Data structures for ray tracing on Cell architecture

Michal Hapala*
Department of Computer Graphics and Interaction
Faculty of Electrical Engineering
Czech Technical University in Prague

Abstract

Today, streaming multiple core architectures are on the rise. This brings an option to use these architectures for ray tracing as these algorithms can heavily benefit from multiple available cores. We show an implementation of two different acceleration structures on the Cell Broadband Engine Architecture, which belongs to the aforementioned group. We show the limitations of Cell and how some of them may be overcome by software caches, branch hinting and branchless code.

Keywords: Ray tracing, Cell, branchless

1 Introduction

Ray tracing is a technique for generating photo-realistic images from 3D scenes composed of objects with defined materials. Every pixel's color is computed by shooting a ray through it and finding the closest intersection between the ray and one of the scene objects. Color is taken from the material of the found object and it is then simple to generate shadows by finding an occluder between the intersection and a light source and also complex light effects by shooting secondary rays, which start in the intersection point and lead outward.

The general idea of ray tracing is a couple of decades old, though only in recent years it started to gain popularity. The reason for it being so looked over in interactive industry was its poor performance in comparison to the main technique used, which is rasterization. It is quite clear that ray tracing is a computationally expensive technique, with its need for shooting millions of rays to find color for pixels in the image. Rasterization with z-buffer, on the other hand, is supported by every major graphics card manufacturer, is easy to work with and has been a standard for years.

However, recent changes in hardware development put ray tracing into prominence. Manufactures realized that pushing circuit frequency to the limit does not yield the best performance anymore and rather moved to multi-core architectures with optimized instructions. This current era spawned the much acclaimed dual and quad core Intel processors for PCs, but also hybrid multimedia units, such as the IBM Cell Broadband Engine Architecture, with one leading CPU and 8 simplified worker processors. Cell's first major commercial application was to be the heart of Sony Playstation 3 gaming console which also boasted an Nvidia graphic card. And graphic cards also experienced a rebirth in recent years as Nvidia followed with the G80 processor in 88xx GeForce series with massively parallel computational elements and CUDA (Compute Unified Device Architecture), API that allows programmer to write GPU oriented general purpose computations code in C.

As ray tracing is "embarrassingly" parallel, i.e. each pixel's color is independent on the others, it heavily profits from this move to parallel architectures. Also, dedicated hardware units are being developed, like the Ray Processing Unit (RPU) from Saarland University in Germany, successor to the SaarCor, an early experimental ray tracing unit.

This paper aims to see how well the standard ray tracing algorithms and data structures can be mapped to IBM Cell, one of the specialized processing units. In particular what problems could emerge and how these problems can be solved. As IBM Cell is one of the few publicly available many-core architectures it is definitely an interesting platform to try.

2 Acceleration structures

Ray tracing is heavily dependent on data structures that divide the scene into smaller parts to speed up the search for intersected objects. They are essentially search trees where instead of intersecting one object after another one can traverse this tree to get quickly to places where intersection with scene objects is most possible.

Acceleration structures can be split into two major groups, based on their splitting algorithm:

- Space partitioning structures – i.e. they divide space, creating a hierarchy of subspaces. Objects are inserted into these subspaces according to their position.
- Object partitioning structures – i.e. they divide objects, creating a hierarchy of volumes that are bounded to the objects that are inside of them.

Typical construction of an acceleration structure starts with a root node that encompasses the whole scene and follows with these recursive steps:

- If terminating conditions are met create a leaf and terminate. These could be reaching a certain depth in the tree or certain number of objects in our node (this is usually set to one)
- Compute split position
- Distribute geometry among left and right child depending on the split position
- Repeat from the start with left and right child

Quality of the splitting method used usually determines the quality of acceleration structure as a whole, as traversal algorithms are typically trivial to

*hapalm1@fel.cvut.cz or rest@mikee.cz

implement and are rarely the source of problems, thus the focus point here is how to build an optimal tree.

The simplest splitting method is the *median* where one will split space/objects at the median of the respective interval. This is definitely the fastest method but will most probably create a sub-optimal structure. In spite of this, Wächter et al. [WK06] proposed quite an effective algorithm based on median splitting.

Currently, the state of the art among splitting methods is the surface area heuristic (SAH), invented in the late 80's, more recently described in e.g. [WH06]. SAH tries to minimize cost computed from surface area of child nodes and number of primitives in these nodes. As SAH computation is quite costly Wald in [W07] showed a faster binning algorithm where the idea is to split an interval into a number of equally large bins and evaluating SAH only on the boundaries of these bins. Shevtsov et al. then expanded on this with a min-max binning algorithm [SSK07] which gives a better approximation than a conventional binning version with boundaries.

