</td>
</tr>
<tr>
<td>3.3 Acceleration Structure</td>
<td>33</td>
</tr>
<tr>
<td>3.4 Ray Traversal Algorithm</td>
<td>36</td>
</tr>
<tr>
<td>3.5 Results</td>
<td>40</td>
</tr>
<tr>
<td>3.6 Conclusions and Future Work</td>
<td>42</td>
</tr>
</tbody>
</table>
## CONTENTS OF THE THESIS

### 4 Stackless Multi-BVH Traversal for CPU, MIC, and GPU
- 4.1 Introduction .................................................. 45
- 4.2 Related Work .................................................. 46
- 4.3 Algorithm Overview ........................................... 49
- 4.4 MBVH2 Traversal ............................................. 49
- 4.5 MBVH4 Traversal ............................................. 52
- 4.6 Implementation .............................................. 54
- 4.7 Results ....................................................... 56
- 4.8 Conclusions and Future Work ............................... 59

### 5 Interactive Ray Tracing of Large Models
- 5.1 Introduction .................................................. 61
- 5.2 Previous Work ................................................. 63
- 5.3 Method Overview ............................................. 64
- 5.4 Out-of-Core Data Structure ................................. 66
- 5.5 Memory Management ........................................ 73
- 5.6 Ray Tracing .................................................... 76
- 5.7 Results and Discussion ...................................... 79
- 5.8 Conclusions and Future Work ............................... 83

### 6 Incoherent Ray Tracing Without Acceleration Structures
- 6.1 Introduction .................................................. 85
- 6.2 DAC Ray Traversal ........................................... 86
- 6.3 Our Algorithm ............................................... 86
- 6.4 Results ....................................................... 92
- 6.5 Conclusions and Future Work ............................... 94

### 7 Conclusions
- 7.1 Future Research ............................................... 96

### A Stackless Multi-BVH Traversal Code
- A.1 CPU Code (C++) ............................................ 97
- A.2 MIC Code (C++) ............................................ 99
- A.3 GPU Code (CUDA) ......................................... 101

### B LOD Kd-Tree Traversal Algorithm

### Bibliography

### Nomenclature
Keywords: computer graphics, realistic image synthesis, ray tracing, parallel algorithms

1 Introduction

One of the most fundamental problems in computer graphics is to generate realistic or stylized images of three-dimensional virtual scenes. This process is called rendering or image synthesis. Rendering has numerous important applications in a wide variety of domains (e.g., computer-aided design, architecture, medical visualization, films, games).

In many cases, the rendered images must be as high-quality and as photorealistic as possible. Ray tracing [Gla89, SM03] is a powerful and elegant rendering algorithm that achieves this by simulating the interactions of light rays with the objects in the scene (see Figure 1).

Ray tracing is an inherently parallel task as the rays of light can be traced independently from each other. This is a very useful property since processors are becoming more and more parallel, but fully exploiting the available parallelism is a challenging problem. Another major issue is the amount of required memory to render an image.

In this thesis, we present a collection of novel high-performance ray tracing algorithms that address the problems mentioned above. These algorithms improve the computational efficiency and reduce the memory requirements of advanced ray tracing based methods on modern parallel processor (CPU, MIC, and GPU) architectures.

Ray Tracing

Ray tracing generates images by constructing light transport paths that connect pixels of the image plane with light sources in the virtual scene. The basis of physically based image synthesis is the rendering equation [Kaj86, ICG86]. Solving the rendering equation is commonly done with Monte Carlo ray tracing methods [Szi08] (e.g., path tracing [Kaj86]).

A fundamental operation in ray tracing is ray shooting, the objective of which is to find the closest intersection of a ray with the scene. The efficiency of ray shooting is one of the
key factors that determine the overall performance of a ray tracing renderer. Thus, we have chosen ray shooting as the central topic of our research.

Modern Parallel Processors

Most modern processor architectures are highly parallel and are able to exploit application parallelism at multiple levels. In this thesis, we focus on three novel processor architectures: Intel Sandy/Ivy Bridge [Int12a] (CPU), Intel Knights Corner [Int13b, Int12b] (MIC), and NVIDIA Kepler GK110 [Nvi12b, Nvi12a] (GPU).

