Understanding the Efficiency of kD-tree Ray-Traversal Techniques over a GPGPU Architecture

Artur Santos · João Marcelo Teixeira · Thiago Farias · Veronica Teichrieb · Judith Kelner

Received: 3 February 2011 / Accepted: 20 August 2011 / Published online: 29 September 2011

Abstract Current GPU computational power enables the execution of complex and parallel algorithms, such as ray tracing techniques supported by kD-trees for 3D scene rendering in real time. This work describes in detail the study and implementation of eight different kD-tree traversal algorithms using the parallel framework NVIDIA Compute Unified Device Architecture, in order to point their pros and cons regarding performance, memory consumption, branch divergencies and scalability on multiple GPUs. In addition, two new algorithms are proposed by the authors based on this analysis, aiming to performance improvement. Both of them are capable of reaching speedup gains up to 3$\times$ when compared to recent and optimized parallel traversal implementations. As a consequence, interactive frame rates are possible for scenes with 1,408 $\times$ 768 pixels of resolution and 3.6 million primitives.

Keywords Ray tracing · kD-tree · Traversal · CUDA

A. Santos (✉) · J. M. Teixeira · T. Farias · V. Teichrieb · J. Kelner
Computer Science Center, Virtual Reality and Multimedia Research Group, Federal University of Pernambuco, Recife, Brazil
e-mail: als3@cin.ufpe.br

J. M. Teixeira
e-mail: jmxnt@cin.ufpe.br

T. Farias
e-mail: tsmcf@cin.ufpe.br

V. Teichrieb
e-mail: vt@cin.ufpe.br

J. Kelner
e-mail: jk@cin.ufpe.br

1 Introduction

Ray tracing has been used, since its first appearance with the work of [1] in 1968, as an effective technique applied to generate detailed images based on natural physics phenomena simulation. Many efforts were done by the scientific community to establish ray tracing techniques that reach lifelike 3D scene renderization [2]. As any computing technique, ray tracing has its advantages and drawbacks. While the render results achieve an accurate simulation of the real world, the computational power demanded for this task made unfeasible the use of the technique for real time applications. The main goal of a ray tracing program is to cast rays from the virtual camera, passing through every pixel on the image and deciding what color the pixel should have. The color placement is based on details regarding the intersection between rays and scene objects, considering as well light sources, material properties and the scene itself.

The main application areas for ray tracing generated images are the computer graphics animation, special effects addition on movies and offline content generation systems. Despite being computationally expensive, an advantage comes with the implementation of a ray tracing based renderization: physical behaviors are easy to code and to represent in these systems. Thus, shading, reflection, refraction, shadow generation, global illumination, sub surface scattering, and many other features become ray generation constraints and material properties.

There is no sense in using ray tracing techniques without considering space partition algorithms and search optimizations. This kind of data structures speeds up the intersection test of each primitive present in a scene (triangles, spheres, Bezier surfaces, etc). The most usual space partition structures are kD-trees [3], which have many algorithms to perform the ray traversal.

In spite of the speed up reached with the use of spatial queries data structures, they are not enough to make ray tracing suitable for real time applications. Although, the ray tracing problem is highly parallelizable, since parts of the image can be processed in separate. According to this fact, the image could be distributed through a cluster (often named render farms), or multicore processors and/or coprocessors to run the entire solution on a single machine, still having a good result.

Since GPU pipelines became programmable, they are used as a coprocessor for implementing highly parallelizable algorithms, such as ray tracing. GPU programming has become even more suitable for general purpose problems since the programming model was unified by the NVIDIA CUDA [4] architecture, which delivers a many-core solution containing parallel stream processors. This work exploits this architecture to implement spatial data structures that run queries in parallel in order to achieve real time frame rates for ray tracing techniques.

Furthermore, this work contextualizes the scenario of most used algorithms for kD-tree traversal, from the standard version to the proposed adaptations. Ten techniques were implemented using CUDA in order to evaluate their real time performance, where two of them are novel approaches (Ropes++ and 8-byte Standard). The techniques were used at the core of a highly optimized GPU real time ray tracer for static scenes in order to achieve the results presented in the following sections. A comparison between the Ropes++ technique and the NVIDIA OPTIX [5] ray tracing engine, resulted in a favorable argument to the new approach evaluation and analysis.

Thus, it could be stated that Ropes++ surpasses the performance of current GPU counterparts.

2 Acceleration Structures

Ray tracing techniques commonly present a high computational cost since they need to perform, for each ray **r** emitted into the scene, a search for the closest geometric primitive intersected by **r**. A simple way of solving this problem is to test intersections between **r** and every primitive in the scene [6]. Since this is a "brute-force" approach, search complexity increases linearly depending on the amount of objects present in the scene [3].

Due to the fact that ray–object intersection involves dozens of floating point arithmetical operations, it is costly in most computer architectures. Consequently, this approach becomes prohibitive for real time ray tracing of scenes containing millions of geometric primitives. Using this method, even for scenes with a few hundred primitives, about 95% of processing time is spent with intersection calculations [6].

This problem can be minimized by the use of special data structures capable of spatially organizing or grouping scene objects, in order to considerably reduce the amount of intersection tests involved in the search. This way, instead of following a "blind" search for the closest intersection, only objects near to the path of the ray are tested, discarding the remaining ones. Since these remaining objects comprise the most of the scene, the increase in performance is significant when compared to the "brute-force" approach. It can be proved [3] that such structures can reduce search complexity to a sub-linear level. They can be created by using concepts such as spatial scene subdivision (Partitioning Space, like BSP Trees [3,7]), subdivision involving sets of objects (Partitioning Object List, like BVHs [8]), or hybrid approaches taking benefit of both previous concepts [9]. Among many existing acceleration structures, the ones most used in ray tracing are related to Binary Space Partitioning Trees (BSP Trees). Such tree recursively divides the scene at each branch, separating all scene objects into sub-groups, as shown in Fig. 1.