Two different structures, KD-tree and Bounding Interval Hierarchy, are used in this paper, where KD-tree is implemented with the aforementioned min-max binning SAH and BIH with the median split described in [WK06]. Wald et al. [WH06] summarized techniques for KD-tree construction and proposed an *O(N log N)* algorithm, which reaches the theoretical lower bound for KD-tree construction. SAH KD-tree is believed to be the best structure for ray tracing of static scenes, though it struggles with dynamic ones as its construction is costly, also due to the need to copy objects in each split step. [SSK07] expanded on these ideas by showing an approximation scheme for computing SAH without the need to sort objects in advance. Our KD-tree traversal algorithm is based on TA_{rec} described in [H01].

Bounding interval hierarchy (BIH) [WK06] is in a sense an analogue to a Bounded KD-tree [WMS06]. BIH stores only two bounding planes, one for each child. This is in contrast to a standard Bounding Volume Hierarchy storing the whole AABB for each node. Left and right child are then reconstructed in traversal by replacing maximum or minimum value with one of the split values.

What is really different from a BKD-tree is the construction method. Wächter et al. proposed a new non-greedy global heuristic that is based on a global bounding box of a scene, not on subsequent node bounding boxes. This approach, in effect, chooses split plane candidates from a regular grid.

3 IBM Cell

Before moving to the implementation specifics we will summarize IBM Cell architecture and its abilities. Cell is a broadband architecture developed jointly by Sony, Toshiba and IBM. It is a microprocessor designed for computationally intensive, mainly multimedia, tasks. Cell consists of one PPE (Power Processor Element), a Power architecture based processor, and eight Synergistic Processor Elements (SPEs), RISC processors with most of its instructions being 128-bit wide SIMD. The PPE is intended as a work distributor with SPEs as worker units.

The SPE is designed as a simple processor for streamlined workload and because of this it can run on high frequencies of up to 4GHz. Its instructions have fixed latency of 2-7 cycles and they are processed in-order with no branch prediction, though it has a branch hint instruction which can be used to help the processor in instruction pre-fetching. SPE cannot access main memory directly and it has no cache. Instead, all SPEs have a 256kB local store memory for both data and code. Also, it has a large register base consisting of 128 128bit SIMD registers.

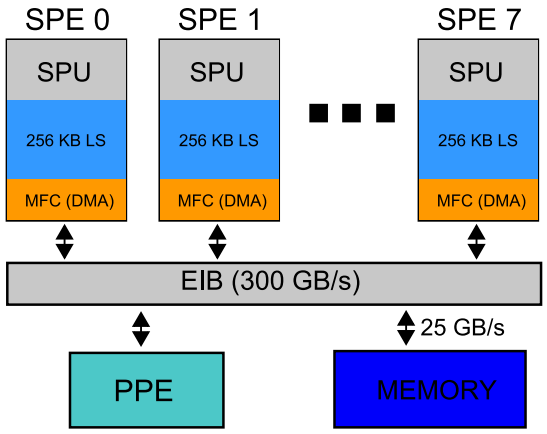


Figure 3-1 IBM Cell Architecture

Element Interconnect Bus (EIB) is used as a connection between all on-chip elements: the PPE, eight SPEs, memory controller and two I/O interfaces. It is capable of transferring 96 bytes per cycle in between its elements.

Cell has a few limitations/features that have to be taken into account, when one wants to run a ray tracing algorithm on this architecture. Those are mainly:

- In-order execution of instructions - In contrast to a contemporary PC processor, Cell SPEs does not support out-of-order execution, relying solely on the quality of a code compiler to pre-process the code. The compiler has to resolve chain dependencies in the code to minimize pipeline stalls.
- Single-instruction-multiple-data instructions – SPE is designed for working on multiple data at once, by the means of SIMD instruction. Therefore, one has to ensure enough independent data to work on at all times, otherwise even the smartest compiler cannot ensure minimum of stalls.
- Memory access – Each SPE has access to three levels of memory: 128 SIMD local registers, 256kB local memory and a main memory access through asynchronous DMA transfer. Main memory access has latency of a few hundred cycles; thus one has to plan his data transfers to larger blocks to avoid stalls.

- Branch hinting - There can be only one branch hint active at once and for the branch hint to be effective one must place it at least 11 cycles and four instruction pairs in front of the branch that is to be hinted.
- Programming model – Eight parallel SPE units gives us choice between homogenous approach, where each SPE runs identical program but on different data, and a heterogeneous approach, where SPEs create a pipeline with data pushed through one SPE after another. Both are feasible options, the latter being possible because of a large 300GB/s bandwidth in between SPEs.

A model how to map ray tracing onto Cell's architecture is given in [BWSF06]. Benthin et al. use homogenous approach with Bounding Volume Hierarchy, running identical ray tracing kernel on all SPEs but on different pixels, using 8x8 packets of rays. They also describe the usage of 256kB local store as an emulated software cache. Cache is organized into a separate BVH Node Cache and a Triangle cache.

