TDT24 presentation

Holger Ludvigsen

September 29, 2009

Part I

Ray Tracing

Ray tracing is a technique for generating an image of three-dimensional scene. The technique consists of tracing imaginary rays from the position of the camera through a plane towards the objects in the scene. The rays intersect and bounce around the scene like real light rays do. Each ray contributes to their part of the plane which in the end will be the generated image. See figure 1.

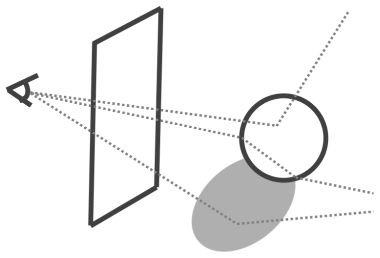


Figure 1: The ray tracing technique

The ray tracing computational problem

There are two big tasks when ray tracing:

1. Finding what object in the scene a ray intersects first
2. Finding the exact location and shading of the point on that object

With a complex scene, the former task take up to 95[1] % of the time spent ray tracing.

Acceleration structures

To speed up the task of finding intersecting objects, we put the geometry in acceleration structures. A common acceleration structure is the bounded volume hierarchy (BVH). This structure is a spatially organized tree with the objects as leaf nodes. See figure 2. When the geometry of the objects change, the BVH needs to be updated.

[1] Turner Whitted. An improved illumination model for shaded display. Commun. ACM, 23(6):343-349, 1980.

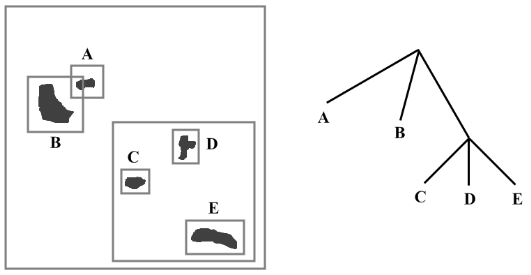


Figure 2: A scene with associated bounding volume hierarchy

Interactive and real time ray tracing

By interactive ray tracing we mean a continously updated view of a scene at about 1-5 frames per second. This enables instant visual feedback during e.g. design of automobiles or making a Toy Story like movie. To be referred to as real time, we need about 20-30 or more frames per second.

Part II

Summary of

Timo Alia and Samuli Laine
NVIDIA Research

Understanding the Efficiency on Ray Traversal of GPUs

In this paper, the authors point out that very little is known about the performance of ray tracing kernels and their theoretical optimal performance. With this in mind, the made a GPU simulator that executes kernels and determine their performance under theoretical optimal conditions. When testing kernels implementing previously known methods for ray tracing on a real GPU, they observed performance of only about 40-60 % of theoretical optimum in the simulator. The authors then analyze this and propose some simple techniques to push the performance closer to the method's theoretical limits, and they achieve about 90 % of optimum.

SIMT and SIMD efficiency

The GT200 generation of Nvidia GPUs does SIMT, which means single instruction, multiple threads. SIMT is a superset of SIMD. In SIMD, the same instruction is executed in parallel on different data. In SIMT, multiple threads (a warp) start at the same instruction with different data, but they are allowed to branch. Threads on the same path continue to execute with other threads in the warp disabled. Then vice versa with the other paths until all threads are on the same instruction.

SIMD efficiency The percentage of threads that are not disabled in a warp.

The test system

- Nvidia GTX285 for real performance
- Custom built simulator for theoretical optimal performance
- Hand-optimized CUDA 2.1 assembly code as kernels
- Bounding volume hierarchy as acceleration structure
- Three scenes at 282K, 174K and 80K triangle polygons

Details about the simulator

- Optimal dual issue rate: A streaming multiprocessor has 8 streaming processors, but also 2 special-function units (SFU). Every instruction that can execute in the SFU is assumed to do so.
- All memory accesses are hidden by work (simulated by returning immediately).

Ray tracing methods tested

The main difference between the methods is how they traverse the BVH tree.