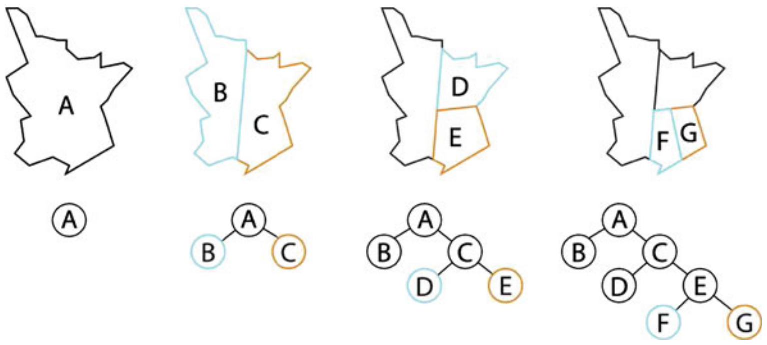

Fig. 1 Binary Space Partitioning tree (*BSP*)

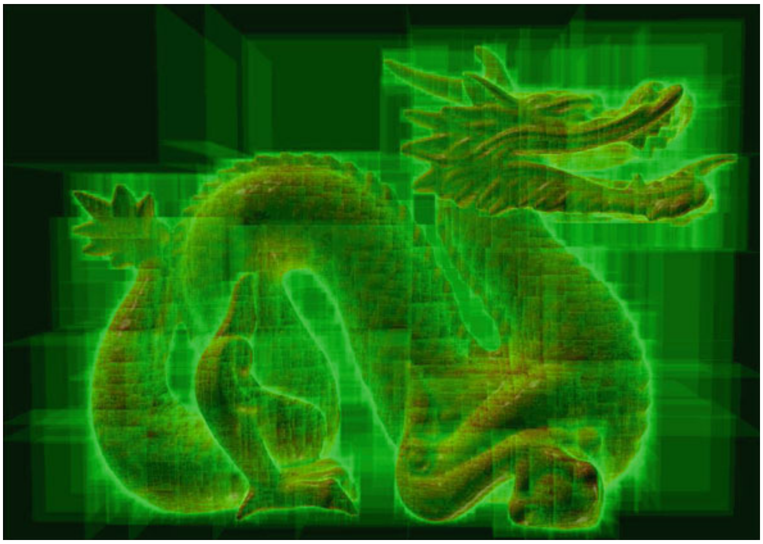

Fig. 2 Stanford Dragon model subdivided into small bounding boxes. *Light areas* represent high density kD-tree nodes

The recursive BSP subdivision is originated from a chosen splitting plane located on each branch of the tree. The splitting plane choice can be restricted to one of the coordinate axes, thus simplifying the search algorithm. This specific tree is known as kD-tree (k Dimensions-tree), as illustrated in Fig. 2. On a kD-tree, every internal node stores the dimension and position of the splitting plane that will divide the scene in two sub-spaces, and the node's children function as representatives of such sub-spaces. The leaves represent the smallest sub-spaces of the tree and store a list of objects located within them. Since the normals of kD-tree splitting planes are always canonical vectors of world coordinates, each scene sub-space is represented by Axis-Aligned Bounding Boxes (AABBs). In this case, the tree's root node represents the scene bounding box. This way, every kD-tree search (also known as traversal algorithm) is reduced to traverse the ray through the nodes bounding boxes, performing ray-AABB intersection tests until intersecting an object inside the AABB of a leaf node, or until the ray leaves the scene bounding box without intersecting any object. Since a single axis is used as splitting plane support for each node, the ray-AABB intersection test per node traversal is simplified to one dimension, which considerably reduces the number of arithmetical operations used.

3 Related Work

Recent efforts regarding ray tracing have been about optimizing existing algorithms and data structures to achieve the best image quality/speed on given hardware architectures. The modification of these algorithms guides optimizations on such hardware architectures to better suit the needs of computer graphics developers.

Modern CPU implementations take benefit of coherent rays [10] when using them with packet traversals [11], by grouping the search computation of such rays in groups of SIMD (Single Instruction, Multiple Data) instructions. However, a modern GPU SIMT [4] (Single Instruction, Multiple Threads) architecture seems to be even more adequate to process coherent rays, since the traversal algorithm does not always need to be manually modified and as consequence no overhead is added, unlike happens with SIMD packet traversals.

Furthermore, as graphics hardware has become increasingly programmable and suited to more general-purpose computation, developers were attracted to this platform. By bringing hardware and software together, it is possible to explore how to best design hardware as well as software systems and what are the best interfaces between them. As witnessed in recent years, the cycle of rapid innovation in hardware systems enabling new software approaches, and innovative new graphics software driving the hardware architectures of the future has brought great benefit to the area of high performance graphics.

Horn et al. [12] take advantage of the GPU processing power to perform ray tracing on an X1900 XTX. Their implementation uses ATI's CTM (Close To the Metal) toolkit and pixel-shader 3.0. The entire process runs at 12–14 frames per second, over 1,024 $\times$ 1,024 scenes with shadows and Phong shading.

Popov et al. [13] propose a stackless packet traversal algorithm that supports arbitrary ray bundles and handles complex ray configurations efficiently. Their GPU implementation uses CUDA and runs on a GeForce 8800 GTX. Allowing primary rays only, they achieve about 15 frames per second over scenes with 1,024 $\times$ 1,024 pixels of resolution.

Gunther et al. [14] present a bounding volume hierarchy (BVH) ray tracer with a parallel packet traversal algorithm. Using a GeForce 8800 GTX, they achieve 3 frames per second for the well-known Power Plant scene, with 12.7 million triangles and 1,024 $\times$ 1,024 pixels of resolution.

All these works converge taking advantage of the capabilities of hardware systems. Exploiting existing hardware running novel algorithms and data structures, improving both the hardware and the software, and discussing the best applicable programming models and interfaces to hardware capabilities are key challenges facing all of us.

3.1 kD-tree Ray-traversal Techniques

The first known kD-tree ray-traversal algorithm was introduced by Kaplan [15]. This traversal (later referred by Havran [3] as Sequential traversal) is based on a sequential, but also cyclic top-down point-location search along the ray path within the kD-tree nodes. This way, interior nodes (specially the root node) have to be repeatedly visited. This concept is illustrated in Fig. 3.