4 Implementation

Although Cell is still halfway in between a many-core architecture and a regular CPU, as there's a single "mother" PPE element to run and control the smaller, RISC-like, SPEs, it is generally regarded as a challenging environment for software development. IBM offers a free SDK for anyone to use who has a Linux machine running (we have used the Fedora 8 distribution). Also, if one wants to use a graphic development environment, plugins for Eclipse IDE are available for download. Code can be debugged with a modified version of GDB or it can be profiled on a cycle-precise simulator that is developed separately.

Beyond simple code rewrite we've had to tackle different challenges, mainly the absence of hardware cache and branch prediction on the SPEs. We've done measurements of our first implementation and they've indicated that our traversal algorithm is stalling mostly on branch misses. We've then decided to implement a branchless version. We have also tested all possible cache layouts and an asynchronous cache access to find out settings that will give us a performance gain.

Both of the acceleration structures we've implemented on Cell reacted positively to our improvements as they both gained performance with a move to branchless code and asynchronous cache access.

4.1 PPE

Choices had to be made about which parts of code would be run on PPE and which on the smaller SPEs. It was decided to keep initialization and hierarchy build on the PPE, as there was no parallel code in place. Ray traversal was to be executed on the SPE units in parallel.

Acceleration structure was split into two, one to be held in main memory with build statistics and other excess member variables, and one that is transferred to the SPE with only the necessary pointers (e.g. to node array, primitive array etc.) for the SPE to work with. The PPE side program now runs these phases:

- Initialization – All basic structures are allocated and initialized
- Hierarchy build – BIH or a KD-tree build
- SPE Init – PPE spawns a thread for each SPE. Inside it a premade traversal kernel is loaded onto the SPE and started, which blocks this thread. Main thread then sends a memory pointer to the already built acceleration structure.
- Trace - A global packet counter is initialized which holds pixel coordinates of the last packet traced. PPE spawns a thread for each SPE and until there's a valid packet coordinate waiting, it will:
    1. increment global pixel coordinates accordingly
    2. if this is the last pixel then terminate, otherwise continue to 3
    3. initialize a packet based on these coordinates
    4. send a packet pointer to the SPE
    5. wait for SPE to send back the color of the traced pixels in the packet
    6. save this color into the frame buffer
    7. go back to 1

Main thread uses *pthread_join* to wait until all SPE threads have finished.

There are two ways how to transfer data from the main memory to the SPE. First option is a DMA transfer, to be used for larger amounts of data and it usually takes considerable amount of cycles, or mailboxing, a direct 32bit transfer through a dedicated pipeline. Both have their usage in our code. Mailboxing functions are used to send memory pointers to structures that are then DMAed in by the SPE. As they are able to block execution until a message arrives, they are also used for synchronization on both the SPE and PPE side. Software caches, which are used in the SPE code, also use DMA transfer when a cache miss occurs, though this can be minimized by clever memory setup.

4.2 SPE

SPE binary has to be small with an upper limit of 250kB (size of SPE local store) to cover code and local memory to be used by it. Parts of its inner workings were revealed in the previous chapter, mainly about data exchange in between SPE and PPE through DMA and mailboxing. Every SPE holds in local store pointers to node and primitive arrays, data from these are loaded from main memory on demand and cached as they are needed by the traversal and triangle intersection code.

SPE kernel first receives a hierarchy pointer, loads it to local store and then enters an infinite while loop, in which it waits for a ray packet pointer, DMAs the packet in after receiving it and then traces it. After it is finished the SPE will DMA the results back to the main memory and waits for another packet.

4.3 SPE Software caches

As access to main memory is costly (in the range of 900 cycles) it is advisable to use a cache to store data that are used frequently or load data speculatively from main memory to exploit data locality. SPU provides us with other memory than registers, a 250kb local store which has an access time of 7 to 11 cycles. Luckily, in SDK version 3.0 there is implementation available of a directly mapped or a 4-way associative software cache, with a couple of handy macros to change cache type, cache line width and number of sets. Great benefit of having a software cache is definitely in the option to optimize it for the current code by changing its topology and internal algorithms.

We have used the SDK implementation with success, with only minor changes in branch hinting, as we expected a branch hit to occur most of the time. Hence a branch hint was used in the code to always pre-load instructions for a branch hit, instead of a miss.

We have employed two separate directly mapped software caches as was proposed in [BWSF06], one for nodes and the other for primitives, both the size of 16kB, which was the maximum we were able to achieve due to the 256kB local store limit. Different cache line widths and number of sets were used and cache hit/miss ratios from these tests can be found in section 5, though we were limited by the maximum size the cache can have.

