Efficient acceleration structure layout for 64-bit many-core architectures

Maxim Shevtsov
Intel Corporation
30 Turgeneva Street,
603024, Russia, Nizhny Novgorod
maxim.y.shevtsov@intel.com

Alexei Soupikov
Intel Corporation
30 Turgeneva Street,
603024, Russia, Nizhny Novgorod
alexei.soupikov@intel.com

a) Asian Dragon model, 7.2M triangles, 64-bit extension consumes only 2Mb of 1.3Gb acceleration structure, extension processing time is <0.5% of rendering time

b) Thai Statue model, 10.2M triangles, 64-bit extension consumes only 2.1Mb of 1.4Gb acceleration structure, extension processing time is <0.5% of rendering time

d) Thai Statue model replicated 7 times 64-bit extension consumes only 4Mb of 7Gb acceleration structure, extension processing time is <0.5% of rendering time

Figure 1. 64-bit extension overheads for large (yet on-core) models. Models are ray traced with shadows at 1024x1024 on a 2-way 3GHz Intel ®Core™2 Duo machine (4 cores, 1 thread/core), 8Gb RAM, Vista64

ABSTRACT

A lot of rendering solutions use an acceleration structure to reduce the complexity of solving geometric proximity search problems. Although acceleration structures are well studied, data exceeding 32 bit address space require an acceleration structure with special properties, such as compact memory layout, efficient traversal capability, memory address space independence, parallel construction capability and 32/64 bit efficiency.

We propose a specific memory layout for a kd-tree and methods of processing that data structure handling massive models with the highest efficiency possible. The components of that are easily applied to other hierarchical acceleration structure types as well.

Keywords

Rendering, acceleration structure, kd-tree, ray- tracing, proximity search.

1. INTRODUCTION

Rasterization or ray tracing of models with large polygon counts usually rely on fast methods of geometrical proximity search. A good quality acceleration structure reduces complexity of the search queries from O(N) to O(log(N)), where N is the number of primitives [Hav01]. The most efficient structures are based on non-balanced binary trees like kd-tree, BVH, BIH [WK06] or BSP, refer to [Hav01] for an overview. An acceleration structure practical for high-speed parallel processing must satisfy the following requirements:

- Efficient traversal capability – compact representation do not slow down the traversal step
- Memory address space independence –the acceleration structure is easy to save/load/transfer

- Parallel construction - the acceleration structure should support creation in multiple parallel threads
- 32 and 64 bit efficiency – the acceleration structure size should not explode on 64 bit architectures. The 32 bit mode acceleration structure mode should have exactly the same binary representation on 64 bit architectures.

In this paper we propose specific memory layout solving the above problems. We use kd-tree as example, but the solution we proposed is also applicable to a wide range of partitioning hierarchies. Furthermore it has a backward compatibility with previous layouts one may have.

2. PREVIOUS WORK

Kd-tree is a binary tree in which each node corresponds to a spatial cell. A kd-tree construction proceeds in a top-down fashion using a cost metric to determine split plane position in a current node until some termination criteria is reached and the node becomes a leaf. An inner node stores splitting plane position and references to the two child nodes. Each leaf node refers to a corresponding list of primitives.

The representation of a non-balanced kd-tree node requires a flag indicating whether the current node is an inner node or a leaf. The inner node stores a single address offset. Adding the offset to the memory address of a given node gives the memory address of the two child nodes [WBWS01]. A kd-tree node occupies eight bytes only. In combination with a proper memory alignment, the layout allows storing the split dimension in the two least significant bits of the offset. The highest bit indicates inner node or leaf, while the remaining 29 bits encode an unsigned address offset (substituting a pointer) to either the child nodes or to the list of primitive indices:

```
/* basic 8-byte layout for a kd-tree node */
struct KDTreeNode {
        union{
                //position of axis-aligned split plane
                float split_position;
                // or number of primitives in the leaf
                unsigned int items;
        }
        unsigned int dim_offset_flag;
//'dim_offset_flag' bits encode multiple data:
// bits[0..1]: encode the split plane dimension
// bits[2..30]: encode an unsigned address offset
// bit[31]: encodes whether node is an inner node or a leaf
};
// macros for extracting node information
#define DIMENSION(n) (n->dim_offset_flag & 0x3)
#define ISLEAF(n) (n->dim_offset_flag & (UINT)(1<<31))
#define OFFSET(n) (n->dim_offset_flag & 0x7FFFFFFC)
```

Figure 2. Basic eight byte kd-tree node layout. Refer to [Ben06] or [Wal01] for details.