When intercepting one node, a ray can traverse just one or both children, as shown in Fig. 4. For traversing both children, in the Standard traversal algorithm [3], a stack is needed in order to store the child-node located most far away (far child). This information may be used in a posterior search (see Fig. 3), since the traversal first follows the near child path, and later the far child one. The stack guarantees that the traversal

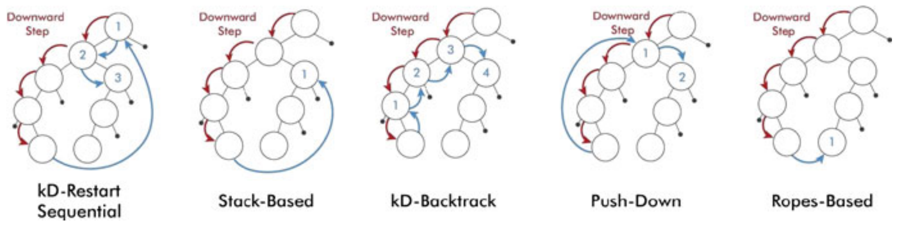

Fig. 3 Different approaches used in ray-node traversals. The *darker arrows* represent the down-traversal to leaf nodes, while the *lighter ones* guide the search to the next unvisited node

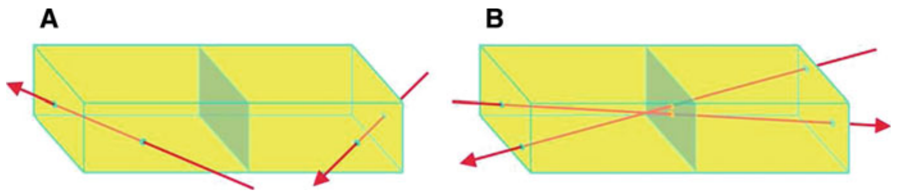

Fig. 4 Ray-node intersection cases: **a** one child, **b** both children

occurs according to the order in which bounding boxes are intercepted by the ray, visiting a single leaf node at a time.

Using statistical analysis of ray-node intersection cases, Havran [3] modified the Standard traversal algorithm, achieving a more robust and faster CPU ray-traversal algorithm. In Havran's tests, his traversal algorithm implementation on a MIPS architecture achieved up to 65% of speedup compared to Standard traversal.

During last years, the fast computational power growth of graphics processors has attracted the attention of researchers seeking for efficient ray tracer GPU implementations. This work was also focused on kD-trees. Foley et al. [16] have adapted the Standard traversal algorithm, creating two new traversal algorithms that do not require a stack for the search, reducing "memory footprint". These modifications made possible a kD-tree traversal implementation on graphics cards with limited programmable stack memory.

The so called kD-Restart algorithm starts over the traversal to the tree root every time it reaches a leaf of the tree, as shown in Fig. 3. This "restart" operation continues until an intersection with any primitive is found or the ray leaves the scene bounding box. It is not necessary anymore to store a possible far child node, since the down traversal is always initiated in the root node. This eliminates the need for the stack required by the Standard algorithm. However, the average cost of this search becomes considerably higher than the Standard one, since many nodes are revisited when search goes back to the root.

In order to reduce the amount of revisited nodes, the second algorithm, called kD-Backtrack, stores in each node its bounding box (AABB) and a pointer to its parent. Using the parent pointer, it is not necessary anymore to return to the root every time a leaf is found; the algorithm just needs to return to a node that enables traversing other unvisited nodes, as shown in Fig. 3. In spite of reducing the number of visited nodes, the kD-Backtrack algorithm demands about an order of magnitude of more primary memory to store the entire tree in comparison to the Standard algorithm, due to the necessity of keeping information about parents pointers and AABBs. Therefore, kD-Backtrack algorithm is not suitable for complex scenes, with more than 10 million polygons.

In order to obtain a performance closer to the Standard traversal algorithm one, Horn et al. [12] proposed a few modifications to the kD-Restart algorithm to reduce the total amount of visited nodes. The Push-Down technique keeps the last visited node **n** that can be considered as query root. In such node, the ray hits only one of the node's children. This way, whenever a restart event occurs, instead of returning to the root, the search restarts at node **n**, which reduces the amount of revisited nodes, as illustrated in Fig. 3.

Besides the Push-Down technique, Horn et al. [12] proposed the Short-Stack algorithm, which uses a small circular stack instead of the regular tree-depth size stack used in the Standard algorithm. In case the stack empties and the ray did not hit any primitive, the restart event is processed in the same way of the kD-Restart, redirecting the query to the root.

Push-Down and Short-Stack algorithms optimize different parts of kD-Restart and can be used together. In this case, if a restart event is raised in Short-Stack, instead of returning to the root, the search goes back to the node previously stored by Push-Down. We call such hybrid traversal PD&SS.

Popov et al. [13] demonstrate a CUDA implementation of Ropes, a stackless traversal using the concepts of ropes [17], in which each leaf stores its corresponding bounding box and 6 pointers (ropes) to the neighbor nodes of the faces of its bounding box. This way, when a leaf is reached, instead of returning to a node stored in the stack, one of the 6 stored pointers is used. This concept is shown in Fig. 3. Consequently, the traversal using ropes visits fewer nodes than every algorithm cited before, with the cost of having to perform more memory accesses for fetching bounding boxes and rope information.

4 Proposed Implementation

Global lighting techniques such as ray tracing are based on basic concepts of Optics [2] in order to render more realistic scenes in comparison to rasterized ones. Images with complex visual effects like transparent or reflexive objects are easily obtained with the use of ray tracing, being this technique indicated for more precise renderings, with plenty of details.

Our implemented ray tracing system supports reflections and shadows according to the simple model described by Whitted [6]. Due to hardware limitations, CUDA does not support recursive calls. As a consequence, we adapted the standard recursive algorithm to an iterative one. By using only an instance of the primary ray, whenever an intersection is found, the ray is translated to the intersection point and has its direction modified according to the type of ray to be generated (shadow or reflected ray). The iterations end when the reflected ray does not hit any object or when it reaches the maximum level of reflection previously determined.

4.1 Ray Tracer Kernel

Our implementation distributes the workload of each pixel to different threads, in a way that **n** pixels are processed by **n** parallel threads. Since CUDA presents itself as a SIMT (Single Instruction, Multiple Thread) architecture, it is important that near threads follow the same sequence of instructions whenever possible, to guarantee that the instruction is simultaneously executed in all threads. Because of that fact, near pixels are treated by near threads, favoring parallel execution. Since primary rays generated by near pixels have equal origins and similar directions, there is a high probability that they perform a similar traversal on the scene, intersecting the same objects while executing the same group of instructions. Such alike rays are known as coherent rays [10].

By assigning the ray tracing of each pixel to one corresponding thread, it is possible to use a single kernel for processing the entire algorithm in GPU, including the kD-tree traversal and shading. This way, each CUDA block of threads represents an image sub-region to be rendered, commonly known as tile, as shown in Fig. 5. This approach offers less overhead in comparison to multi-kernel implementations, since for each change of pipeline stage it is necessary to store in global memory all the information that must be passed to the next stage.