4.4 SPE evaluation and optimization

Our main focus was set on the traversal code as a main unit of execution, which, with the first implementation which was described up to this point, was performing terribly as far as clocks per instruction (CPI) was concerned. This was caused by branch and dependency

```
Perf. Cycle count  2305725
Perf. Instr. Count 1243760 (1162657)
Performance CPI    1.85 (1.98)

Single cycle       805931 ( 35.0%)
Dual cycle         178363 (  7.7%)
Branch miss stalls 566692 ( 24.6%)
Dependency stalls  743987 ( 32.3%)
```

stalls:

Therefore we have decided to optimize the code for branches, i.e. implement a branchless traversal algorithm, as we saw no place where to interleave the code to lessen the dependency hit. Asynchronous access to cache was further employed to improve the branchless code's CPI. Moreover, we have experienced what we believe is a compiler error, which had to be solved with a sort of a hack.

4.5 Branchless code

Idea of transforming branched code to a branchless one is based on running computations in both the taken and not taken part sequentially and then choosing one based on the boolean value that was previously used to branch the code. In SIMD this can be achieved by a "blending" instruction that will combine two SIMD values into one according to another SIMD. SPE instruction set calls this instruction *spu_sel*.

Our branchless traversal algorithm [H07] is based on the idea that instead of using branches when deciding which node is going to be traversed or possibly pushed on the stack, one can push both nodes and with simple moving the stack pointer forward or backward skip those that are not needed.

This can be demonstrated on a branchless version of a KD-tree traversal code:

```
  bool c = t < tmax;
  bool d = t < tmin;

  stack[stackPtr].node = farChild;
  stack[stackPtr].tmin = max(t,tmin);
  stack[stackPtr].tmax = tmax;
  stackPtr += c;

  stack[stackPtr].node = nearChild;
  stack[stackPtr].tmin = tmin;
  stack[stackPtr].tmax = min(t,tmax);
  stackPtr -= d;
  /* stack pop is already included
here */

  currNode   = stack[stackPtr].node;
  tmin       = stack[stackPtr].tmin;
  tmax       = stack[stackPtr].tmax;
```

4.6 Asynchronous cache access

In the first version of our code a synchronous cache access was employed, e.g. when a cache miss occurred code stalled until the necessary DMA transfer finished. Cell SDK's software cache implementation supports a so-called "unsafe" (asynchronous) approach that allows us to only signify that we will need some data in the future, do some computations in the meantime, and then ask for the data. This can be done with *cache_touch* and *cache_wait,* where the former will start DMA transfer when the data is not cached and the latter will wait for this transfer to finish or return cached data immediately when there's no DMA transfer pending.

Asynchronous access to cache proved useful in the branchless code where both branches are computed sequentially, thus both nodes were read from cache every time. With cache touching one can use pointers to structures (synchronous access to cache needs actual structures), only signify that he will need data on those pointers (children of the actual node) sometimes in the future and read them in via DMA in the background.

An error can occur, however, when other cache operations are done in between touch and wait, as we can evict the line we asked for in the first place. It is clear that this can lead to serious artifacts in the image produced, as sometimes cache would give us a different node than what we asked for. A correct approach is to use a cache touch before each wait to make sure that no cache eviction has occurred. The worst that can happen is that we will really have to wait for the data to be transferred via DMA again, hence to utilize this we have to balance cache line size with cache eviction probability.

4.7 While branch hint

A small inconsistency led us to an idea how to optimize hints in a while cycle, where we were unable to persuade the compiler to behave as we wanted it to. What happened is that we've put a branch hint on a while cycle condition and it never actually got into the code on the place we expected. While cycle compiles to a code with two jump instructions and when using while { } we cannot effect in any way where our hint would be placed.

It can be wise to do this decomposition (while to do-while) ourselves and try to define that we want the hint on both branches (this is still only a hint to the compiler itself, it does not mean it will end up in code).

In our code this little trick helped, and we were able to save approximately 3% of branch miss stalls and apparently due to code reformatting another 3% of dependency stalls.

5 Results

In this section we will summarize results measured on both structures that were implemented on a set of testing scenes. These were chosen in a variety of triangle count and scene layouts and some of them are also quite commonly used in scientific papers mentioned throughout this text.