Storing offsets instead of pointers makes the data structure independent of its base address, thus no pre-processing is required for storing/loading. The 31st (sign) bit of the offset field as leaf indicator results in efficient leaf/node test if offset are always non-negative. However, a 29 bit offset limits the displacement to 2^{29} bytes, which becomes insufficient for models larger than 10M triangles. Naïve replacement of 4 bytes with 8 bytes to get 61 bit offset on 64 bit machine leads to explosive growth of the memory footprint.

We propose to address that problem with a 64 bit extension mechanism that uses 32 bit offset field for the majority of nodes extending the offset to 64 bits for only a small fraction of nodes. Also a kd-tree fitting into 32-bit address space will have exactly the same binary layout on a 64 bit machine as it had on a 32-bit machine.

Positive offset ([Wal01]) assumes that child nodes are always located at higher addresses than their parents, which limits choices of memory allocation strategies, especially for multi-threaded builds. Construction threads usually allocate memory by continuous chunks. Once chunk is full a thread requests a new region from memory allocation system [SSK07]. A multi-threaded memory allocation system cannot guarantee positive offsets between branches of kd-tree constructed with multiple threads. So using positive offsets require additional transformation pass (similar to [ZHWG08]). We propose using negative offsets and that enables building using multiple regions rather than a single continuous array. To the best of our knowledge in-place construction of a kd-tree in multiple threads has never been done using offset-based representation.

As in [Wal01] we store both children of a node next to each other, both nodes are stored in the same cache line, so they're always fetched together automatically. However we noticed that less care was paid to the leaf/internal node test. As traversing a BSP node is by far the most frequent operation in a ray tracer, it has to be implemented with extreme care. Our leaf node test needs exactly one instruction before branch.

In a compiler field there is an intensive research on automatic pointers compression for 64-bit address space [LA05]. In our case only small number of nodes is really compressed/decompressed. As a result, even for high memory regions granularity the slowdown of rendering is less than 0.5% in compare to having 32-bit offsets only.

3. SOLUTION

A solution we propose still uses only eight bytes for kd-tree node layout. What is really new is how information is encoded and the amount of additional

information we manage to store within the same bytes, refer to Figure 3.

Since nodes and leaf data arrays are naturally aligned by 4-byte boundary, an offset between any two of them has 2 least significant bits available to store additional information:

- values 1, 2, 3 indicate internal node and split plane orientation (correspondingly X, Y, Z axis)
- value 0 is leaf indicator.

Our kd-tree node layout uses least significant bits for node/leaf flag, thus allowing for negative offsets. The changes in layout are highlighted in red below.

```
/* 8-byte layout for a kd-tree node */
struct KDTreeNode {
        union{
            float split_position;
            unsigned int items;
        }
        int dim_offset_flag;
// 'dim_offset_flag' bits encode data in a new way
// bits[0..1] : indicate either
// • a leaf(if set to 0)
   // if 'items' field is >=0 it is true leaf
   // otherwise it is 64-bit extension
// • an inner node with split plane dimension
//   (if set to 1,2,3 for x,y,z axis corresp.)
// bits[2..31] : encode a signed address offset
};
```

Figure 3. The proposed kd-tree node layout. Changes in layout are highlighted in red.

Efficient leaf/node test

During traversal the leaf/internal node test is executed at each traversal step, so its performance is critical. Having 0 as a leaf indicator (Fig.3) allows reducing the test to exactly 1 instruction before branch:

```
and     Node, 0x03
jz      ProcessLeaf
```

Our experiments with the proposed test demonstrated rendering performance improvement of ~5% on average (in compare to Fig.2 layout).

