Ray Tracing on the Cell Processor

Carsten Benthin[†] Ingo Wald[◇] Michael Scherbaum[†] Heiko Friedrich[‡]

[†] *in*Trace Realtime Ray Tracing GmbH [◇] SCI Institute, University of Utah [‡] Saarland University
{benthin, scherbaum}@intrace.com, wald@sci.utah.edu, heiko@graphics.cs.uni-sb.de

Abstract

Over the last three decades, higher CPU performance has been achieved almost exclusively by raising the CPU's clock rate. Today, the resulting power consumption and heat dissipation threaten to end this trend, and CPU designers are looking for alternative ways of providing more compute power. In particular, they are looking towards three concepts: a *streaming* compute model, vector-like *SIMD units*, and *multi-core architectures*. One particular example of such an architecture is the Cell Broadband Engine Architecture (CBEA), a multi-core processor that offers a raw compute power of up to 200 GFlops per 3.2 GHz chip. The Cell bears a huge potential for compute-intensive applications like ray tracing, but also requires addressing the challenges caused by this processor's unconventional architecture. In this paper, we describe an implementation of realtime ray tracing on a Cell. Using a combination of low-level optimized kernel routines, a streaming software architecture, explicit caching, and a virtual software-hyperthreading approach to hide DMA latencies, we achieve for a single Cell a pure ray tracing performance of nearly one order of magnitude over that achieved by a commodity CPU.

Keywords: Ray tracing, Cell processor, multi-core.

1 Introduction

In the last decades, most increases in CPU performance came from raising the CPU's clock rate (to nowadays more than 3.6 GHz). This rise in clock rate was made possible through an exponential increase in the circuits available on the die (a process widely known as Moore's law [15]). On one hand, this increase in circuits allowed for putting more and more computational units into hardware: while first CPUs were rather simple, they gradually evolved to have floating point units, SIMD units, complicated branch prediction, etc. More importantly, however, a higher integration density implies shorter circuit paths on the chip, which allows for higher clock rates. Together, these two factors have laid the foundation for the staggering increase in serial (i.e., single-threaded) CPU performance that could be witnessed over the last few decades.

Nowadays, this exponential increase of serial CPU performance seems to slowly come to a grinding halt. First, for single-threaded CPUs there is a diminishing return in having more transistors: while early CPUs could be improved by adding more computational units, e.g. integer or floating point units, it is increasingly difficult to extract enough instruction level parallelism out of the serial instruction stream to keep these additional units busy. Second, memory speed is not rising as fast as clock speed, so cache misses become ever more costly; as a result, more and more die space has to be spent on complex branch prediction, out-of-order execution, large caches, etc just to sustain such high clock rates at all. Finally, power and heat dissipation become an ever more challenging problem.

Thus, CPU designers are currently looking for alternatives to the approach of simply raising the clock rate to increase compute power. In particular, they are looking at "streaming" (or "broadband") architectures[1] to hide memory latencies, at exposing parallelism through SIMD units, and at multi-core architectures. At least for specialized tasks such as triangle rasterization, these concepts have been proven as very powerful, and have made modern GPUs as fast as they are; for example, a Nvidia 7800 GTX offers 313 GFlops [16], 35 times more than a 2.2 GHz AMD Opteron CPU. Today, there seems to be a convergence between CPUs and GPUs, with GPUs—already using the streaming compute model, SIMD, and multi-core—become increasingly programmable, and CPUs are getting equipped with more and more cores and streaming functionalities. Most commodity CPUs already offer 2–8 cores [1, 9, 10, 25], and desktop PCs with more cores can be built by stacking and interconnecting smaller multi-core systems (mostly dual-processor boards). Commodity GPUs have even more cores (e.g., a ATI Radeon X1900 has 48 SIMD pixel processors), making it likely that this will continue.

One newer, more radical design is the Cell processor. Instead of slowly evolving towards a streaming SIMD multi-core architecture, the Cell processor was designed from scratch with these concepts in mind. As such, it is located somewhere in-between a CPU and a GPU, offering a 200 GFlops per 3.2 GHz Cell (which is competitive with a GPU), while in terms of programmability being much closer to a CPU than to a GPU. The Cell processor bears an interesting potential in particular for realtime ray tracing. Ray tracing is well-known to benefit from parallelism [17, 18], and most of the more recent ray tracing algorithms are already capable of exploiting SIMD operations [30, 21, 27, 28].

Despite its huge potential, as we detail below the Cell is quite different from more conventional CPUs, and mapping existing ray tracing knowledge to it is more complicated than just re-compiling. In this paper, we investigate how a ray tracer can efficiently be mapped to the Cell. In particular, we have to deal with how to map the ray tracing algorithm to the individual cores, how to design the kernel operations to run fast on a Cell core, and how to deal with the Cell processor's unique approach of accessing memory.

1.1 Outline

The paper starts with a description of the Cell processor in Section 2, discussing its unique architecture and the differences to conventional CPU architectures. Section 3 illustrates the impact of the Cell's unique architecture in terms of fast ray tracing, in particular discussing the efficiency of different implementation approaches and other fundamental high level design decisions. Based on these high-level decisions, efficient software-managed caching and optimized ray tracing kernels (based on bounding volume hierarchies) are presented in Section 4 and Section 5. In order to hide memory latency, Section 6 presents an implementation of fast software-based hyperthreading, which extends the fast ray tracing kernel. Besides an optimized kernel for ray tracing, Section 7 introduces a framework for efficient shading, which in combination with the optimized parallelization framework of Section 8 makes complete ray tracing possible in realtime. Note that for secondary rays, the current implementation does only support shadow rays. We summarize and discuss our approach in Section 9, and conclude in Section 10.

[1] A streaming compute model is one in which small kernels operate on entire *streams* of data items, such as shading a stream of fragments [13].

2 The Cell Broadband Architecture

Before discussing how to map ray tracing to the Cell processor, we first discuss the unique features of this architecture, in particular where it differs from traditional CPU architectures. One of the most important differences to conventional multi-core CPUs is that the Cell is not a *homogeneous* system with multiple copies of the same core [11]. Instead, it is a *heterogeneous* system, consisting of one 64bit PowerPC core (PPE) and eight "synergistic co-processor elements" (SPEs), each of which contains a "synergistic processing unit" (SPU) (see Figure 1). The SPE differs from standard CPU core as well, in that it is much smaller, and exclusively designed for streaming workloads. In its intended use, the Cell's SPEs do the 'real' work in a data parallel or streaming (pipeline) manner, while the PPE core performs synchronization tasks and executes non-parallizeable code.

Instead of a memory cache, each SPE has 256KB of "local store" memory. Direct access to main memory from the SPE is not possible. Instead the SPE can explicitly perform (asynchronous) DMA transfers to or from main memory. The local store has to accommodate both data and code. The SPEs have their own reduced RISC-like instruction set, where each instruction is a 32bit word with a fixed execution latency of 2-7 cycles (double precision floating point instructions are not considered here).