The testing scenes used were:

A10

218652 tris

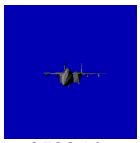
City

68497 tris

Crown

3092 tris

F15



3588 tris

Bunny

70255 tris

Fairie forest

174117 tris

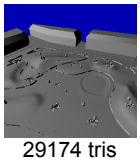
Moon

35520 tris

Sponza atrium

67462 tris

Park



29174 tris

Sibenik cathedral

80479 tris

* A10 scene was modeled by and is a courtesy of Ondřej Karlík (keymaster@keymaster.cz)

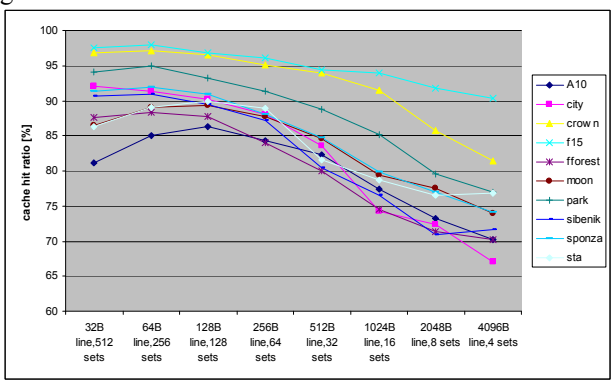
5.1 Cell cache statistics

First section discussing statistics is about caches, as high cache hit rate is principal to our code performance on the SPE. As we had two distinct software caches we've made measurements for both of them with all possible settings in cache line size and number of sets. However we were limited by SPE code size, as software cache is essentially a part of the executable as any other static data.

We were interested in the cache hit ratio for different cache layouts. All tests were also timed to get an idea how the overall code is performing in response to layout changes. Caches were assessed separately and to keep comparable results the one not being measured was always set to constant values of 128B cache line with 128 sets.

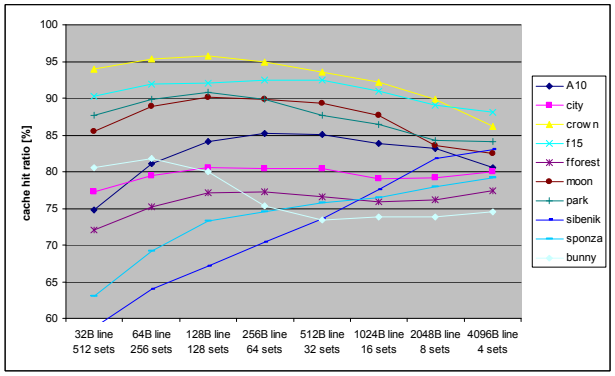
5.2 Node cache

There is a major difference between BIH and KD-tree (KDT) cache hit profiles. KDT usually checks only one of node's children when traversing, so the KDT profile looks as expected; for a bigger cache line more cache evictions will occur, thus lowering hit rate. BIH on the other hand will usually in the end try both of its nodes, putting one on stack. Larger cache lines here befits scenes that traverse more nodes per ray, as they are positioned close to each other in memory which explains why the cache hit rate can sometimes rise even with a greater cache line.

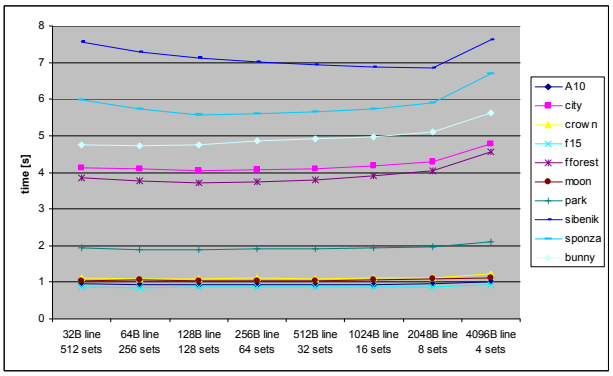


Graph 5-1 KDT branchless cache – node cache hit ratio

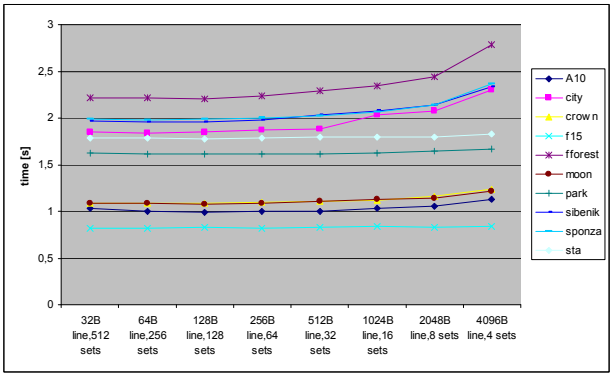
Considering time, KD-tree case is clear. A small rise up to 128B is gained from having few (8 nodes for 128B cache line) sequential nodes prepared in memory. Beyond 128B cache line evictions will start to take their toll and performance deteriorate very quickly. BIH's hit rate is not falling as swiftly, or is even rising, though that is not enough to overcome more costly DMA transfers and beyond 256B cache line size performance will almost always only get worse.