32 and 64 bit efficiency

To handle an unpredictability of a kd-tree size a construction algorithm allocates memory by reasonably sized continuous regions. The algorithm continues construction in the current region until it's full and then requests a new region from memory allocation system, Figure 4. As a result each such region contains a large connected portion of a constructed tree (one or more sub-trees). The number of links between those sub-trees is relatively small (<<1% of total number of links), thus the number of nodes pointing to children located in another memory region is also small.

The typical region size is way smaller than 4GBs. So inside a continuous region the nodes can use 32-bit offsets as far as they reference children within the same region. The only nodes that potentially need 64-bit offsets are the nodes having children located in another memory region. *A node needing 64-bit offset is encoded as a special extension of a regular node.* To avoid frequent checks if a node is extended we extend leaves rather than internal nodes.

Multi-threaded construction

The tree is usually constructed in top-down manner from parent nodes to children nodes. When the tree is constructed in multiple threads each thread builds some sub-tree [SSK07]. Thus different threads may create a parent node and its children nodes. So when a parent is created the offset to children nodes may be unknown. That fact prevents from allocating 64-bit offset data next to a node (when 32-bit offset is insufficient). The actual data of 64-bit extended node is stored in a special per-thread relocation table:

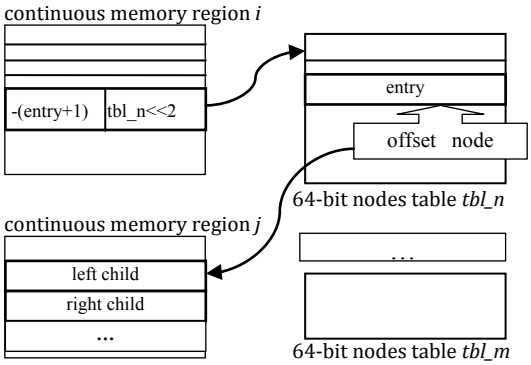

Figure 4. Mem. allocation by continuous chunks

As described in previous section, 64-bit extension node is a special type of leaf:

a) *-(entry+1),* where *entry* is a table entry number, is stored in **items** field (see Figure 3). Negative value indicates special leaf. Adding 1 distinguishes from empty leaf;

b) *(tbl)<<2* where *tbl* is a per-thread table number, is stored in **dim_offset_flag** field. The shift is required to zero 2 least significant bits, indicating a leaf.

Each construction thread creates its own 64-bit node table. So there is no contention between threads for updating or reallocating (when full) the tables. Since each table is small its usage does not affect construction performance. Tests on models with up to 70M polygons demonstrated that 256-entry per-thread tables were never full. Storage or transmission of a tree located in multiple memory regions requires relocation of cross-region offsets. Since 64-bit node tables are exactly nodes with cross-region references,

the relocation operation is a simple update of nodes in the tables rather than a scan and update of the whole tree.

```
struct TableEntry{// relocation table entries
    //actual leaf/node but with zero offset in dim_offset_flag
            KDTreeNode node;
    //true offset
            __int64 offset;
};
#define NOTLEAF(n) (n.dim_offset_flag&0x3)
#define DIMENSION(n) ((n.dim_offset_flag&0x3)-1)
#define IS_64BIT_EXT(n) (n.items<0)
#define MAKELEAF(n,its,ofs) n.items = its; \
   n. dim_offset_flag = ofs;

#define ENCODE64BIT_EXT(n,table_id,entry_id) \
          MAKELEAF(n,-(entry_id+1),table_id<<2)
#define DECODE64BIT_EXT(node, newadr) \
  int tab_id = (node.dim_offset_flag)>>2; \
  int entry_id = -node.items-1; \
  TableEntry e = m_tables[tab_id][entry_id]; \
  newadr = &node + e.offset;   node = e.node;
```

Figure 5. Kd-tree layout+64-bit extensions macros

Modifications of traversal algorithm

The conventional 32-bit tree can be rendered by 64-bit code without any modifications. For the specific 64-bit extensions of the traversal algorithm, refer to Figure 6, Figure 7. Since the probability of traversing leaf is way smaller than probability of traversing internal node, the additional 64-bit extension test is performed at a very small fraction of traversal steps.