Contributions of This Thesis

This thesis has the following main contributions:

1. The AVX instruction set introduced with the Intel Sandy Bridge architecture has doubled the peak floating point processing power of x86 CPUs. However, efficiently utilizing the 8-wide SIMD units for ray tracing is a difficult problem. We propose AVX-optimized ray traversal algorithms for both coherent and incoherent rays that provide higher performance than state-of-the-art SSE-based approaches. We use binary and 8-way branching BVHs as acceleration structures. We have measured improvements of up to 74% for coherent rays and up to 25% for incoherent rays. [Áfr11, Áfr13]

2. Extracting hidden coherence from random ray distributions requires the processing of very large ray batches [PKGH97, ENSB13]. Most hierarchical ray traversal algorithms maintain a stack, which prohibitively increases the memory requirements of tracing many rays in parallel. Consequently, the size of the ray batches must be relatively small, which leads to suboptimal coherence, and thus performance. The solution is to use
stackless approaches instead, but for some efficient acceleration structures like the multi bounding volume hierarchy (Multi-BVH or MBVH), no such algorithms have been proposed so far. We present a stackless traversal algorithm for 4-way and binary MBVHs, which replaces the regular traversal stack with a small bitstack. The bitstack encodes the per-level traversal state using skip codes. This reduces the total traversal state size by about 22–51×. We demonstrate that our approach has low computational overhead (9–31%) on the latest CPU, MIC, and GPU architectures. [ÁS13]

3. Rendering massive models consisting of hundreds of millions or even billions of primitives has many challenges. Such scenes usually exceed the size of the available memory, in which case special out-of-core rendering methods [YGKM08] must be used. Unfortunately, those usually have severe limitations in interactivity, visual quality, and shading complexity. We propose a new ray tracing based out-of-core rendering method, which supports accurate shadows and indirect illumination, and runs at interactive speeds on multi-core CPUs. Its key components are: an out-of-core hierarchical model representation that stores triangles and level-of-detail (LOD) voxels, an efficient memory management method with asynchronous I/O, and a LOD-based kd-tree traversal algorithm suitable for many ray types. Our renderer has a unique set of features, combining the advantages of previous state-of-the-art approaches. [Áfr12b]

4. Standard ray tracing methods build an acceleration structure for the scene to render, which must be updated or rebuilt every time the geometry changes. This makes rendering dynamic scenes with ray tracing much more difficult than with rasterization. A recently introduced approach called divide-and-conquer (DAC) ray tracing [KW11] eliminates the need to maintain an acceleration structure while providing competitive performance. However, very little research has been done on the implementation of this approach on parallel architectures. Another unexplored area is taking advantage of the actual ray distribution to perform more efficient primitive partitioning, which is not possible with prebuilt acceleration structures. We introduce an efficient DAC ray traversal algorithm optimized for incoherent rays and SIMD processing (using SSE and AVX instructions). In addition to these optimizations, we propose adaptive partitioning based on the active ray/primitive ratio, which reduces the partitioning overhead. We demonstrate that our approach outperforms the previous state of the art by up to 2.2×. [Áfr12a]

2 Coherent Ray Packet Traversal Using the AVX Instruction Set

Introduction

Ray packet algorithms [WSBW01, WBS07, ORM08] enable the fast tracing of coherent rays, which is especially important for real-time ray tracing solutions. One major source of perfor-
Figure 2: Single and ranged (implemented with SSE and AVX) traversal performance for primary rays in Mray/s.

The performance improvement is the use of SIMD operations provided by the CPU.

Until recently, popular CPU architectures, like the x86, were able to operate on up to 4 single-precision floating-point numbers simultaneously. This has changed with the introduction of the 256-bit AVX (Advanced Vector Extensions) instruction set [Int13a], which has doubled the SIMD width of SSE, AltiVec, NEON, etc.

We have optimized two BVH packet traversal algorithms for AVX [Áfr11]: ranged traversal [WBS07] and partition traversal [ORM08].