Packet traversal

In this scheme, we group rays into warp sized *packets*. The rays of a packet traverse the tree together by sharing the traversal stack. This means that some rays will visit many nodes that they do not intersect with. The benefits are coherent memory accesses and lesser branching in a warp becouse grouped rays usually hit the same objects.

Per-ray traversal

In this scheme, each ray traverses the tree independently with their own traversal stack. This is theoretically superior to packet traversal since each ray only visits nodes they intersect. The authors use two implementations for per-ray traversal: "while-while" and "if-if".

While-while

```
while ray not terminated do
  while node does not contain primitives do
    traverse to the next node
  end while
  while node contains untested primitives do
    Perform a ray-primitive intersection test
  end while
end while
```

If-if

```
while ray not terminated do
  if node does not contain primitives then
    traverse to the next node
  end if
  if node contains untested primitives then
    Perform a ray-primitive intersection test
  end if
end while
```

Test results and interpretation

Table 1 in the paper shows the results from simulation and real system testing. I have compiled this into figure 3 by taking the average across all scenes and across all ray types.

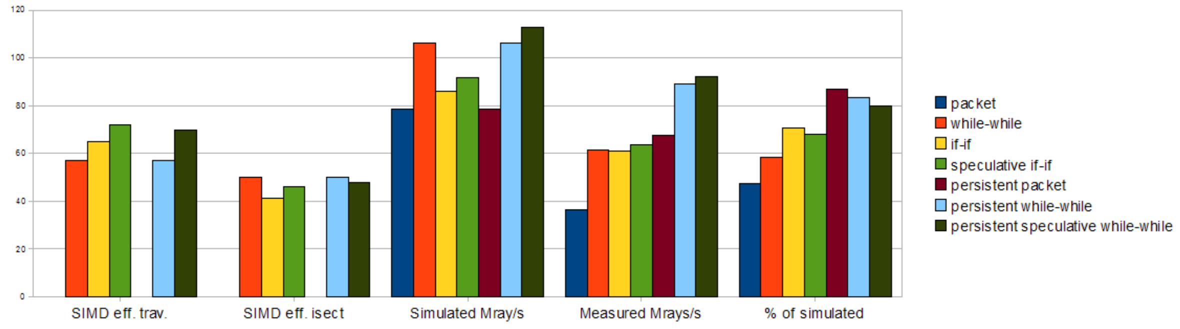


Figure 3: A compilation of table 1 in paper

Per-ray traversal has less coherent memory accesses. Since the simulator hides memory accesses, it was expected that the difference between real and simulated performance would be bigger for per-ray than packet traversal. However, the gap was actually smaller for packet traversal. This indicates that memory bandwith is actually not the major issue.

In simulations, if-if was about 20 % slower than while-while because of less coherent memory accesses. But their real performance was about the same. Additional testing reveiled the reason for this: if-if leads to fewer *exceptionally* long-running warps than while-while. Such long-running warps lead to under-utilization of the cores due to inconsistent execution times.

Improvement: Persistent threads

The idea is to launching just enough threads to fill the GPU. These *persistent* threads are long-running and fetch work from a global pool. The implementations of packet traversal and while-while per-ray traversal using persistent threads increased performance to about 80-90 % of theoretical optimum. Due to ideal conditions in the simulator. This is as high as one can expect to get. The authors claim this gives the fastest ray tracer to date[2].

[2]paper is published August 2009

Further improvements

The authors also include ideas for further improvement. The first is *speculative traversal.* This improves performance of ray tracing of primary rays by 5 %, but for other ray types the performance is the same or lower. The rest of the improvements does not pay off at the moment, but the simulations showed that by adding some instructions to the GPU they become beneficial:

population count Returns the number of threads for which a condition is true

prefix sum Returns a unique index to threads for which a condition is true

Speculative traversal

The idea here is that if some rays in a warp are going to traversal, then all rays can do the traversal even though they have found an object they want to do intersection testing with. If the intersection testing fails, then the work of further traversal is already done. If the intersection test indicated a hit, then the overhead of unnecessary traversal is not that high. Speculative traversal improves SIMD efficiency.