Compared to standard CPUs, each SPU has a rather large unified register file of 128 $\times$ 128bit SIMD registers. Most of the SPE instructions are SIMD instructions, in particular for performing multimedia and general floating-point processing. These instructions are more flexible than e.g. Intel's SSE/SSE2/SSE3 instruction sets [7] and include, for example, three-operand instructions with throughput of one SIMD multiply-add per cycle, allowing for a theoretical peak performance of 25.6 (single-precision) GFlops on a 3.2 GHz SPE. Each SPE has two pipelines, each specialized for a certain type of instructions (load/store vs. arithmetic). This allows for dispatching two (independent) instructions in parallel, achieving a theoretical throughput of 6.4 billion SIMD instructions per second per SPE. Besides the special instruction set, the SPE has no branch prediction as known from standard CPU cores. Instead a branch hint instruction is supported which helps the SPE to predict if a certain branch has to be taken or not.

Both PPE and SPEs are in-order processors. As the 256KB local store of the SPE has a fixed access latency (of 7 cycles), in-order instruction execution is a suitable simplification: The compiler can predict memory access and therefore schedule the instructions for maximum performance. However, in-order execution on the PPE with its standard 32 KB L1 and 512 KB L2 cache harms memory intensive applications, making the core less powerful compared to standard CPUs with out-of-order execution. In order to reduce the impact of cache misses, the PPE uses 2-way symmetric multi-threading [11] which is comparable to Intel's Hyperthreading [8].

To keep the SPEs supplied with data—and to allow efficient communication between the SPEs—the Cell uses a high-performance element interconnection bus (EIB). The EIB is capable of transferring 96 bytes per cycle between the different elements—PPE, SPEs, I/O interface, and system memory. The DMA engine can support up to 16 concurrent requests per SPE, and the DMA bandwidth between the local store and the EIB is 8 bytes per cycle in each direction. The aggregate bandwidth to system memory is 25.6 GB/s, and the bandwidth between SPEs can be more than 300 GB/s on a 3.2 GHz Cell. SPE-to-SPE transfers are kept within the EIB, avoiding any main memory transaction.

The minimalistic design of the SPEs (no cache, no branch prediction, in-order execution, ...) allows for very high clock rates. For experiments we had access to a dual Cell-blade evaluation system (with 512 MB of XDR main memory) in which the Cells are clocked at only 2.4GHz. The PlayStation 3 will clock its Cell at 3.2 GHz, and even more than 4 GHz have shown to be possible [4].

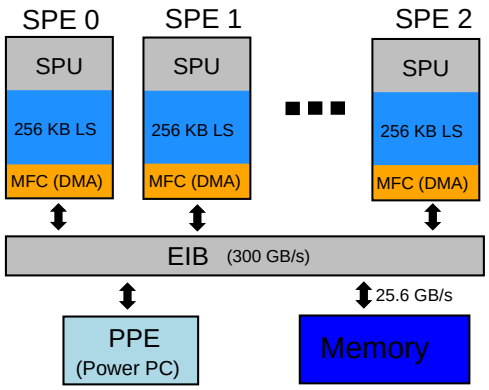

Figure 1: Each Cell consists of a 64bit PowerPC core (PPE) and eight "synergistic co-processor elements" (SPEs). Each SPE has 256 KB local store, a memory flow controller (MFC) and an "synergistic processing unit" (SPU) with a SIMD processing unit and 128 registers of 128 bits each. An "element interconnection bus" (EIB) with an internal bandwidth of more than 300 GB/s (per 3.2 GHz Cell processor), is responsible of transferring data between the SPEs. The maximum bandwidth from the SPEs to main memory is 25 GB/s.

3 Ray Tracing on the Cell

As just shown, the Cell is more powerful than a commodity CPU, but also quite different. In order to enable efficient ray tracing on the Cell, the following differences have to be taken care of:

In-order Execution and SIMD Instructions: An SPE executes the instructions in-order, which means that pipeline stalls, caused by code dependencies or mispredicted branches, are more expensive than on a CPU with out-of-order execution. To avoid this, the compiler is responsible for a suitable instruction scheduling and to untangle code dependency chains. Most of the time the compiler resolves the dependencies automatically, but sometimes the algorithms have to be (manually) adapted to help the compiler find independent instruction sequences. These instruction sequences can then be interleaved to prevent stalls efficiently.

As the SPE's instruction set is designed for SIMD processing, most of the instructions operate on multiple data elements at once (two to sixteen elements depending on element size). As an instruction has a throughput of one per cycle and a latency between 2-7 cycles, one has to ensure enough independent data to work on. Otherwise, dependency chains, and therefore pipeline stalls, are unavoidable. Unfortunately, the instruction set is sub-optimal for scalar code, so even simple operations such as increasing an unaligned counter in memory require a costly read-modify-write sequence.

Memory Access: Each SPE has an explicit three-level memory hierarchy: a 128 $\times$ 128bit register file, a 256 KB local store, and main memory. As the local store does not work as hardware-managed memory cache, all main memory accesses must be done explicitly by DMA transfers. Even though the memory bandwidth of 25.6 GB/s is rather high, each memory access has a high latency of several hundred SPE clock cycles. In order to hide the latency, the DMA engine supports asynchronous transfers, whose states can be requested on demand. Even though this setting is ideal for streaming operations in which huge blocks of data are being processed sequentially, it is challenging for a data-intensive application with irregular memory accesses such as a ray tracer.

Parallel Execution on multiple SPEs: Each Cell has 8 SPEs; a dual-Cell system has 16. There are different ways of mapping an algorithm onto such a parallel architecture, and the exact way this is done will have a significant impact on performance.

3.1 Programming Model

The first design decision to consider is how to map the ray tracing algorithm to the multiple SPEs. In a *heterogeneous* approach, each SPE runs a different kernel, and the results are sent from one SPE to another. In a *homogeneous* approach, each SPE runs a full ray tracer, but on different pixels. Traditional multi-core architectures favor the latter, but traditional streaming architectures are usually intended to be used in the heterogeneous way (also see [13, 20, 6]).

The work performed by a ray tracer can be broken into the following tasks: generating primary rays, traversing rays through a spatial index structure, intersecting the rays with geometric primitives, and shading the corresponding intersection points, including the generation of secondary rays and their recursive evaluation.

One way of mapping ray tracing to the Cell is to have each SPE perform only one of these tasks, and to send its results to the SPE that performs the next task in the chain. For example, one SPE could generate primary rays which are then sent to one or more SPEs doing the traversal, which in turn send ray-triangle intersection tasks to other SPEs. In fact, the Cell's architecture is able to support such a streaming workload: the high inter-SPE bandwidth (of up to 300GB/s) lets the SPEs communicate with each other; and the asynchronous DMA transfers allow for transferring one SPE's output to another while both operate on the next block of data in the stream. In principle, mapping a ray tracer to such an environment is possible, and has been demonstrated for both Smart Memories [6] and GPUs [20].