**AVX Ray Packet Tracing**

The smallest ray primitive of both the ranged and partition traversal algorithms is the SIMD ray, which consists of multiple rays that are traced together throughout the entire algorithm. In 4-wide SIMD implementations, a SIMD ray usually contains $2 \times 2$ rays, thus, nearby rays are packed together to maximize coherence. For AVX, we employ $4 \times 2$ SIMD rays.

Ray packets can be quite large, commonly having a size of 256 or even 1024 rays, therefore, it is important to store the ray data in a cache-friendly way. This can be achieved by using an array-of-structures-of-arrays (AoS) layout.

The performance of ray packet tracing can be improved by applying frustum culling in addition to SIMD ray techniques. We use interval arithmetic (IA) [WBS07] for culling nodes, and corner rays for culling triangles, as described in [BWS06].

**Results**

All tests were run on a system with an Intel Core i5-2400 processor. The rendering resolution was set to 1024 × 768 pixels.
3 Incoherent Ray Traversal Using AVX

Introduction

In this chapter, we propose an AVX-optimized single-ray traversal algorithm for the MBVH acceleration structure [Áfr13]. The MBVH enables high SIMD utilization for single-ray traversal; therefore, it is a good choice for incoherent ray tracing. Our approach offers higher path tracing performance than previous SSE-based methods for a wide variety of test cases.

Ray Traversal Algorithm

In our algorithm, we trace rays in batches (e.g., 256 rays), but we trace them individually. Similarly to the binary BVH traversal algorithm [WBS07], we traverse the N-way MBVH with depth-first ordered traversal, which needs a traversal stack. We use the intersection distances provided by the box test algorithm to determine the traversal order of the intersected child nodes. A straightforward way to achieve this is to do horizontal SIMD sorting [FAN07], which unfortunately is quite costly and has suboptimal SIMD efficiency.

If there are less than N intersected nodes, SIMD sorting is even less efficient because it always sorts N values. For about 90% of the valid multi-node intersections, only 1–3 children are hit. A single hit is the most likely (40–60%). Therefore, SIMD sorting almost always works at a very low efficiency, especially for wide branching factors.
We sort the nodes with a scalar method optimized for a low number of hits, which was introduced in the Intel Embree ray tracer for 4-wide MBVH ray traversal [Ern11]. This is significantly faster than SIMD sorting. The main idea of the method is to use specialized implementations of sorting for the most frequent numbers of hits. We extend the original algorithm to handle wider trees by implementing the following cases: 1, 2, 3, 4, and 5–N hits.

Results

We compared our method with MBVH4 single-ray traversal and also with MBVH RS traversal, which, unlike the other methods, is able to extract hidden coherence from rays. Our benchmark system had an Intel Core i7-3770 processor. We used SSE for the MBVH4 traversal methods and AVX for the MBVH8 ones.

The performance results in million rays per second for 8-bounce diffuse rays are shown in Figure 3. Our method, MBVH8 single traversal implemented with AVX, yields a speedup of 2–25% relative to MBVH4 for the tested ray types and scenes. Also, it is faster than MBVH RS in all our tests, including primary ray benchmarks.

4 Stackless Multi-BVH Traversal for CPU, MIC, and GPU Ray Tracing

Introduction

Ray traversal algorithms can be divided into two main categories: stack-based and stackless algorithms. Using a stack for the traversal is typically the most straightforward and efficient approach. However, if many rays are traced in parallel, the storage and bandwidth costs of maintaining a full stack for each ray can be very high.

In this chapter, we propose a new efficient stackless ray traversal algorithm for MBVHs that supports distance-based ordered traversal without restarts [ÁS13]. We add parent and sibling pointers to the tree without necessarily increasing the memory footprint, and we replace the regular stack with a compact bitstack, an integer that fits into one or two machine registers. In the bitstack we store skip codes that indicate which siblings of a node must be traversed.