However, the single kernel approach is not the best choice when dealing with scenes composed by many reflective objects: a significant amount of low coherent secondary rays is generated, harnessing the degree of parallelism due to divergent instructions and random accesses to memory. This is a known problem of CPU implementations that becomes far more serious on GPUs. Besides that, it is not difficult to find cases when only a small amount of the block generates secondary rays, in a way that many threads stay in "idle" state, waiting for the busy ones to complete. GPU resources are only made available when the entire block of threads being used has finished its processing, which does not occur in CPU, where a new pixel is processed every time one is finalized.

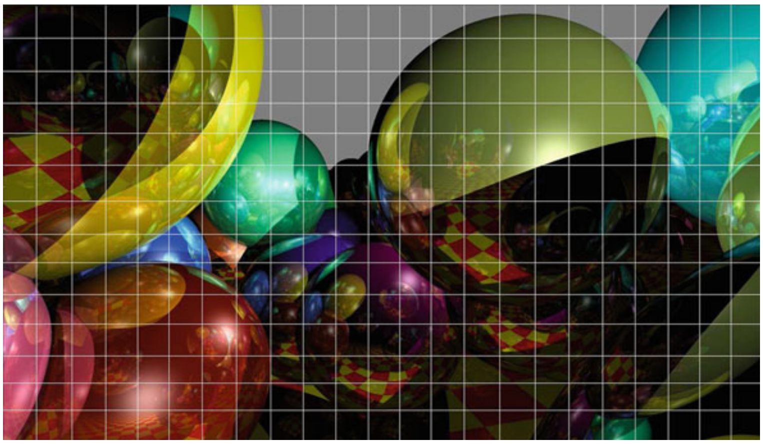

Fig. 5 Tiles used to process neighbor pixels

Given those problems, the implemented ray tracer makes use of a single kernel approach whenever rendering just primary rays (ray casting). In case of scenes with reflective objects, a new approach for dealing with secondary rays based on "screen space" compression is used in order to reduce the amount of time wasted by idle threads in the same warp (group of 32 neighbor threads running in parallel, in the same multiprocessor). To solve the problem of idle threads in different warps, but in the same block, our implementation takes advantage of persistent threads [18], in a way that kernel functions are changed to allow the programmer to create a pool of tasks for each warp. Using this pool of tasks, the warp has some work to do until all pixels are processed. Instead of creating a pool for each warp, our implementation creates a pool (in shared memory) for each multiprocessor and then, each multiprocessor pool is used as "task source" for the warps. This way, the number of costly atomic operations on global memory (necessary for task allocations) is proportionally reduced from the number of parallel warps to the number of multiprocessors.

4.2 Secondary Rays

In order to lessen the secondary rays problem, the pixel computation is subdivided into phases, each one corresponding to a new generated reflective ray to be computed.

At first, the kernel responsible for processing all primary rays is executed. A screen space Boolean mapping is created as a result of this first phase, in which pixels with value 1 represent the need for secondary rays. After that, the Boolean map is compressed using a parallel prefix sum technique [19]. Such map enables the execution of a new ray tracing kernel in which all active pixels are grouped into contiguous blocks. This guarantees a balanced processing load for every thread in a warp, and consequently increases the thread occupancy on the stream processors by reducing the amount of idle threads. Despite the overhead caused by more kernel invocations and more global memory usage, this approach enables gains up to 20% in comparison to the single kernel approach. It is important to notice that each one of these new kernels is also implemented using the concept of persistent threads [18].

4.3 Memory Usage

Since this work is focused on a comparative study of ray-traversal implementations in GPU and consequently does not require scene reconstruction at every new frame, the kD-tree construction process is performed by the CPU and the structure built is later sent to the GPU. The kD-tree construction utilizes a spatial subdivision technique based on Surface Area Heuristic with perfect splits [20]. As result, it is possible to pre-build a high quality kD-tree for complex scenes, with more than one million objects.

On the GPU side, both kD-tree and geometrical/material properties of objects are stored in global memory. In order to reduce the access penalty to such type of memory, texture bindings on global memory are used. This artifice provides an 8 KB cache system with 256 bytes of cache line size on current graphics cards.

Despite the fact that ray tracing presents a high degree of parallelism, there is also a high demand for bandwidth due to the great amount of accesses to global memory.

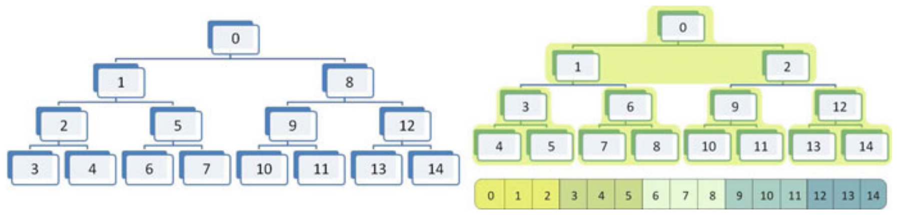

Fig. 6 Depth-first tree layout (*left*) versus van Emde Boas tree layout (*right*)

All threads from our implementation realize a considerably high number of memory accesses searching for objects in the scene. In case these accesses are performed inadequately (randomly, for example), the obtained results may not reach performance expectations.

In order to reduce the number of accesses to global memory, our implementation organizes the 3D scene in a way that geometric primitives near in space may be located in neighbor memory blocks. This way, by traversing the scene, the ray will probably intersect near objects based on the principle of data locality. This relationship is analogous, since near objects are stored in memory addresses close to each other, which helps increasing the cache hit rate for such objects.

The scene kD-tree is specially designed to increase the cache hit rates. For that, the tree is structured in a way a node can fit in 64 bits, following a depth-first order, shown in Fig. 6. The van Emde Boas Implicit Layout [3,21] shown in Fig. 6 was also tested, but it was slower in GPU, because it needed 128 bits per node, requiring more global memory reads. Since the layout doesn't affect the ray-traversal algorithms, our comparative study is based solely on the faster, 64 bits node layout.

When the CUDA compiler is not capable of reducing the number of necessary variables in a certain scope in order to maintain a fixed number of used registers, all "extra" variables are placed in local memory. Local data access might be up to 600 times more to access data in local memory, in comparison to register access. The shared memory can be used as a support for storing variables. This is advantageous since the transfer time between registers and shared memory is about 4 clock cycles [4] for accesses without bank conflicts, which costs the same as a floating point "add" operation.