The streaming approach works best if the individual tasks can be cascaded and if there is a steady flow of data from one task (i.e., SPE) to the next (as in video or speech processing, or in scientific computations) [31]. A ray tracer, unfortunately, has a much more complex execution flow: the traversal unit does not pass results unidirectionally to the intersection unit, but also has to wait for its results; the shader not only shades intersection points, but can also trigger additional rays to be shot; etc. Such dependency chains require one task to pause and wait for the results of another, creating stalls. In addition, this approach makes it hard to balance the load of different SPEs, as the relative cost of traversal, intersection, shading, etc., varies from pixel to pixel. As soon as one SPE in the chain becomes a bottleneck, it starves the other ones.

To a certain degree this starvation can be avoided by buffering the SPEs' in- and outputs in main memory, and then frequently switch the kernels each SPE executes depending on what tasks need to be done most. This implies a non-trivial system design (synchronization, load-balancing, etc), and also poses significant strain on the memory system (whose bandwidth is more than an order of magnitude lower than the inter-SPE bandwidth). Even though 25 GB/s seem plentiful, we target several dozen frames per second, each frame requiring at least one million rays of at least 32 bytes each, and reading/writing each ray several times when passing it from task to task might easily create a bottleneck.

The above considerations have led us to follow an approach typically used on conventional shared-memory multiprocessor architectures: each SPE independently runs a full ray tracer, and parallelization is achieved by SPEs working on different pixels (see Section 8). Having each SPE work independently ensures less communication between SPEs, and avoids exchanging intermediate results with either other SPEs or main memory. In addition, it avoids dependency chains between different SPEs' in- and outputs, and facilitates high SPE utilization. On the downside, having each SPE run a full ray tracer forces us to operate the SPEs in a way they are not designed to be used: in particular, each SPE may access any data in the scene database, in random-access manner. As the local store is too small to store the entire scene, this requires appropriate ways of accessing and caching the scene data, as well as means to handle the resulting SPE-memory dependencies.

3.2 Spatial Index Structure and Traversal Method

Having decided on the programming model, the next decision is which spatial index structure to use. Efficient ray tracing requires the use of efficiency data structures, such as bounding volume hierarchies (BVHs) [22], Grids [2], or kd-trees [12]. In particular, tracing coherent packets of rays [30]—possibly accelerated by looking at the packet's bounding frustum [21]—has been shown to be an important factor in reaching high performance [26, 3]. Tracing packets of rays allows for amortizing memory accesses over multiple rays, allows for efficiently using SIMD extensions, and increases the compute-to-memory access ratio. Though already important for a conventional CPU, these advantages are even more important for a Cell, which depends on dense SIMD-code, and for which memory accesses are even more costly than for a standard CPU.

Though originally invented for kd-trees [26, 21, 3] such coherent traversal schemes are now also available for grids [28], and for BVHs [27]. As grid and BVH allow for handling dynamic scenes, we focus on these two. Among those, the Grid is more general in the kind of animations it supports, and its more regular structure would nicely fit a streaming architecture. For example, a straightforward extension of the technique proposed in [14] would allow for prefetching the grid cells before traversing them. However, the BVH currently seems to be faster than the grid, more suitable for complex scenes, and somewhat more robust for secondary rays [27]. We have therefore chosen to use a BVH (with 8×8 rays per packet), but most of our ideas generalize to a grid as well.

Compared to a kd-tree, a BVH offers several advantages that are particularly interesting for a Cell-like architecture: a BVH will have fewer memory accesses and a higher compute-to-memory access ratio, because it has fewer nodes than a kd-tree, and more arithmetic to be done per node. In addition, a BVH references each triangle exactly once, which—if the triangles are stored in the correct order—allows for directly referencing the triangles without an additional indirection through a triangle ID list per leaf.

4 Cell-specific Traversal and Intersection

As mentioned in Section 3.2, our system closely follows the traversal proposed in [27], and the traversal algorithm and triangle intersection are exactly the same. Nevertheless, the Cell is not like the CPUs that the original traversal and intersection framework was designed for. Therefore, special optimizations have to be done to efficiently map these routines to an SPE.

4.1 BVH Traversal

Branch mispredictions on the Cell are costly. Unfortunately, the BVH traversal proposed in [27] has two conditionals in its inner loop, both of which have a 40-45% chance of being taken: to reduce ray-box tests, one does not test each ray against every node, but first performs two tests that can often decide the traversal case without having to look at all the individual rays. First, one tests the first ray that hit the parent node, and immediately descends if it hits; if not, the node is tested against the packet's bounding frustum, leading to an immediate exit if the frustum misses the box.

These two tests cover around 80-90% of the traversal cases, making an efficient implementation mandatory. As a serial execution of the two tests introduces dependency chains and therefore pipeline stalls, we perform the two tests in parallel, while postponing the branches as far as possible. Moreover, the branches are arranged in such a way, that mispredicted branches occur only for the third traversal case. The parallel computation completely avoids dependency stalls and increases the double instruction dispatch rate to 35%, yielding total costs of 51 cycles (without a misprediction-stall), which is an 10-20% performance improvement, compared to a serial test execution. The average cycles-per-instruction (CPI) ratio for the code is 0.65, where 0.5 is the optimum.

4.2 Triangle Test

For the packet-triangle test, we use the algorithm proposed in [26, 3]: this test is particular suited for SIMD processing, and in addition, stores all data required for the triangle test in a single memory location. As proposed in [27], we also use a SIMD frustum culling step to detect if the frustum completely misses a triangle.

As with the BVH traversal, the triangle test was originally designed for a x86 CPU, and the Cells in-order execution model requires some changes to remain efficient. The triangle test consists of four individual tests: one first tests the distance to the triangle's embedding plane, and then computes and tests the three barycentric coordinates of the point where the ray pierces the plane. On a x86 CPU, the best performance is achieved if each of these tests is immediately followed by a branch that skips the remaining computations if it failed. On the Cell, these branches cause dependency chains and frequent branch misprediction stalls, which cause the same code to run quite inefficiently. To avoid these, we remove all branches, always performing eight SIMD tests (i.e., 32 rays) in parallel, and updating the results branch-free via conditional moves.

With these modifications, we can intersect a packet of 64 rays in only 520 cycles, or 8.125 cycles per ray on average. In particular the Cell's large number of registers is quite helpful: while on an x86 CPU even a single test can lead to register spilling, the SPE's 128 registers allow for unrolling the intersection test eight times, which yields a double instruction dispatch of 37.5% and a CPI ratio of 0.71. Moreover, all required triangle data can be held in registers, without having to re-load—and re-shuffle—the triangle data for every new batch of rays in the packet.