Graph 5-2 BIH branchless cache – node cache hit ratio

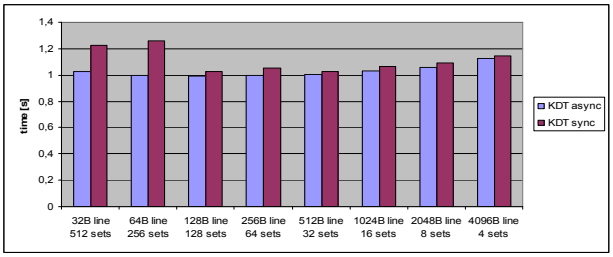


Graph 5-3 BIH branchless cache – rendering time depending on node cache layout

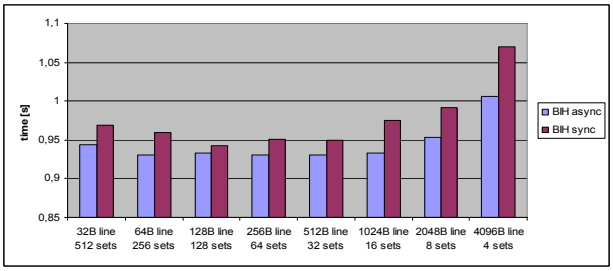


Graph 5-4 KD-tree branchless cache – rendering time depending on node cache layout

Lastly, we will compare a synchronous access to cache against the asynchronous one. This will be shown for both implementations, where even if cache hit ratio is higher for the synchronous access many of the nodes gotten are lost as in our branchless algorithm both nodes are loaded onto stack and then possibly one or both are immediately popped depending on the traversal computation. Thus it is clearly wiser to only "touch" those nodes that are pushed and wait for the DMA transfer to complete only when the node is really needed.



Graph 5-5 A10 scene KDT async/sync rendering time depending on node cache layout

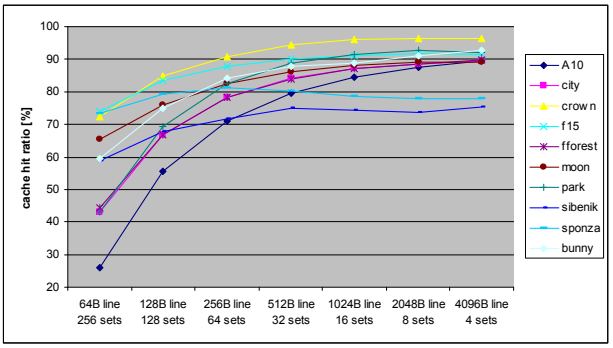


Graph 5-6 A10 scene BIH async/sync rendering time depending on node cache layout

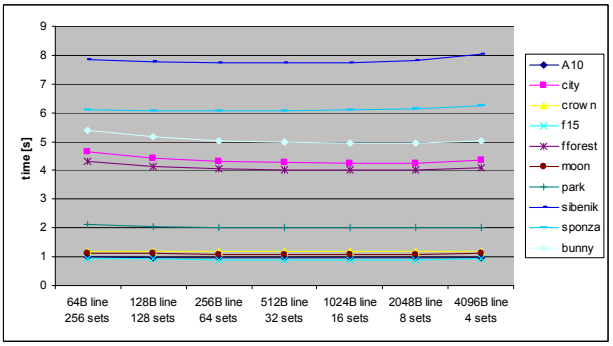
Graph 5-5 and Graph 5-6 show that an async approach is clearly better.

5.3 Triangle cache

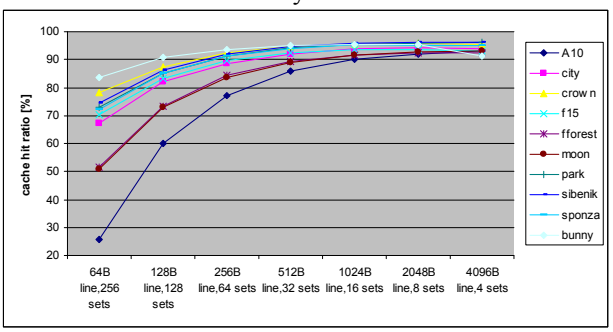
Triangle caches for both KD-tree and BIH show similar behavior. Their hit rate rises with the cache line up to a limit of about 1024B where it either rises very slowly or falls down. As in previous section it is useful to have a look at the measured rendering times as these will give us an idea if the cache line growth is useful overall.



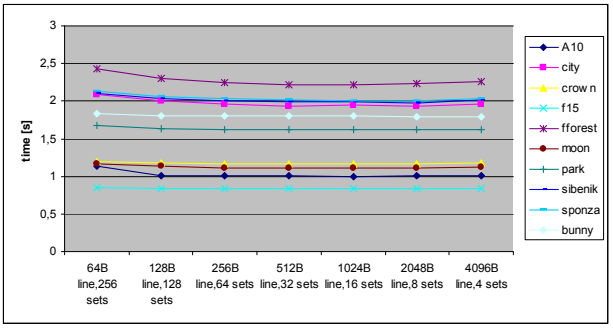
Graph 5-7 BIH triangle cache hit ratio



Graph 5-8 BIH rendering time depending on tri-cache layout



Graph 5-9 KDT triangle cache hit ratio



Graph 5-10 KDT rendering depending on tri-cache layout

As we can observe from Graph 5-8 and Graph 5-10 it is clear that 1024B is indeed the upper limit and beyond that performance either stays the same or degrades.