Improvements requiring new instructions (to be effective)

Replacing terminated rays: Replace persistent threads that have terminated with new ones that start from the root. The computational overhead is halved by the two new instructions.

Work queues: Utilize a work queue to obtain almost perfect SIMD efficiency. In simulation, almost 100 % efficiency was obtained.

Wide trees: Use wide tree acceleration structures with many branches to enhance memory access coherence.

Part III

Summary of

James Bigler, Abe Stephens and Steven G. Parker
University of Utah

Design for Parallel Interactive Ray Tracing Systems

In this paper, the authors present an architecture for an interactive ray tracer running on a parallel computer, and explain the considerations that needs to be taken into account for such a system. It is not directed They have implemented their ideas as the freely available open-source Manta ray tracer, and briefly present some applications where it was used.

Hardware trends

- Early interactive ray tracers were implemented on supercomputers
  - clusters
  - shared memory
  - high-performance network.
- Today, processors are becoming parallel and has high enough performance to do ray tracing
  - SIMD
  - single chip systems

Software design philosophy

- Scalable, flexible and configurable
- Do not enforce specific acceleration structure or other ray tracing techniques
- Develop components that follow design pattern, implement acceleration structures, primitives, material shaders and rendering techniques in this framework
- Flexibility imposes overhead and parallelism challenges, but overcome this by using wide ray packets and pipeline models

Manta architecture

Manta consists of two groups of modular components: the pipeline and the rendering stack. Each component has an API. The primary structure for ray tracing data is the wide ray packet.

Wide ray packets

The wide ray packets contain individual rays plus all of the data needed for intersection routines, shaders and other components. Ray packets are wider than SIMD vectors and are processed in smaller pieces in tight loops. Expensive values are lazily evaluated. A flag is set in ray packet that other components can check if they need the same value. The authors found that the ray packet size should be 64 rays to fit in L1 cache. During tracing, a subset of rays in a packet probably will hit other objects than the rest. Manta manages this by continuously splitting packets into coherent sections. This is done by finding a sequential run of rays with common characteristics in the packet and make this a new smaller packet. Splitting ensures coherence, but impose a challenge in keeping the data alignement in the packets right.

Parallel pipeline

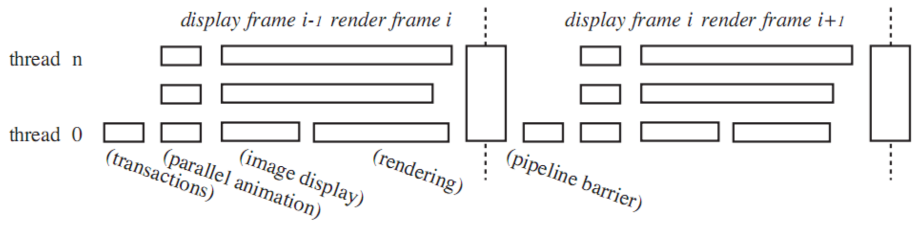


Figure 4: Two stages of the parallel pipeline

Figure 4 shows the parallel pipeline. For each stage, frame i is displayed and frame i+1 is rendered. To achieve high performance, threads are synchronized only at certain points in the pipeline. This is done before the *transactions* phase. Tasks that are easy to load balance, like parallel animation, are then executed. Then imbalanced tasks such as image display is executed. In the typical configuration, image display is done by one thread. Lastly done are tasks that are dynamically load balanced, like rendering.

The transaction phase handles updates from the user interface, changes in acceleration structures and other events that can lead to race conditions. The transactions are callbacks to execute code that modifies the state at a ”safe” time. The transactions are managed as a queue. Having a transaction phase ensures that events that are not synchronized with the pipeline is handled safely without excessive synchronizations.