Two variations of the algorithm are presented: one variation for 4-way branching MBVHs (MBVH4) and one for binary BVHs having two child boxes per node (MBVH2). The MBVH4 is primarily used on CPUs with 4-wide or 8-wide SIMD, and also on the recent Intel MIC architecture with 16-wide SIMD. On the other hand, the MBVH2 is the preferred choice on current NVIDIA GPUs [AL09, ALK12]. We have optimized our method and evaluated its performance for all these hardware platforms.
Figure 4: Test scenes used for the performance measurements of the ray traversal algorithms. The images were rendered using simple 8-bounce diffuse path tracing.
Algorithm Overview

Our algorithm replaces the stack pop of standard stack-based approaches with backtracking in the tree from the current node. The purpose of this operation is to find the next unprocessed node, which is a sibling of either the current node or one of its ancestors. To be able to ascend in the tree, we add a parent pointer to each node. We also store pointers to the siblings for accessing them without taking a round trip to the parent.

The backtracking is guided by a bitmask that encodes which part of the $N$-way tree needs to be traversed. It stores $N-1$ bits for each visited tree level (except the root level), and is updated similarly to a stack, using bitwise push and pop operations. Hence, we call this special bitmask a bitstack. The per-level values in the bitstack are skip codes. These indicate which siblings of the most recently visited node on the respective level must be skipped.

Results

We evaluated the performance of our stackless traversal algorithms and the corresponding stack-based ones using a simple but highly optimized diffuse path tracer on all three architectures: Intel Core i7-3770 (CPU), Intel Xeon Phi SE10P (MIC), and NVIDIA Tesla K20c (GPU).

The performance results are shown in Figure 5. Our stackless algorithms, similarly to previous methods, are somewhat slower than the reference stack-based ones when used for ordinary ray tracing; however, they maintain about 22–51× smaller traversal states. For our test scenes (Figure 4), stackless traversal is slower by 9–17% on the CPU, 13–16% on the MIC, and 20–31% on the GPU.
5 Interactive Ray Tracing of Large Models Using Voxel Hierarchies

Introduction

This chapter presents a new massive model rendering method based on ray tracing \cite{Afri12b}, efficiently combining the advantages and techniques of different existing approaches.

Several testing examples demonstrate that our method works effectively for different types of complex models, achieving interactive frame rates on a quad-core desktop PC. It supports a wide variety of ray traced shading algorithms, which include direct lighting with shadows, ambient occlusion, and global illumination (see Figure 6).

Method Overview

We first construct a hierarchical out-of-core data structure, which contains, in a compressed format, the original triangles and several LOD levels consisting of voxels.

Thanks to the hierarchical LOD mechanism, it is possible to render huge data sets that cannot be completely loaded into the system memory. During rendering, we load the necessary details asynchronously, thus, there is no stuttering due to insufficient available data.

We organize all primitives (i.e., the triangles and voxels) into a \textit{kd-tree}. This out-of-core \textit{kd-tree} has a dual purpose in our approach: it speeds up the ray intersections with the triangles, and stores the voxel hierarchy.

A subset of the \textit{kd-tree} nodes contain a single \textit{LOD voxel}, which is a primitive rendered as an axis-aligned box. It roughly approximates the original primitives stored in the subtree of the corresponding node and holds \textit{shading attributes} (e.g., normal, color) per box face.

The entire \textit{kd-tree} is decomposed into \textit{treelets}, which are grouped into equally sized \textit{blocks}. In order to reduce storage requirements, the blocks are encoded using a lossless data compression algorithm.

We employ a custom, purely software-based \textit{memory manager}, which is responsible for the loading of the blocks required by the renderer.

The tight integration of the LOD levels with the acceleration structure enables an efficient model representation and ray traversal algorithm. By using LOD voxels, significantly higher frame rates can be achieved, with minimal loss of image quality. We provide fast LOD error metrics for primary, shadow, ambient occlusion, and diffuse interreflection rays.

Results

All benchmarks were performed on a desktop PC with an Intel Core i7-2600 CPU, 8 GB RAM, an NVIDIA GeForce GTX 560 Ti GPU, and two 7200 RPM hard disks in RAID 0 setup.
Figure 6: The BOEING 777 model (337M triangles) rendered interactively with shadows and one-bounce indirect illumination.
We have selected test models from different application domains: Power Plant (12M triangles), Asian Dragon (7M triangles), Lucy (28M triangles), MPI v1.0 (73M triangles), Boeing 777 (337M triangles), and Mandelbulb (354M triangles).