5.4 Cell timings and instruction statistics

Here we will show time measurements for the fastest cache layouts as was discussed in the previous chapter. Scenes would be measured with 1 to 6 SPE's active to see if our code is scalable and if it is then how much. These would be shown for the fastest code, branchless variant with asynchronous cache access, of both acceleration structures. All renderings were done in 512x512 resolution using primary and shadow rays with one light source with the following cache settings:

- 1024B cache line for triangle cache
- 256B cache line for BIH node cache
- 128B cache line for KD-tree node cache
-

BIH	1	2	3	4	5	6
A10	0,908	0,687	0,686	0,642	0,638	0,624
CITY	3,770	2,267	1,955	1,682	1,517	1,506
CROWN	1,077	0,793	0,777	0,719	0,718	0,723
F15	0,857	0,632	0,621	0,603	0,625	0,605
FFOREST	3,475	2,182	1,962	1,619	1,554	1,476
MOON	1,043	0,776	0,733	0,721	0,722	0,730
PARK	1,808	1,374	1,271	1,158	1,133	1,142
SIBENIK	6,734	3,385	2,548	2,207	2,040	1,892
SPONZA	5,397	2,827	2,342	2,141	1,740	1,688
BUNNY	4,526	2,398	1,766	1,687	1,570	1,504

Table 5-1 BIH Cell performance for 1 to 6 SPEs in seconds

Clearly the performance has a lower bound, as we can observe that it is not much scalable beyond the 2 active SPEs. This can be caused by a sub-optimal PPE code performance, whereas it cannot supply packet data fast enough. Also, as we process only one 2x2 packet at once it is clear that smaller scenes that will compute their packets too fast will have to wait longer for the PPE to get to them and their scalability will deteriorate much faster.

KDT	1	2	3	4	5	6
A10	1,000	0,777	0,735	0,731	0,727	0,743
CITY	1,764	1,372	1,180	1,075	1,146	1,096
CROWN	1,059	0,750	0,697	0,698	0,700	0,706
F15	0,811	0,625	0,591	0,590	0,590	0,614
FFOREST	2,059	1,533	1,397	1,300	1,203	1,195
MOON	1,056	0,804	0,742	0,746	0,744	0,758
PARK	1,578	1,185	0,991	0,940	0,972	0,947
SIBENIK	1,894	1,298	1,297	1,187	1,154	1,186
SPONZA	1,889	1,331	1,256	1,198	1,265	1,206
BUNNY	1,744	0,909	0,817	0,821	0,833	0,818

Table 5-2 KDT Cell performance for 1 to 6 SPEs in seconds

	TOT	INTERN
A10	6,47	1,28
CITY	1,34	1,28
CROWN	5,21	1,29
F15	23,04	1,28
FFOREST	1,63	1,28
MOON	6,71	1,28
PARK	1,93	1,29
SIBENIK	1,29	1,28
SPONZA	1,32	1,28
BUNNY	1,3	1,28

Table 5-3 BIH total versus internal while CPI

	TOT	INTERN
A10	4,12	1,16
CITY	2,12	1,16
CROWN	6,91	1,16
F15	13,37	1,16
FFOREST	3,17	1,16
MOON	4,91	1,16
PARK	2,71	1,16
SIBENIK	2,47	1,16
SPONZA	2,54	1,16
BUNNY	2,05	1,17

Table 5-4 KDT total versus internal while CPI

We have measured total CPI over the whole program for one active SPE and compared it with CPI measured over the traversal internal while cycle (which was our performance target). Tables 5-3 and 5-4 above clearly show that there is indeed something wrong with smaller scenes even without the scalability limitation.

	SINGLE CYCLES	DUAL CYCLES	BR. MISS STALLS	DEPENDENCY STALLS	CHANNEL STALLS
CROWN	7,6 %	2,3 %	4,1 %	4,3 %	81,1 %
F15	3,5 %	1,1 %	1,8 %	1,9 %	91,4 %
MOON	10,4 %	2,9 %	6,4 %	6,8 %	72,8 %

Table 5-5 BIH Small scenes channel stalls

To further test the PPE-SPE communication performance, we've measured channel stalls for all of our smaller scenes to see if the SPE is not overly waiting on DMA transfers and Table 5-5 shows that performance was worsening precisely for this reason. Also the smaller the scene, i.e. the shallower resulting acceleration structure and faster packet traversal, the higher the overall percentage of channel stalls.