Therefore, the ray structure, represented by an origin point and a direction vector, is stored in shared memory. This way, ray information is used in every phase of the algorithm and the use of registers is significantly reduced. Another advantage in storing ray information in shared memory space is the ability of using memory indexing, which is necessary in ray-triangle intersection tests and kD-tree traversal. Since in compile time it is not known which axis will be used, it is necessary to represent the axes as a 3-value vector, whose indexes vary from 0 to 2. In case kernel code indexes a vector variable, it will be automatically stored in local memory in order to allow the indexing operation. By using shared memory it is possible to overcome this problem, maintaining the indexing and performing faster memory operations.

CUDA constant memory space is used to store most kernel fixed parameters, such as virtual camera configurations and maximum ray tracing depth. CUDA compiler automatically stores parameters used by kernel functions in shared memory space.

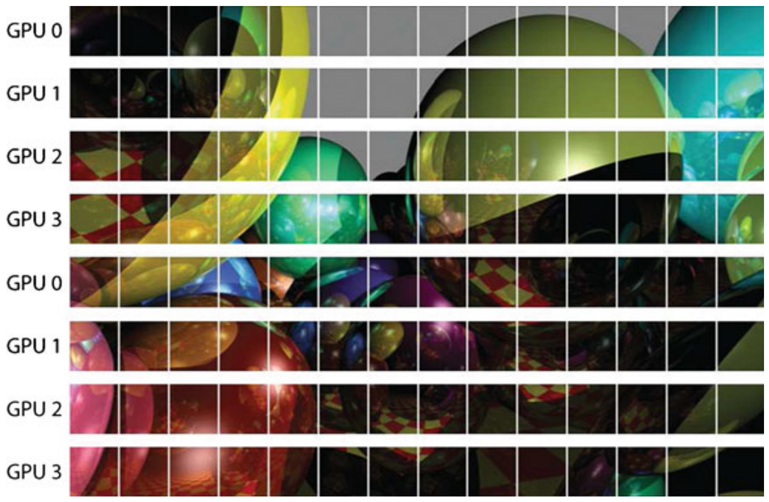

Fig. 7 Work balance between multiple GPUs using the interlacing blocks approach

Because of the fact that this memory region is already being used for storing data regarding the rays, we decided to manually place these parameters in constant memory. Since constant memory access in CUDA is cached and consequently fast, this does not present significant performance losses in comparison to shared memory usage.

4.4 Multi-GPU

It is possible to take advantage of multiple graphics cards attached to the same computer in order to achieve a better performance. Each device has his own memory space, and depending on the type of the problem, it may be necessary to create copies of the same data to be processed for each device. It can be difficult to efficiently merge the partial results from each device into a single memory space, usually resident in the primary memory of the computer. This is the case of our ray tracer: all the scene is completely loaded by each device and the computation result (pixels) is merged into one array by the CPU.

Our ray tracer takes advantage of multiple CUDA devices by splitting the screen in rows of tiles to be processed by each device, following an interleaved pattern, as shown in Fig. 7. Interleaved lines achieved better load balance between the devices, but worsened performance, what can be explained by the reduction of ray coherence between neighbor pixels when compared to interlacing by block of contiguous lines.

4.5 Ropes++

The new traversal approach proposed in this work (Ropes++), shown in Algorithm 1, is based on the Ropes traversal [17]. The Ropes++ reduces the number of global memory reads and arithmetical operations necessary to the leaf traversal. To accomplish that, the intersection algorithm of the original Ropes algorithm is adapted in a way that the test is reduced to locate the AABB's exit point by reading only half of the AABB information. This is possible based on the fact that the exit face of a ray-AABB intersection depends only on the ray's direction, and that there are always three possible intersection faces, one for each ray axis direction. The reduction of memory reads is achieved by to the substitution of the access to six floats, representing the leaf bounding box, by the three floating point accesses necessary to the AABB's exit faces calculations. Since only three floats are necessary instead of six, the register pressure is slightly reduced, enabling a higher number of threads effectively processed in parallel.

Our rope traversal also performs a simplification of the first ray-scene intersection test, initially calculating only the entry point distance. Based on the fact that when a ray does not intersect the scene's AABB it will not intersect a sub-scene volume either, the ray-scene test can use as exit point distance the intersected exit face of the first traversed leaf, computation that is always necessary for ray-scene hit cases.

Algorithm 1 Modified traversal with ropes (Ropes++)

```
1: Ray $r$ = (org, dir); {Origin Point + Direction Vector}
2: Node $node$ = root;
3: Float $tMin$ = Entry distance of **r** in the tree; {only $tMin$ is necessary at this point}
4: Float $t$; {distance ray-primitive}
5: Primitive $intersected$; {possibly intersected primitive}
6: **repeat**
7:    Point $pEntry = r.\text{org} + tMin * r.dir$;
8:    **while** $\neg node$.isLeaf() **do**
9:       **if** $pEntry[node.\text{axis}] \leq node.\text{splitPosition}$ **then**
10:         $node = node.\text{left}$;
11:      **else**
12:         $node = node.\text{right}$;
13:      **end if**
14:   **end while**
      {AABB represented by 6 floats (min and max 3D points)} {We only need to test against 3 parameterized values} {Branchless ray-AABB exit point intersection test: $r.\text{dir.axis} \geq 0$}
15:   $localFarX = (node.\text{AABB}[(r.\text{dir.x} \geq 0)*3] - r.\text{org.x})/r.\text{dir.x}$;
16:   $localFarY = (node.\text{AABB}[(r.\text{dir.y} \geq 0)*3 + 1] - r.\text{org.y})/r.\text{dir.y}$;
17:   $localFarZ = (node.\text{AABB}[(r.\text{dir.z} \geq 0)*3 + 2] - r.\text{org.z})/r.\text{dir.z}$;
18:   $tMax = \min(localFarX, localFarY, localFarZ)$;
19:   **if** $\text{tMax} \leq \text{tMin}$ **then**
20:      **return** MISS;
21:   **end if**
22:   **for all** Primitive $p$ in $node$ **do**
23:      I = Intersection($r,p,tMin,tMax$);
24:      **if** $I \neq$ null **then**
25:         Update $t$ and $intersected$, using best (smaller $t$) result;
26:      **end if**
27:   **end for**
28:   **if** $intersected \neq$ null **then**
29:      **return** HIT($intersected,t$);
30:   **end if**
31:   $exitFace = \text{minLocalFarAxis} + (r.\text{dir}[\text{minLocalFarAxis}] \geq 0)*3$;
32:   $node = node.\text{ropes}[exitFace]$;
33: **until** $node ==$ null
34: **return** MISS;
```

4.6 8-bytes Standard Traversal

The Standard traversal is a simple and elegant way of traversing a kD-tree. However, the stack requirement has a high performance impact on GPU implementations when compared to CPU ones, since the last one has a very efficient cache system, significantly reducing the stack access penalty. Since a high quality kD-tree is not balanced, tipically presenting a depth greater than one hundred, the only two CUDA memory spaces capable of allocating the stack are the global and local memories. Accesses to local memory are as expensive as accesses to global memory [4], but accesses to local memory are always coalesced, since they belong to a per-thread scope. This justifies the decision of placing our stack in local memory.