The scaling of the ray casting performance with the number of triangles is illustrated in Figure 7. Notice that without LOD, the render time increases logarithmically, as expected from employing a kd-tree as an acceleration structure. However, if we enable the use of LOD voxels, the performance becomes nearly constant after a certain point.

We demonstrate that even the most complex models from the test suite can be rendered at interactive speeds with our approach.

6 Incoherent Ray Tracing Without Acceleration Structures

Introduction

A ray tracer typically consists of two main parts: ray traversal and acceleration structure building. Keller and Wächter [KW11] recently proposed a largely different and elegant approach called divide-and-conquer ray tracing, which does not require an acceleration structure.

In this chapter, we propose a new DAC traversal algorithm [Áfr12a] based on the core method by Keller et al. Our approach is generally more efficient than Mora’s method [Mor11], and it exploits the AVX instruction set. We have optimized our method for incoherent rays.

Ray Filtering

The filtering can be executed in-place by rearranging the ray list to create an active and an inactive partition. A ray is specified using a point of origin, a direction vector, an interval
[0, t_{max}] defining a line segment, and an ID. The total size of a ray is 32 bytes, which means that it fits into a single AVX register or two SSE registers.

We avoid caching problems by simply reordering the rays in the original array. Rays can be quickly copied in blocks of 32 (with AVX) or 16 bytes (with SSE).

We simultaneously intersect 4 rays when using SSE and 8 rays when using AVX. Before doing so, the ray data, which consists of 8 values per ray, must be rearranged into SoA (structure-of-arrays) format.

**Triangle Partitioning**

Triangle partitioning divides a list of triangles into two disjoint sublists. We use two different partitioning methods: *middle partitioning* and *SAH partitioning*.

The partitioning algorithms do not process the triangles themselves, but only their AABBs (axis-aligned bounding boxes), which we precompute. In contrast with ray filtering, we manage a triangle ID array instead of directly reordering the AABBs.

Always partitioning with the SAH does not necessarily lead to the highest possible ray tracing performance. We solve this problem by adaptively deciding between SAH and middle partitioning. In each partitioning step, the ratio of the number of active rays and current triangles is checked against a predefined threshold (e.g., 1–2).

**Triangle Intersection**

In our method, a special triangle representation is used to save memory space and bandwidth. Similarly to the ray filtering routine, multiple rays are intersected with the current triangle using SIMD.

**Ordered Traversal**

For primary rays, front-to-back traversal has a significant positive impact on the ray tracing speed. However, the improvement is small for incoherent rays. We determine the traversal order with the very cheap approach from the packet tracer by Wald et al. [WBS07].

**Results**

The benchmarks were run on two different systems: on an Intel Core i7-960 with 24 GB RAM (triple channel), and on an Intel Core i7-2600 with 8 GB RAM (dual channel). We tested the algorithms using a 1-bounce and an 8-bounce Monte Carlo path tracer with diffuse reflections.

Our method is quite competitive to a highly optimized static ray tracer that uses the MBVH acceleration structure [Ern11]. For example, MBVH is only 12% faster for the 8-bounce path tracing of the *CONFERENCE* scene, on a single thread of the i7-960. However, the difference is greater on multiple threads, especially for *HAIRBALL*, where our method is 4× slower.
Compared to Mora’s method, our approach filters rays more efficiently, uses higher quality object partitioning, exploits wider SIMD, and is optimized for incoherent rays (see Figure 8).

7 Conclusions

In this thesis, we investigated high-performance ray tracing on modern parallel processor architectures. Specifically, we proposed methods that better exploit the available parallelism and memory of latest-generation hardware, and enable superior image quality and interactivity than previous approaches. We provided efficient solutions for both offline and real-time rendering.

Possible future research directions include: MBVH4 ray traversal on GPUs, general purpose out-of-core ray tracing, higher quality voxel-based LOD, and DAC ray tracing on massively parallel processors.


BIBLIOGRAPHY