5 Explicit Caching of Scene Data

The downside of our chosen programming model is that each SPE now requires access to all scene data, which does not fit into local store. As main memory accesses can only be performed by DMA transfers, we emulate caching by creating small self-maintained memory caches within the SPE's local store.

The lack of a hardware-supported cache means that all cache logic has to be performed (using serial code) in software, which is costly. In addition, cache misses result in high-latency DMA transfers, and are quite costly. Even cache hits require a short instruction sequence to obtain the data, increasing access latency of cached data. Fortunately, caching of scene data in a ray tracer has shown to yield high cache hit rates [23, 24, 32] and cache accesses can be additionally amortized over an entire packet of 8×8 rays.

5.1 Types of Caches

Instead of a unified memory cache, we follow the approach recently used for designing ray tracing hardware [23, 32], and use specialized caches for each kind of scene data. Having specialized caches allows for fine-tuning each individual cache's organization and granularity. Due to the DMA transfer granularity, all cache granularities must be powers of two. For the kd-tree based hardware architectures, three types of caches are required: node caches, triangle caches, and item list caches. A BVH references each triangle exactly once, and item lists can be completely abandoned.

BVH Node Cache. Each BVH node stores minimum and maximum box extent, a pointer to the first child (for inner nodes) or first triangle (for leaves), and some additional bits. These can be stored within a 32-byte data structure, which is also a power of two. In addition, we know that a BVH—in contrast to a kd-tree—*always* has to test both children of a node. We therefore do not cache individual BVH nodes, but instead use a 64-byte granularity and cache pairs of BVH node siblings. Compared to caching individual BVH nodes, this yields a roughly 10% higher cache hit rate.

Triangle Cache. As mentioned above, we use the triangle test proposed in [26], which uses a precomputed record of 12 words (48 bytes) for the triangle test. This data is fully sufficient for the intersection test, so no additional caches for triangle connectivity or vertex positions are required, which greatly simplifies the cache design. As 48 is not a power of two, we chose the triangle cache's cache line size to be 64 bytes. The excess 16 bytes are then used to store indices to the three vertices respectively normals and an index to global shader list. These are not required for the intersection test, but are required when shading the intersection points.

Cache hit rates (in %)	ERW6 (804 tris)		Conference (280k tris)		VW Beetle (680k tris)	
	4-way	DM	4-way	DM	4-way	DM
BC (128)	99.8	99.4	95.7	91.2	87.6	84.4
BC (256)	99.9	99.7	98.0	94.4	91.6	88.3
BC (512)	99.9	99.9	98.5	96.9	93.5	91.6
BC (1024)	99.9	99.9	98.7	97.8	94.1	93.2
TC (128)	98.1	96.8	71.0	61.5	45.1	39.8
TC (256)	98.4	97.4	80.0	74.8	50.9	45.8
TC (512)	98.7	98.4	86.7	82.7	55.1	50.4
TC (1024)	98.7	98.7	89.2	86.5	57.2	53.9

Table 1: Cache hit rates for 4-way associative (4-way) vs direct mapped (DM) caches for BVH nodes cache (BC) and triangle cache (TC). Measured with casting primary rays at a resolution of 1024^2 pixels, 8×8 rays per packet, and a default setting of 256 BC entries (17KB), and 256 TC entries (17KB).

5.2 Cache Organization and Efficiency of Caching

Ray traversal offers a high degree of spatial coherence, and even a simple direct mapped cache offers high cache hit rates (see Table 1). A four-way associative cache (with a least-recently-used replacement policy) provides an additional 1-5% higher cache hit rate but requires significantly more complex logic, which increases cache access latency. As a cache access has to be performed in every traversal step, better performance can be achieved with a direct mapped cache, even though it has a somewhat lower hit rate. Though we use a BVH instead of a kd-tree, our cache statistics in Table 1 show nearly identical results to those reported in [23, 32], showing similarly good cache hit rates.

In order to reduce instruction dependency chains, we apply speculative execution: In parallel to the cache hit test, the data is speculatively loaded from the cache and reformatted for further processing. The potential branch to the miss handler is slightly postponed, which allows for hiding cache access latency by interleaving the instruction sequence with surrounding code. As the cache hit rate is typically very high, the increased number of instructions executed in case of a cache miss does not have a significant impact. Due to the high cache hit rates, we use branch hints to optimize all cache access branches for hits. Thus, the hit logic is cheap, and a costly branch miss occurs only in the case of a cache miss.

5.3 Cache Sizes

In addition to the caches, the SPE's local store must also accommodate program code, ray packet data (rays and intersection points), and some auxiliary buffers. Since local store is scarce, the cache sizes must be chosen carefully. Table 1 shows that for our three test scenes a BVH node cache of 256 entries is a good compromise between cache hit rate ($> 88\%$) and memory consumption (17KB); doubling the cache size increases hit rates by a mere 3%, but doubles memory consumption. For the triangle cache, the situation is more complicated. Since triangle intersections are performed where the rays are least coherent (at the leaves), the triangle cache has a much lower hit rate, hence a large cache is beneficial. Nevertheless, even where the hit rates are very low—down to 40-55% for

the Beetle scene—even a much larger cache cannot significantly improve the hit rate: for finely tessellated geometry, triangles are smaller than the spatial extent spawned by the 8×8 rays in a packet, and will therefore often be intersected by a single packet only. Even though the triangle cache's hit rates of only around 50% look devastating, triangle accesses are rare compared to BVH traversal steps, so the total impact of these misses stays tolerable. Because of the costly DMA transfers, a cache hit rate of 50% still ensures a higher performance than using no cache at all.

5.4 Traversal Performance including Caching

With cached access to scene data, we can now evaluate the per-SPE performance of our traversal and intersection code on three example scenes with different geometric complexity (see Section 9.1 for a detailed description). Table 2 gives performance data per SPE, casting only primary rays (no shading operations are applied). DMA transfers invoked by cache misses are performed as blocking operations, making cache misses quite costly. For the rather simple ERW6 scene—which also features very high hit rates—a single SPE (clocked at 2.4GHz) achieves 30 million rays per second (MRays/sec), the more complex conference and VW beetle scenes still achieve 6.7 MRays/sec and 5.3 MRays/sec, respectively.

Scene	ERW6	Conference	VW Beetle
#Traversals Per Packet	18.73	55.33	43.90
#Triangle Isecs/Packet	1.47	5.94	7.21
Traversal Early Exits	44%	40%	34%
Traversal Early Hits	52%	52%	48%
Performance (MRays/sec)	30.08	6.7	5.3

Table 2: Performance per 2.4 GHz SPE, in frames per second for casting primary rays (no shading) at a resolution of 1024×1024 pixels, with 8×8 rays per packet, 256 BVH cache entries (17KB) and 256 triangle cache entries (17KB). Triangle intersections means triangle intersections after SIMD frustum culling. Even though only a small amount of local store is reserved for caches (<35KB), and all DMA transfers are performed as blocking operations, a single SPE achieves a performance of 5-30 million rays per second.