There was still a possibility that these would be DMA's run by cache code. This hypothesis was proven incorrect by another measurement, which measured complete traversal code, leaving outside of performance code only the DMAs to PPE (packet retrieval and result save). This was shown on one small and one large scene and Table 5-6 shows that for both the channel stalls are negligible.

	SINGLE CYCLES	DUAL CYCLES	BR. MISS STALLS	DEPENDENCY STALLS	CHANNEL STALLS
F15	49,9 %	15,1 %	10,7 %	21,6 %	0,2 %
SPONZA	51 %	13,2 %	9 %	24,8 %	0 %

Table 5-6 BIH Complete traversal code performance

As our overall implementation is suboptimal we had another choice how to compare the quality of code and that is by comparing CPI. We have measured instruction statistic over the internal traversal while loop, as this we

were trying to gradually improve. Our statistics show that internal cycle statistics are the same (in the range of 1.0% of each other) for every measured scene, thus the statistics here would show only one (moon) scene.

	CPI	SINGLE CYCLES	DUAL CYCLES	BR. MISS STALLS	DEPEND. STALLS
BIH BRANCH SYNC	2,05	33,6 %	7,6 %	27,6 %	30,1 %
BIH BRANCHLESS ASYNC	1,28	54,3 %	11,8 %	5,9 %	27,4 %
KD-TREE BRANCH SYNC	1,87	33,7 %	9,9 %	14,6 %	40,8 %
KD-TREE BRANCHLESS ASYNC	1,16	49 %	18,6 %	6,5 %	23,3 %

Table 5-7 Traversal while cycle instruction statistics for BIH

We can see that we've achieved an improvement in CPI values with our branchless code. This was caused with two factors:

- Branchless code successfully lowered overall cycles lost to branch miss
- Code with more instructions in a row can be interleaved better to lower dependency misses.

6 Summary and Conclusion

This paper focused on evaluating ray tracing acceleration structures implementations on the Cell architecture. From our results we can summarize the following:

- There was a difference in the best node cache layout, which we attribute to a different traversal algorithm, where BIH as an object partitioning structure usually traverses both children, thus needing more local nodes than a KD-tree.
- Software cache in SPE local store is feasible and the option to change its layout as one see fit can be used to gain performance advantage.
- Asynchronous cache access clearly improves performance.
- Our branchless code outperforms our branched code. It has lower CPI and better timings. This is due to a smaller number of branch and dependency misses.
- Branch hints are only suggestions to the compiler, thus one has to watch if his hints actually got into the resulting compiled code. We have gained performance by analyzing branch and hint histories and changing our code accordingly.
- 2x2 ray packets are too small for the code to be scalable, as DMA stalls starts to greatly hinder performance. To overcome it one has to use bigger packets to reduce communication with SPEs.

We've not concerned ourselves with more high-level algorithms as they can be used to improve performance on basically all architectures, and showing this was not the point of this paper. However in future work we will definitely want to look into optimizing these higher levels, as it could lead to a smaller amount of PPE-SPE communication and thus less channel stalls that now hinder the overall performance.

Acknowledgments

We would like to thank Vlastimil Havran for invaluable help and remarks. Our thanks also goes to Tomáš Davidovič, who has aided greatly on different, mostly hardware related, issues and Lukáš Maršálek, who provided input on some specific Cell problems.

References

[CBEA08] A. Arevalo, R. M. Matinata, M. Pandian, E. Peri, K. Ruby, F. Thomas, C. Almond. Programming the Cell Broadband Engine Architecture: Examples and Best Practices, IBM Redbooks, August 2008

[BWSF06] C. Benthin, I. Wald, M. Scherbaum, H. Friedrich. Ray Tracing on the Cell Processor. Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, 2006

[H01] V. Havran. Heuristic Ray Shooting Algorithms. Ph.D. dissertation, 2001

[H07] V.Havran. Analysis of Fast Branchless KD-tree Traversal Algorithms. Manuscript, Personal Communication, December 2007

[SSK07] M. Shevtsov, A. Soupikov, A. Kapustin. Highly Parallel Fast KD-tree Construction for Interactive Ray Tracing of Dynamic Scenes. Computer Graphics Forum, 26(3), pages 395- 404, September 2007

[W07] I. Wald. On fast Construction of SAH-based Bounding Volume Hierarchies. Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007, pages 33-40, September 2007

[WH06] I. Wald, V. Havran. On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N). Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, pages 61-69, 2006

[WK06] C. Wächter, A. Keller. Instant Ray Tracing: The Bounding Interval Hierarchy. Proceedings of the 17th Eurographics Symposium on Rendering, 2006

[WMS06] S. Woop, G. Marmitt, P. Slusallek. B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. Graphics Hardware 2006, September 2006