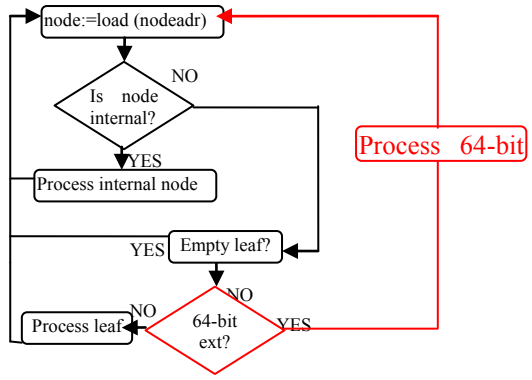

Figure 6. 64-bit extensions for traversal algorithm modification (in red).

4. Performance impact

Tests on large models demonstrated that with the proposed layout memory, footprint of trees constructed on 64-bit machine have almost the same size as the ones constructed on 32-bit machine (i.e. using 32-bit offsets only). Managing per-thread 64-bit node tables in our measurements demonstrated that construction slowdown is <1% and thus is negligible. We also performed tests for 2-128 construction threads with wide range of models (1-100M polygons). For all the tests, 64-entry per-thread tables are more than sufficient to connect portions of a tree constructed with different threads.

The performance of rendering using new layout supporting 64-bit extensions is the same as of rendering the efficient layout supporting 32-bits only (see Figure 1 for examples). Even on complex models and high memory region granularity the slowdown using the proposed layout was less than 0.5% comparing with 32-bit only offsets and one continuous memory region for the whole tree.

```
register KDTreeNode node = m_root;
// ADRINT is  int or __int64 (32/64-bit architectures)
ADRINT newadr = &node;
traverse_loop:
while (NOTLEAF(node)){
    //get dimension, traversal order, etc
            const ADRINT adr0 = newadr+…;//front child
            const ADRINT adr1 = newadr+…;//back child
     //traverse of either back/front child or both
     //…
}
//processing leaves
if(node.items >0){
    //...
}
#ifndef _M_X64
else if(IS_64BIT_EXT(node)){
//64-bit extensions processing:
//newadr is patched using relocation table
            DECODE64BIT_EXT(node, newadr);
    goto traverse_loop; //another option is to duplicate
traversal/leaf- processing code here
 }
#endif// _M_X64
```

Figure 7. Pseudo-code for handling of 64-bit extensions in the traversal algorithm (in italic/bold)

5. Future Work

Transparent support of multiple continuous memory regions that we proposed, allows implementing simple and efficient paging/caching mechanisms in spirit of [YM06].

6. REFERENCES

[Ben06] C. Benthin, "*Realtime Ray Tracing on current CPU Architectures*", PhD thesis, Saarland University, 2006.

[Hav01] V. Havran: "*Heuristic Ray Shooting Algorithms*". PhD thesis, Czech Technical University in Prague, 2001.

[LA05] C. Lattner and V.Adve: "*Transparent Pointer Compression for Linked Data Structures*". In Proceedings of the ACM Workshop on Memory System Performance (2005).

[SSK07] M. Shevtsov, A. Soupikov, and A. Kapustin.: "*Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes*". In Proceedings of Eurographics (2007).

[Wal01] I. Wald, "*Realtime ray tracing and interactive global illumination*", PhD thesis, Saarland University, 2004.

[WBWS01] I. Wald, C. Benthin, M. Wagner, and P. Slusallek, "*Interactive Rendering with Coherent Ray Tracing*". Computer Graphics Forum, 20(3) (2001).

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

[YM06] S.-E. Yoon, D. Manocha: "*Cache-Efficient Layouts of Bounding Volume Hierarchies*". Computer Graphics Forum 25(3) (2006).

[ZHWG08] Kun Zhou, Qiming Hou, Rui Wang, Baining Guo: "*Real-time KD-tree construction on graphics hardware*" In Proceedings of ACM SIGGRAPH Asia (2008).