Previous works [3,12,15] related to the Standard traversal follow a stack element model of three variables, which store information necessary to keep the traversing going on without returning to the root node each time the algorithm reaches a leaf node. These three stored values are the index of the far node to be traversed later and the parametric ray interval (**tMin** and **tMax**) to be traversed. The two parametric values represent the intersection points between the ray and the bounding box of the far node. Since each one of this information can fit in a 32 bit variable, they are usually stored in a 12-byte struct format. After implementing all ray-traversals, it became clear that, with some small modifications to the original algorithm, it was possible to remove one of the three information, the **tMin** variable, to create a smaller struct of 8 bytes instead of 12. The unnecessary stacked information was actually always present during the traversal, stored in the last **tMax** value found after reaching a leaf node. Imagining the ray in a continuous path, **tMax** represents the last point traversed in the ray, meaning that this last point will also be the first point to be traversed when the algorithm pops from the stack the last far node to be traversed. Then, the **tMin** stored in the stack will have the same value as the **tMax** found in the previous leaf node. This concept is shown in Fig. 8. Since the stack has 8-bytes per element, we name it 8-bytes Standard traversal, and the default implementation as 12-bytes Standard traversal. It is important to notice that this new stack format can also be used in the short-stack algorithm. The 8-byte Standard traversal is shown as pseudo-code in Algorithm 2.

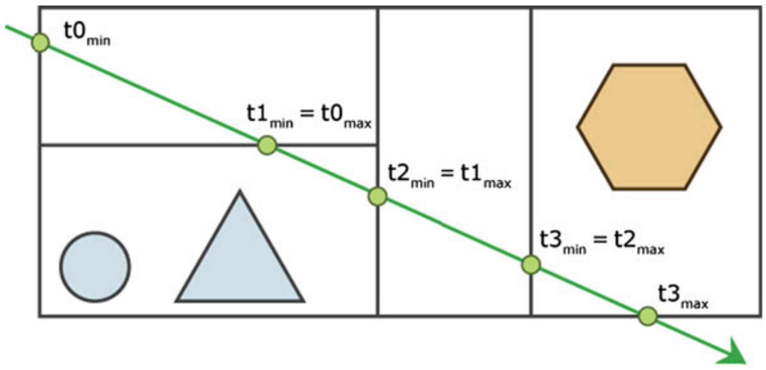

Fig. 8 Equivalence between current leaf's tMax and next node's tMin

Algorithm 2 8-byte Standard traversal

```
1: Ray $r$ = (org, dir); {Origin Point + Direction Vector}
2: Primitive *intersected*; {possible intersected primitive}
3: Float $t$; {distance ray-primitive}
4: Stack < Node, Float > *stack*; {traversal stack (8 bytes per element)}
5: Float *tMin*,*tMax* = Entry/Exit distance of **r** in the tree;
6: Node *node* = root; {current Node}
7: while TRUE do
8:    while ¬node.isLeaf() do
9:       axis = node.axis();
10:      diff = node.splitPosition - r.org[axis];
11:      tDist = diff / r.dir[axis];
12:      (nearNode,farNode) = getOrderedNodes(r,node);
13:      if $tDist < 0 \vee tDist \geq tMax$ then
14:         node = nearNode;
15:      else
16:         if $tDist \leq tMin$ then
17:            node = farNode;
18:         else
19:            stack.push(farNode,tMax);
20:            node = nearNode;
21:            tMax = tDist;
22:         end if
23:      end if
24:   end while
25:   for all Primitive p in node do
26:      I = Intersection(r,p,tMin,tMax);
27:      if I ≠ null then
28:         Update t and intersected, using best (smaller t) result;
29:      end if
30:   end for
31:   if intersected ≠ null then
32:      return HIT(intersected,t);
33:   end if
34:   if stack.isEmpty() then
35:      return MISS;
36:   end if
37:   tMin = tMax;
38:   (node,tMax) = stack.pop();
39: end while
40: return MISS;
```

Both CPU and GPU implementations can take advantage of this new approach, since it is not a concept dependent of architecture. One direct advantage in GPU is the possibility to create even deeper kD-trees, since there is more space available for the stack. Another one is faster memory accesses, since in local memory a 8-byte (64 bits) load is faster than a 12-byte (96 bits) one [4]. A comparative analysis between these two traversal aproaches is described in the next section.

5 Comparative Analysis and Results

This section shows the differences between all the traversal algorithms, in many different aspects, such as number of traversal steps, execution time related to number of

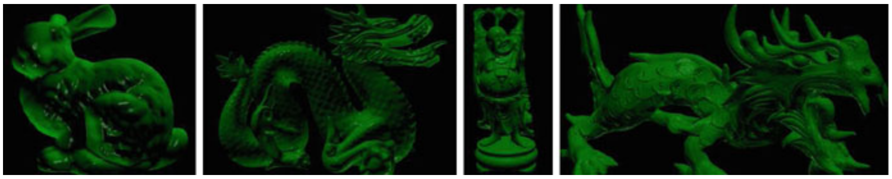

Fig. 9 Models used in tests. From *left* to *right*: Bunny, Dragon, Happy Buddha and Asian Dragon

reflections and scene complexity, number of divergent branches, multi-GPU analysis and kD-tree memory consumption. The results were collected using an Intel Core i7 2.66 GHz CPU with 12 GB of RAM and two NVIDIA GeForce GTX 295 (for both single and multi-GPU tests) in Quad-SLI mode, running Microsoft Windows 7 Professional 64-bit and using CUDA 3.0 toolkit. With the intent to ease comparisons between previous and future publications, all 3D models used in the tests were taken from the Stanford 3D Scanning Repository [22], comprising four different models: Bunny (69k triangles), Happy Buddha (1.08 million triangles), Dragon (1.0 million triangles) and Asian Dragon (3.6 million triangles). These different models are shown in Fig. 9. The original Stanford Asian Dragon has more than 7.2 million triangles. Since a kD-tree of 7 million triangles exceeds the maximum memory space of the GTX 295 card, we had to use a 3D editor tool to simplify its mesh, reducing by half its original triangle count.