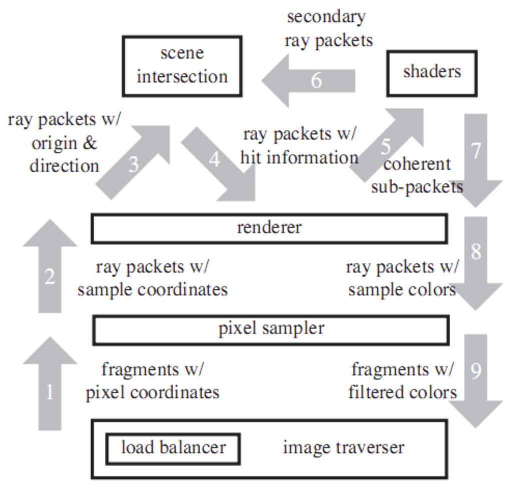


Figure 5: Rendering stack during rendering stage in pipeline

Rendering stack

Figure 5 shows the renderings stack that is executed in the rendering stage of the pipeline. The process is as follows:

1. The *image traverser* divides frame into regions and assigning regions to threads. A *load balancer* in the image traverser ensures that the work is distributed as evenly as possible.

2. The *pixel sampler* map rays to samples in the regions an organizes these in ray packets.

3. The *renderer* traces the rays through the scene and invokes material shaders upon intersection. The renderer is also responsible for splitting packet into coherent subpackets.

Manta applications

The author's implemented Manta ray tracing system has been used for several applications. Some worth mentioning is massive model visualization, direct iso-surface rendering and multi-modal visualization.

Massive model visualization

Massive models have 100s of millions of triangles. Ray tracing scales sub-linearly in number of objects due to acceleration structures. Thus, ray tracing is espe-

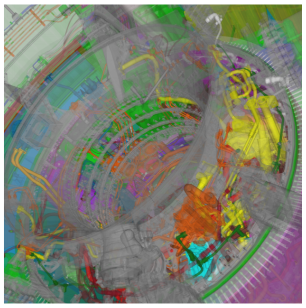


Figure 6: Massive model visualization

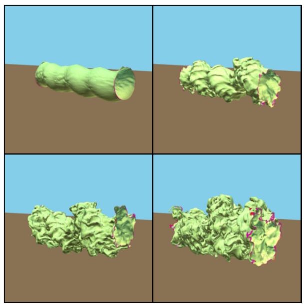


Figure 7: Direct iso-surface rendering

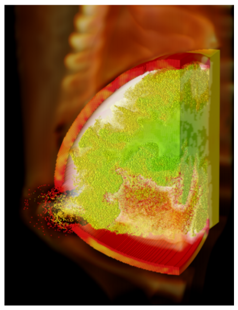


Figure 8: Multi-modal visualization

cially suitable for massive model visualization. Figure 6 shows a model of a Boeing 777 engine that consists of 350 million triangle polygons.

Direct iso-surface rendering

An iso-surface is a surface in space where a 3D function has a specific value. This is suitable for ray tracing since the surface function can be used directly in intersection calculation. Figure 7 shows an iso-surface representing the boundary between a fast flowing and stationary fluid.