6 Software-Hyperthreading

Though a set of small self-maintained caches within the local store allows for efficiently caching a large fraction of the scene data, having only comparatively tiny caches of 256 entries (for more than half a million triangles), cache hit rates in particular for the triangle cache drop quickly with increasing geometric complexity. Being a streaming processors, the Cell is optimized for high-bandwidth transfers of large data streams, not for low-latency random memory accesses. All memory accesses are performed via DMA requests, which have a latency of several hundred SPE cycles.

The discrepancy between bandwidth and memory latency is not a phenomenon unique to the Cell processor, but exists similarly for every one of today's CPU architectures. One of the most powerful concepts to counter this problem is hyperthreading (also known as simultaneous multi-threading): the CPU works exclusively on one thread as long as possible, but as soon as this thread invokes a cache miss, it is suspended, the data is fetched asynchronously, and another thread is being worked on in the meantime. If thread switching is fast enough, and if enough threads are available, hyperthreading can lead to a significant reduction of pipeline stalls and can therefore lead to higher resource utilization.

Though hyperthreading is most commonly associated with CPUs, it is also used in other contexts. For example, the RPU [32] architecture makes heavy use of hyperthreading, and uses 32 simultaneous threads per RPU core to hide memory latencies. Similarly, the same concept has been used in Wald et al. [29], albeit one level higher up in the memory hierarchy: instead of switching on a memory access, the system in [29] switched to a different packet if a network access was required. Other systems use similar concepts (e.g., [19]).

Algorithm 1 Pseudo-code for BVH traversal with software-hyperthreading. Once a cache miss occurs, the current context is saved, an asynchronous data transfer is invoked, and the traversal continues by restoring the next (not yet terminated) packet.

```
packetIndex = 0
goto startBVHTraversal
processNextPacket:
repeat
  packetIndex = (packetIndex+1) % NUM_VHT_PACKETS
until !terminated[packetIndex]
RestoreContext(packetIndex)
startBVHTraversal:
while true do
  if stackIndex == 0 then
    break
  end if
  while true do
    index = stack[–stackIndex]
    if InsideLocalStoreBVHCache(index) == false then
      SaveContext(packetIndex)
      InitiateDMATransfer(index)
      goto processNextPacket
    end if
    box = GetBoxFromLocalStoreBVHCache(index)
    if EarlyHitTest(box) == false then
      if RayBeamMissesBox(box) == true ——
      AllRayPacketsMissBox(box) == true then
        goto startBVHTraversal
      end if
    end if
    if IsLeaf(box) then
      break
    else
      stack[stackIndex++] = GetBackChildIndex(box)
      index = GetFrontChildIndex(box)
    end if
  end while
  PerformRayTriangleIntersectionTests(box)
end while
```

Though the Cells PowerPC-PPE does support hyperthreading, the SPEs do not. Still, similar to [29] we can emulate the concept in software. Having no hardware support for the context switch, a complete SPE context switch which would include saving all registers and the complete local store to memory, would be prohibitively expensive. Therefore, we define a lightweight thread as single 8x8 ray packet, and traverse multiple of them simultaneously. Thus, only a small data set has to be saved and restored.

DMA transfers can be declared as non-blocking (i.e., asynchronous) and their state can be requested any time. Each time a cache miss occurs, an asynchronous DMA transfer is invoked and the traversal continues execution with the next packet; once the original packet is resumed, its data will usually be available.

6.1 Implementation

In addition to the ray and intersection data, each packet also requires its own stack. Due to scarcity of local store, only a limited number of packets can be kept at the same time. In our current implementation, four packets are being used simultaneously.

In order to suspend and to resume ray packets, all packet-specific data—the "ray packet context"—has to be saved and restored. In our implementation, the packet context comprises a pointer to the corresponding ray packet, a stack pointer, a DMA transfer state, etc. Pointers can be represented as 32bit integers, and pointers for the four contexts can be stored within a single integer vector, which allows for quick insertion and extraction of data.

6.2 Results

As can be seen in Algorithm 1, software-hyperthreading is not trivial, and the context saves and restores carry some significant cost as well. Still, this cost is lower than the several hundred cycles that would be incurred by waiting for the memory request to complete. Overall, software-hyperthreading (SHT) gives a noticeable speedup, as can be seen in Table 3, which compares the performance of a caching-only implementation (see Section 5.4) to the performance achieved when applying SHT to both BVH and triangle cache. As expected, SHT cannot give a noticeable benefit for small scenes in which only few cache misses occur anyway. For larger scenes, however, where cache misses become significant, SHT can achieve a 33% improvement in performance.

Scene	ERW6	Conference	VW Beetle
Without SHT	30.08	6.7	5.3
With SHT	30.1	7.8	7.0
Speedup	0.01 %	15.7 %	33.5 %

Table 3: Impact of software-hyperthreading (SHT) on per-SPE performance. Performance in frames per second (1024×1024 pixels, 256 BVH node cache and and triangle cache entries each, no shading) on a 2.4 GHz SPE. For larger scenes, SHT becomes increasingly important.

7 Shading

Once being able to trace rays, we have to shade the resulting intersection points. Ideally, the Cell would be used as a ray tracing processor only, with shading being done on a GPU. In a Playstation 3, for example, GPU and Cell have a high-bandwidth connection, and sending rays back and forth would be feasible. In that setup, the GPU could do what it's best at—shading—and the Cell would only trace rays. Since we currently do not have a Playstation 3, yet, we have to temporarily realize the shading on the Cell.

In the following, we define a set of 8×8 intersection points as an *i-set*. Each intersection point within an i-set comprises the triangle index, the hit point in world space, the shading normal (interpolated from the three vertex normals), the reference to a surface shader etc. For the SPE's SIMD architecture, shading is most efficient if multiple intersection points are shaded in parallel. Unfortunately, neighboring rays may have hit different geometry, requiring different data to be shaded. Since the smallest SIMD-size is 4, we group these 8×8 intersection points into 16 intersection packets of 4 rays each, and work on each of these in a SIMD manner.

Compared to ray packet traversal, parallel shading has a much more complex control flow, and a significantly more complex data access pattern. In particular, while traversal always intersects all rays with the *same* node or triangle, shading each ray may require different shading data, which may have to be fetched from completely different memory locations (material data, vertex positions, vertex normals,...). Though in principle these accesses could be completely random, in practice there is at least some degree of coherence. For instance, neighboring intersection points typically have the same shader (even if they hit different triangles), and neighboring rays even frequently hit the same triangle. Since our SIMD-width is 4—not 64—we always shade four rays at the same time, and store for each intersection packet a flag whether the four intersection points refer to the same triangle. This allows for a more efficient implementation because cache accesses to the scene data can be amortized over the whole intersection packet.