Despite the fact that each traversal algorithm follows a different search concept, it is possible to notice a certain similarity in their structures. Initially, before the search itself is performed, all algorithms execute an intersection test between the ray and scene bounding box. This operation occurs in such a way that, in case the ray does not hit the scene bounding box, there is no meaning in performing the kD-tree traversal. Another reason for this initial test is that most traversal algorithms store both entry and end intersection points distances, for later use as traversal end conditions.

Both Sequential and kD-Restart algorithms do not need a stack, which makes their codes simpler and with less variables, reducing the number of registers used per thread. However, when the search reaches a leaf and goes back to the root, many nodes are revisited, and consequently results in a higher amount of global memory reads, as shown in Fig. 10.

Compared to kD-Restart, the stackless kD-Backtrack algorithm achieved a lower number of traversal steps, as shown in Fig. 10. However, the amount of global memory reads was increased mostly due to additional loads of parents indices necessary to the backtracking process. Since this algorithm needs to store the nodes' AABBs, it approximately consumes 3 times more memory space, as shown in Fig. 11.

The Push-Down reduces the number of visited nodes (see Fig. 10) when compared to kD-Restart. To accomplish this, it adds control structures to the code in order to avoid returning the search to the tree's root. However, such control structures add costly branch divergences (shown in Fig. 12) that consequently decrease performance.

Regarding Short-Stack, our implementation allocates the stack in shared memory, allowing a maximum size of 3 nodes, without the possibility of stack overflows, due to the circular nature of the structure. Accesses to this short stack are faster than the

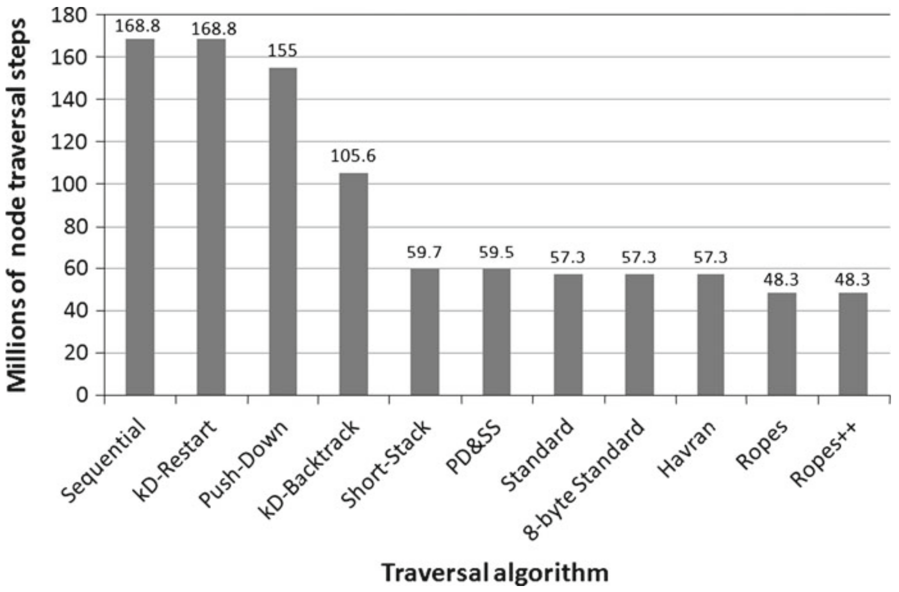

Fig. 10 Number of node traversal steps for each algorithm from a fixed viewpoint using the Asian Dragon model (3.6 million triangles, 1,408 × 768 resolution, single GPU)

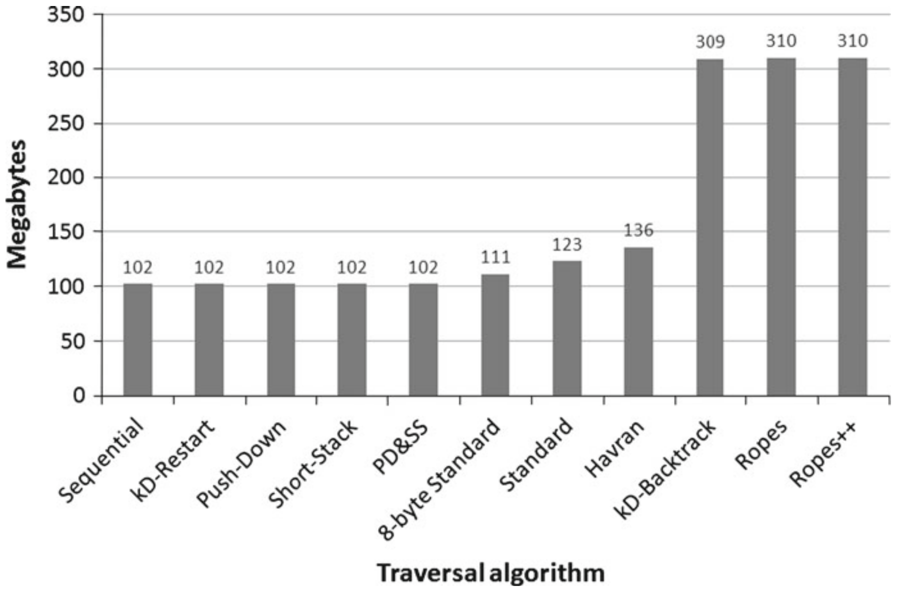

Fig. 11 kD-tree size in megabytes for each algorithm using the Asian Dragon model (3.6 million triangles, single GPU)

Standard algorithm since it is located in shared memory. This approach surpasses kD-Restart's performance as well. However, similar to the kD-Restart, the stack's limited size forces a return to the root and as consequence new searches to already visited