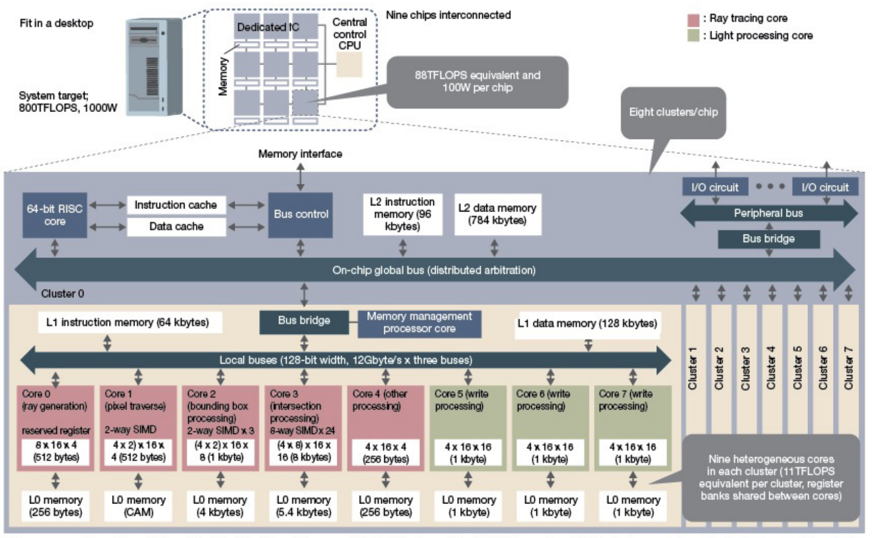
Multi-modal visualization

Multi-modal visualization means that there are different representations of primitives, such as voxels (volumetric pixels), spheres or polygons. Ray tracing enables visualization of different types at the same type. Figure 8 shows a visualization of a gas container rupturing because of heat. Both spheric gas particles and volumetric fire is shown in the same visualization.

Appendix A

800 TFLOP GPU ray tracing super computer

(From `http://techon.nikkeibp.co.jp/article/HONSHI/20090629/172373/`)


Fit in a desktop
System target; 800TFLOPS, 1000W
Dedicated IC
Central control CPU
Memory
Nine chips interconnected
88TFLOPS equivalent and 100W per chip
🟥 : Ray tracing core
🟩 : Light processing core
Eight clusters/chip
Memory interface
64-bit RISC core
Instruction cache
Data cache
Bus control
L2 instruction memory (96 kbytes)
L2 data memory (784 kbytes)
I/O circuit
I/O circuit
Peripheral bus
Bus bridge
On-chip global bus (distributed arbitration)
Cluster 0
L1 instruction memory (64 kbytes)
Bus bridge
Memory management processor core
L1 data memory (128 kbytes)
Local buses (128-bit width, 12Gbyte/s x three buses)
Core 0 (ray generation) reserved register 8 x 16 x 4 (512 bytes)
Core 1 (pixel traverse) 2-way SIMD 4 x 2) x 16 x 4 (512 bytes)
Core 2 (bounding box processing) 2-way SIMD x 3 (4 x 2) x 16 x 8 (1 kbyte)
Core 3 (intersection processing) 8-way SIMD x 24 (4 x 8) x 16 x 16 (8 kbytes)
Core 4 (other processing) 4 x 16 x 4 (256 bytes)
Core 5 (write processing) 4 x 16 x 16 (1 kbyte)
Core 6 (write processing) 4 x 16 x 16 (1 kbyte)
Core 7 (write processing) 4 x 16 x 16 (1 kbyte)
L0 memory (256 bytes)
L0 memory (CAM)
L0 memory (4 kbytes)
L0 memory (5.4 kbytes)
L0 memory (256 bytes)
L0 memory (1 kbyte)
L0 memory (1 kbyte)
L0 memory (1 kbyte)
Cluster 1
Cluster 2
Cluster 3
Cluster 4
Cluster 5
Cluster 6
Cluster 7
Nine heterogeneous cores in each cluster (11TFLOPS equivalent per cluster, register banks shared between cores)

Interconnecting Nine Chips, Each with Nine Cores x Eight Clusters Each IC integrates eight clusters, and each cluster consists of nine heterogeneous cores specifically designed for ray tracing. Nine of these ICs are interconnected to achieve a target system performance of 800TFLOPS. Core internal values shown are data register width (bytes) x number of registers x number of banks.

Figure A.1: 800 TFLOPS Multicore IC for Realtime Ray Tracing

Japanese company Tops Systems Corp has announced a 800 TFLOP GPU chip specialized for real time ray tracing in a desktop system. The chip is intended for the automobile design industry. The design is being developed by Toyota and Unisys and consists of nine identical interconnected integrated circuits, each with nine clusters of eight cores. The total 73 cores are expected to run at 750 MHz. Figure A.1 shows the design of each of the nine ICs.