Smooth shading typically requires a surface normal that is interpolated by the three vertex normals, so an additional cache for vertex data is maintained while filling in the i-set. All vertex data—normal, position, and texture coordinates—is stored within a 64 bytes element, allowing to cache all vertex data in one aligned cache record. In addition to vertex data, we also maintain a cache for material data (diffuse and specular color,...).

The actual shading process is split into several steps. First, we check (by testing the triangle flag) whether rays in the packet have hit the same triangle, using the information to efficiently gather geometric data, in particular, the three vertex normals: the data is loaded once, and then (possibly) replicated across the intersection packet. As can be seen from Table 4, for the 4-ray packets the probability of having hit the same triangle is actually rather high.

The second step uses a multi-pass approach for the shading of an i-set: All *different* surface shaders, which are referenced within the i-set, are sequentially executed. Each of these shading passes works on the full i-set, performing all shading computations for all 64 intersection points, while using bit masks for invalidation of non-related intersection points. In order to speed up the sequential scanning for different surface shaders, each surface shaders invalidates its shader reference in the i-set after execution, ensuring that the corresponding surface shader is not executed again. Table 4 shows that for the test scenes only 1-2 shading passes per i-set are required. After accessing the material cache for a shading pass, no further cache accesses have to be performed, and the intersection points can be efficiently shaded in parallel using SIMD instructions. Note that the current implementation does not support software-based hyperthreading for the geometry or material caches.

For secondary rays, we follow the same approach as Boulos et al. [5]: to generate coherent secondary packets, each 8×8 primary packet generates one reflection packet (of up to 8×8 rays), multiple shadow packets (one per light source), etc. In order to simplify matters, our current implementation uses only a diffuse shading model with shadows, but without reflection or refraction rays.

Scene	ERW6	Conference	VW Beetle
Same tri prob. (in %)	97.18	88.74	78.49
Passes per 8×8	1.005	1.23	1.07
Vertex cache hit (in %)	99.54	96.11	89.35
Material cache hit (in %)	99.99	98.04	99.62

Table 4: Probability of an intersection packet (four rays) sharing the same triangle, and the cache hit rates for the vertex and material cache (direct mapped, 1–2 shading passes). Both the vertex cache and shader cache have 64 entries; all scenes are rendered at 1024^2 (only primary rays).

8 Parallelization across multiple SPEs

So far, we have only considered how to make ray tracing fast on a single SPE. However, each Cell has 8 SPEs, and our dual processor system even has 16 of them. Since we use a homogeneous programming model, and therefore have no SPE-to-SPE communication at all, from a programmers perspective it makes no difference where the SPEs are physically located.

Keeping all 16 SPE utilized requires efficient load balancing. We follow the standard approach of defining the SPE working tasks by subdividing the image plane into a set of image tiles. From this shared task queue, each SPE dynamically fetches a new tile, and renders it. As accesses to this task queue must be synchronized, we employ the Cell's atomic lookup and update capabilities: an integer variable specifying the ID of the next tile to be rendered is allocated in system memory. This variable is visible among all SPEs, and each time an SPE queries the value of the variable, it performs an atomic fetch-and-increment. This atomic update mechanism allows the SPEs to work fully independently from both other SPEs and PPE, requiring no communication among those units. The only explicit synchronization is at the end of each frame, where the PPE waits to receive an 'end frame' signal from each SPE.
Figure 2 shows the efficiency of the dynamic load balancing for our three test scenes, using the same settings as in the previous sections.

For the image tile size, we use 64×64 pixels, resulting in 256 image tiles per frame. Even though the image tiles are only distributed across a single frame (which implies synchronization at the frame end), the approach provides almost linear scalability (without frame buffer transfer and shading) using up to sixteen SPEs.

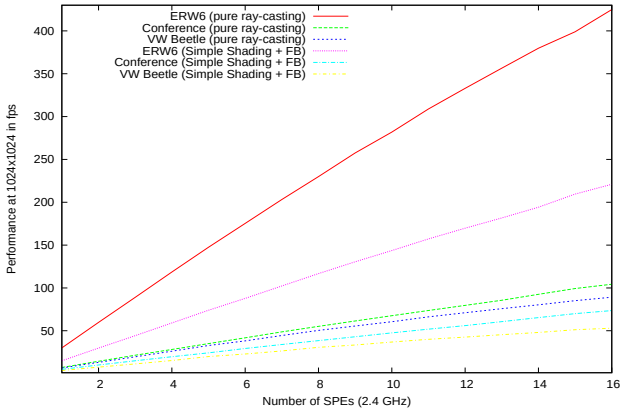

Figure 2: Scalability across several SPEs using dynamic load balancing based on image tiles. The atomic lookup and update capabilities of the Cell makes a fast and efficient implementation possible which is able to provide an almost linear scalability with up to sixteen 2.4 GHz SPEs. All tests were run with 256 BVH box cache entries and 256 triangle data cache entries.

9 Summary and Discussion

In this paper, we have shown how ray tracing can be efficiently mapped to the Cell architecture. Using a SIMD-enabled BVH traversal and specially optimized routines, we achieve a per-SPE performance of several million rays per second. Access to memory is handled via explicitly caching scene data, and software-hyperthreading is used to bridge cache miss latencies. In addition, a load-balanced parallelization scheme achieves nearly linear scalability across multiple SPEs, thereby using all of the Cell's computational resources.

9.1 Comparison to Commodity CPU-based Approaches

In order to evaluate the efficiency of our approach, we have to compare it to alternative approaches. On the Cell processor, no alternative approaches are available, yet[2] and today's fastest published ray tracing results have all been realized on commodity CPUs (using either Pentium-IV CPUs [21, 28], or Opteron CPUs [27]).

To compare against these ray tracers, we have taken some of the scenes also used in [27] and [28], and have measured their performance in our implementation. Table 5 reports the measured we achieve on a single 2.4GHz SPE, as well as on a single and dual 2.4 GHz Cell processor evaluation system with 8 respectively 16 SPEs. As a baseline for comparisons, we have included a x86-based implementation of the algorithm proposed by Wald et al. [27]; for a fairer comparison, we do not use Wald et al.'s code, but a reimplementation that performs exactly the same intersection, traversal, and—in particular—shading computations as on the Cell, but with Opteron-optimized code (thus achieving roughly the same performance as Wald et al.'s system). As we can expect our system's performance to increase roughly linearly in clock rate, we have also extrapolated the performance that would be achievable on a 3.2GHz Cell with 7 SPEs as used in a Playstation 3.

[2]Cell-based ray tracing has been demonstrated by Minor et al. [14], but only for height-field ray casting, not for general 3D ray tracing.

Scene	ERW6		Conference		VW Beetle	
ray casting, no shading						
2.4GHz x86	28.1		8.7		7.7	
2.4GHz SPE	30.1	(+7%)	7.8	(-12%)	7.0	(-10%)
Single-Cell	231.4	(8.2x)	57.2	(6.5x)	51.2	(6.6x)
Dual-Cell	430.1	(15.3x)	108.9	(12.5x)	91.4	(11.8x)
PS3-Cell	270.0	(9.6x)	66.7	(7.6x)	59.7	(7.7x)
ray casting, simple shading						
2.4GHz x86	15.3		6.7		6.6	
2.4GHz SPE	14.9	(-3%)	5.1	(-23%)	3.5	(-47%)
Single-Cell	116.3	(7.6x)	38.7	(5.7x)	27.1	(4.1x)
Dual-Cell	222.4	(14.5x)	73.7	(11x)	47.1	(7.1x)
PS3-Cell	135.6	(8.9x)	45.2	(6.7x)	31.6	(4.8x)
ray casting, shading&shadows						
2.4GHz x86	7.2		3.0		2.5	
2.4GHz SPE	7.4	(+3%)	2.6	(-13%)	1.9	(-24%)
Single-Cell	58.1	(8x)	20	(6.6x)	16.2	(6.4x)
Dual-Cell	110.9	(15.4x)	37.3	(12.4x)	30.6	(12.2x)
PS3-Cell	67.8	(9.4x)	23.2	(7.7x)	18.9	(7.5x)

Table 5: Performance in frames/sec on a 2.4 GHz SPE, a single respectively dual 2.4 GHz Cell processor system, and a 2.4 GHz x86 AMD Opteron CPU using pure ray casting, shading, and shading with shadows (at 1024^2 pixels). Opteron data and 2.4GHz-Cell data are measured, data for the 7-SPE 3.2 GHz processor (as used in the Playstation 3) has been extrapolated from that data. For pure ray casting, our implementation on a single 2.4 GHz SPE is almost roughly on par with a similarly clocked Opteron CPU. In addition, a Cell has 7–8 such SPEs, and can be clocked at a higher rate.

As shown in Table 5, our Cell-based implementation is quite efficient: on a single 2.4 GHz SPE, our implementation achieves a performance that is roughly on par with that achieved by one of the fastest known ray tracing implementations on a full-fledged Opteron CPU. As our system scales well over the available SPEs, our dual-Cell evaluation system is then 7-15 times faster than the AMD Opteron-based system, and the estimated Playstation 3 performance is 5-9 times that of an Opteron CPU.

Applying the profiling features of the Cell Simulator [11] allows for obtaining a detailed dependency stall analysis on the SPEs. As the current simulator does not provide a cycle accurate profiling of DMA transfers, we excluded cache accesses (and the related DMA transfers) from the analysis. For pure ray casting, roughly 25% of all cycles are taken by stalls: 11% for mispredicted branches, 10% by dependency stalls and 4% by branch-hit related stalls. The CPI ratio for complete BVH traversal is 0.99.

9.2 Shading and Secondary Rays

As can also be seen from Table 5, the Cell-based shading does not work as good as the traversal and intersection: in particular for more complex scenes in which rays hit different triangles, even simple shading becomes costly. For example, while for all test scenes pure ray casting on a single SPE is roughly as fast as on a AMD Opteron core, the Opteron is up to twice as fast once shading gets turned on. This, however, is not surprising, as we have mostly concentrated on efficient ray traversal and intersection so far. In particular, profiling the shading code (using the simulator) shows that 43% of the cycles required for shading are wasted in stalls: 21.5% for dependency and 21.5% for mispredicted branches. This is mostly

caused by inefficient instruction scheduling, so we expect that future compiler versions in combination with manual optimizations will provide a significant performance increase for the shading part. Secondary rays, on the other hand, do *not* further widen the performance gap between x86 and the Cell processor, as they again benefit from the optimized ray traversal kernel. In addition, shadow rays could be further accelerated using early shadow ray termination, which hasn't been applied, yet.

For highly recursive shading and realistic lighting effects it is still not clear how to efficiently map them to a packet-based shading framework. First work has already been done [5], and we believe most of this to be directly applicable. However, the limitation of the Cell processor, e.g. limited local store size, makes the realization of complex shading even more challenging.

As a Cell will be used as the CPU for the PlayStation 3, a direct high-bandwidth connection to the PS3's GPU will exist. If a sufficiently high ray traversal performance could be achieved and the shading could entirely be done on the GPU, ray traced effects could finally be delivered to commodity game consoles.

9.3 Dynamic Scenes

Even though dynamic scenes have not been considered here, we believe that the Cell processor is perfectly suited to handle the required algorithms. As shown in [27], handling dynamics requires a fast update of BVH nodes and triangles, which can be efficiently mapped to a streaming work model: An SPE loads a chunk of vertices or BVH nodes, works on the data and stores the results back to memory. Such a streaming approach is perfectly suited, as the Cell's EIB ensures a very high memory bandwidth. Additionally, the memory latency can be efficiently hidden by using double or triple buffering techniques.

9.4 Caching, Software-Hyperthreading, and Bandwidth

As shown in the previous sections, caching works well for BVH nodes, but the cache hit rates for triangles quickly drop with an increasing scene complexity. Under the assumption that enough memory bandwidth can be reserved, one could abandon the triangle cache completely. However, for high frame rates the bandwidth could possibly limit the total performance.

Figure 6 shows that for the complex VW Beetle scene only 12 MB of bandwidth to memory is required. In particular, the largest part of the bandwidth is taken by loading triangle data (4.7 MB). Note that for writing the final color as 32bit RGB values to the frame buffer, 4,096 KB per frame of additional bandwidth is required. Even though a memory bandwidth of 12 MB per frame seems to be low, one should keep in mind that the loading is not performed in large chunks of data, but with small granularities of 16, 32 or 64 bytes. Memory latency has therefore a much higher impact than memory bandwidth. For this kind of latency-bounded memory access, software hyperthreading is a useful approach.

Scene	ERW6	Conference	VW Beetle
BVH node data	24	1,113	3,724
Triangle accel data	43	2,797	4,766
Vertex data	130	1,278	4,303
Shader data	0.03	87	1.53
Σ	202	5,275	12,794

Table 6: Required bandwidth to system memory for loading different types of scene data (in KB per frame). All scenes are rendered at 1024^2 pixels using simple shading, and with 256 entries for both BVH cache and triangle cache. Due to our caching framework, even the complex VW Beetle requires a mere 12 MB of memory bandwidth per frame.

9.5 Architectural Improvements

Even though the Cell—and, in particular, the SPEs—have a powerful architecture and instruction set, even small extensions to either of them could further improve its efficiency for ray tracing.

One of the bottlenecks of the current generation is branching. Only a single branch hint can be specified at a time, and this must be placed at a certain distance before the branch. This results in an increased number of branch mispredictions for branch-intense code. Specifying branch hints for multiple branches in advance could significantly reduce the misprediction rate.

The shading part requires many data gather operations, e.g. loading four word elements from four different locations in the local store, where the four addresses are held within a single register. As each of these word elements does not need to be aligned on a sixteen byte boundary, a long and costly instruction sequence (scalar loading) is required to load and arrange the data. Therefore, an extended load instruction would be very helpful.

10 Conclusion

We have shown how to efficiently map the ray tracing algorithm to the Cell processor, with the result that a single SPE achieves roughly the same traversal performance as the fastest known x86-based systems, and using all of a Cell's SPEs yields nearly an order of magnitude higher traversal performance than on an Opteron.

The remaining bottleneck is shading, which requires many cache accesses, costly data gather operations and a complex control flow, making the Cell architecture less efficient than a commodity x86 core. Therefore, future modifications should directly concentrate on a maybe simplified but efficient shading framework.

As the SPEs are exclusively designed for high clock rates, we can expect future versions of the Cell processor to have a higher clock rate and an increased number of SPEs. Even the current generation of SPEs has been reported to run stable at 5.2 GHz[4], so we can expect a great performance boost from future generations.

Acknowledgments

We would like to thank Mercury Computer Systems for providing the Cell evaluation system and all people that have contributed to this paper, in particular Philipp Slusallek, Kevin O'Brien and Barry Minor for their helpful support.

References

[1] Advanced Micro Devices. AMD Opteron Processor Model 8 Data Sheet. http://www.amd.com/us-en/Processors, 2003.

[2] John Amanatides and Andrew Woo. A Fast Voxel Traversal Algorithm for Ray Tracing. In *Proceedings of Eurographics*, pages 3–10. Eurographics Association, 1987.

[3] Carsten Benthin. *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Saarland University, Saarbrücken, Germany, 2006.

[4] B.Flachs, S.Asano, S.Dhong, P.Hofstee, G.Gervais, R.Kim, T.Le, P.Liu, J.Leenstra, J.Liberty, B.Mchael, S.Mueller, O. Takahashi, Y. Watanabe, A.Hatakeyama, H. Oh, and N.Yano. A Streaming Processing Unit for a Cell Processor, 2005. Available from http://www-306.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine.

[5] Solomon Boulos, Dave Edwards, J Dylan Lacewell, Joe Kniss, Jan Kautz, Peter Shirley, and Ingo Wald. Interactive Distribution Ray Tracing. *Technical Report, SCI Institute, University of Utah, No UUSCI-2006-022*, 2006.

[6] Alan Chalmers, Timothy Davis, and Erik Reinhard, editors. *Practical Parallel Rendering*. A K Peters, 2002. ISBN 1-56881-179-9.

[7] Intel Corp. Intel Pentium III Streaming SIMD Extensions. http://-developer.intel.com/vtune/cbts/simd.htm, 2002.

[8] Intel Corp. Introduction to Hyper-Threading Technology. http://developer.intel.com/technology/hyperthread, 2002.

[9] Intel Corp. Intel Next Generation Micro Architecture. http://www.intel.com/technology/computing/ngma/, 2005.

[10] International Business Machines. IBM Power5. http://www-03.ibm.com/systems/power/, 2005.

[11] International Business Machines. The Cell Project at IBM Research. http://www.research.ibm.com/cell/, 2005.

[12] F. W. Jansen. Data structures for ray tracing. In *Proceedings of the workshop on Data structures for Raster Graphics*, pages 57–73, 1986.

[13] B. Khailany, W.J. Dally, U.J. Kapasi, P. Mattson, J. Namkoong, J.D. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: media processing with streams. *IEEE Micro*, 21(2):35–46, 2001.

[14] B. Minor, G. Fossum, and V. To. TRE : Cell Broadband Optimized Real-Time Ray-Caster. In *Proceedings of GPSx*, 2005.

[15] Gordon E. Moore. http://www.intel.com/technology/silicon/mooreslaw/.

[16] Henry Moreton. 3d graphics hardware: Revolution or evolution, 2005. http://www.graphicshardware.org/previous/www_2005/presentations/moretonpresentationgh05.pdf.

[17] Michael J. Muuss. Towards Real-Time Ray-Tracing of Combinatorial Solid Geometric Models. In *Proceedings of BRL-CAD Symposium*, 1995.

[18] Steven Parker, William Martin, Peter-Pike Sloan, Peter Shirley, Brian Smits, and Charles Hansen. Interactive Ray Tracing. In *Proceedings of Interactive 3D Graphics*, pages 119–126, 1999.

[19] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering Complex Scenes with Memory-Coherent Ray Tracing. *Computer Graphics*, 31(Annual Conference Series):101–108, August 1997.

[20] T.J. Purcell, I. Buck, W.R. Mark, and P. Hanrahan. Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics*, 21(3):703–712, 2002. (Proceedings of ACM SIGGRAPH).

[21] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-Level Ray Tracing Algorithm. *ACM Transaction of Graphics*, 24(3):1176–1185, 2005. (Proceedings of ACM SIGGRAPH).

[22] Steve M. Rubin and Turner Whitted. A three-dimensional representation for fast rendering of complex scenes. *Computer Graphics*, 14(3):110–116, July 1980.

[23] Jörg Schmittler, Ingo Wald, and Philipp Slusallek. SaarCOR – A Hardware Architecture for Ray Tracing. In *Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, pages 27–36, 2002.

[24] Jörg Schmittler, Sven Woop, Daniel Wagner, Wolfgang J. Paul, and Philipp Slusallek. Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip. In *Proceedings of Graphics Hardware*, 2004.

[25] Sun Microsystems. UltraSparc-T1. http://www.sun.com/processors/UltraSPARC-T1/index.xml, 2006.

[26] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004.

[27] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *Technical Report, SCI Institute, University of Utah, No UUSCI-2005-014 (conditionally accepted at ACM Transactions on Graphics)*, 2006.

[28] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G. Parker. Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics*, 25(3):485–493, 2006. (Proceedings of ACM SIGGRAPH 2006).

[29] Ingo Wald, Philipp Slusallek, and Carsten Benthin. Interactive Distributed Ray Tracing of Highly Complex Models. In Steven J. Gortler and Karol Myszkowski, editors, *Rendering Techniques*, Proceedings of the 12th Eurographics Workshop on Rendering Techniques, London, UK, June 25-27, 2001, pages 274–285. Springer, 2001.

[30] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3):153–164, 2001. (Proceedings of Eurographics).

[31] Samuel Williams, John Shalf, Leonid Oliker, Shoal Kamil, Parry Husbands, and Katherine Yelick. The Potenial of the Cell Processor for Scientific Computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20. ACM Press, 2006.

[32] Sven Woop, Jörg Schmittler, and Philipp Slusallek. RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. *Proceedings of ACM SIGGRAPH*, 2005.