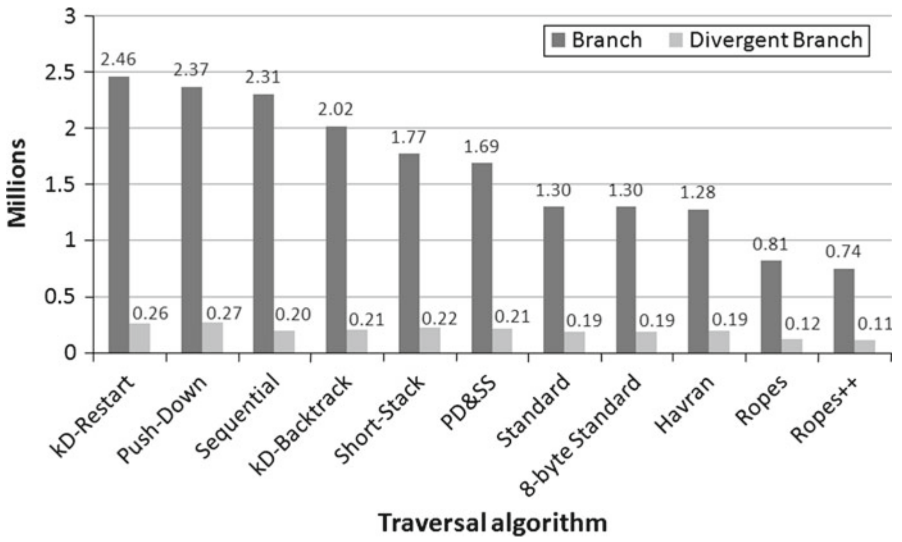

Fig. 12 Number of branches and divergent branches for each algorithm from a fixed viewpoint using the Asian Dragon model (3.6 million triangles, 1,408 × 768 resolution, single GPU)

nodes are performed, as shown in Fig. 10. This way, short-stack performance is lower than the Standard traversal one.

The Push-Down and Short-Stack hybrid algorithm (PD & SS) [12] offers a significant gain of performance in comparison to the single short-stack, even with the addition of Push-Down's divergent control structures and stack operations. This happens due to the fact that Push-Down helps limiting the stack use by transferring the restart event to a scene sub-tree, reducing the number of nodes to be stored, which is convenient because of the small stack size.

The Ropes traversal algorithm proposed by Popov et al. [13] for CUDA architecture results in a lower number of visited nodes in comparison to all previously cited algorithms. In spite of increasing the amount of arithmetical operations, it is important to notice that, contrary to the other algorithms, the Ropes traversal does not need to perform costly division operations for every node traversal. Instead, it makes use of 3 less expensive multiply-add operations [4]. However, an additional ray-AABB intersection test is necessary for every visited leaf to define the new rope to be used. A great and inherent advantage of an approach using ropes refers to shadow and reflective rays, that start their traversal in the last intersected leaves, which contain the origin of such rays. However, an algorithm using ropes demands about 3 times more memory space (see Fig. 11) to store the scene, therefore being a prohibitive alternative for scenes with a high number of primitives (more than 1 million) in low-end CUDA enabled devices.

The algorithm with the second best performance result was the 8-byte Standard traversal. It allocates the stack in local memory, presenting a simple implementation with few control structures and consequently having a lower amount of divergences. The 8-byte layout also achieved a better performance than the 12-byte

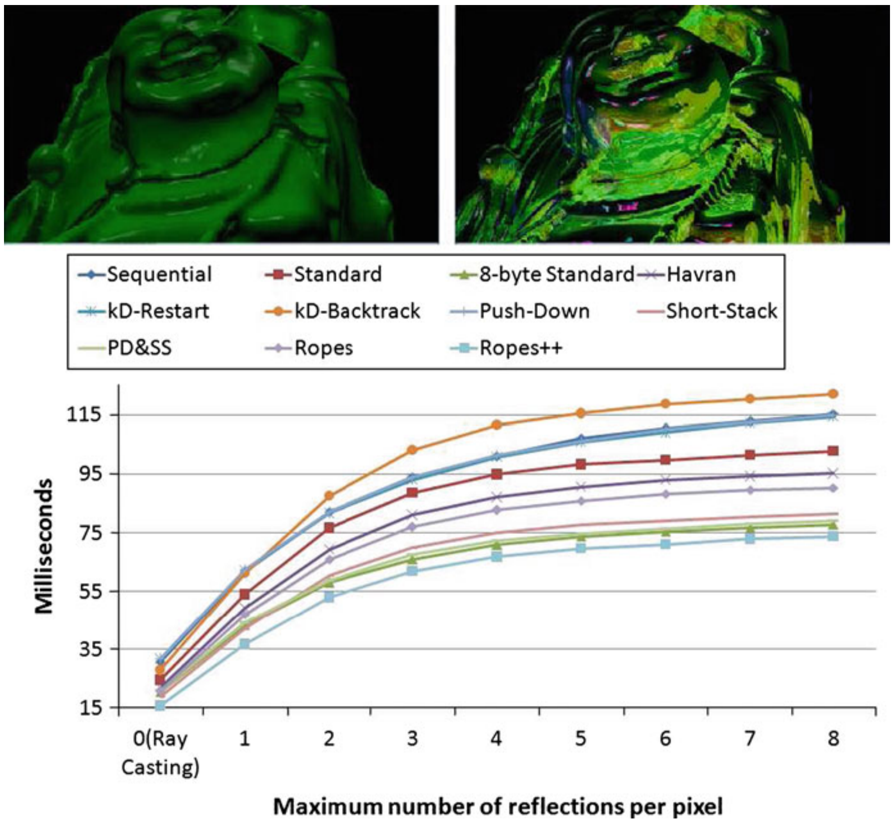

Fig. 13 Amount of time in milliseconds for each algorithm from a fixed viewpoint, varying the level of reflections and using the Buddha + 4 Spheres + 1 Plane scene (1.08 million triangles, 1,408 × 768 resolution, single GPU). Scenes on the *top-left* and *top-right* represent ray casting (no reflections) and 8 levels of reflection enabled, respectively

format, as shown in Figs. 13 and 14, which is explained by faster access from a 64-bit memory load than a 128-bit one. The Havran traversal algorithm achieved a performance between the 8-byte and 12-byte Standard traversals. This can be explained by the slight increase of traversal cost when compared to the standard approach.

As shown in Figs. 13 and 14, the Ropes++ traversal algorithm achieved the best performance results for almost all tested scenes. The reason for such performance enhancement is directly related to a lower number of visited nodes and the Ropes++ ray-AABB intersection optimizations, which significantly reduce the amount of global memory accesses. Another reason is the lower number of branches and divergent branches (see Fig. 12). Unfortunately, as the Ropes algorithm, the Ropes++ demands much more memory when compared to the Standard traversal, as shown in Fig. 11.

Figure 15 shows the achieved performance using multiple CUDA devices and for different image resolutions. Two different image splitting approaches were analyzed:

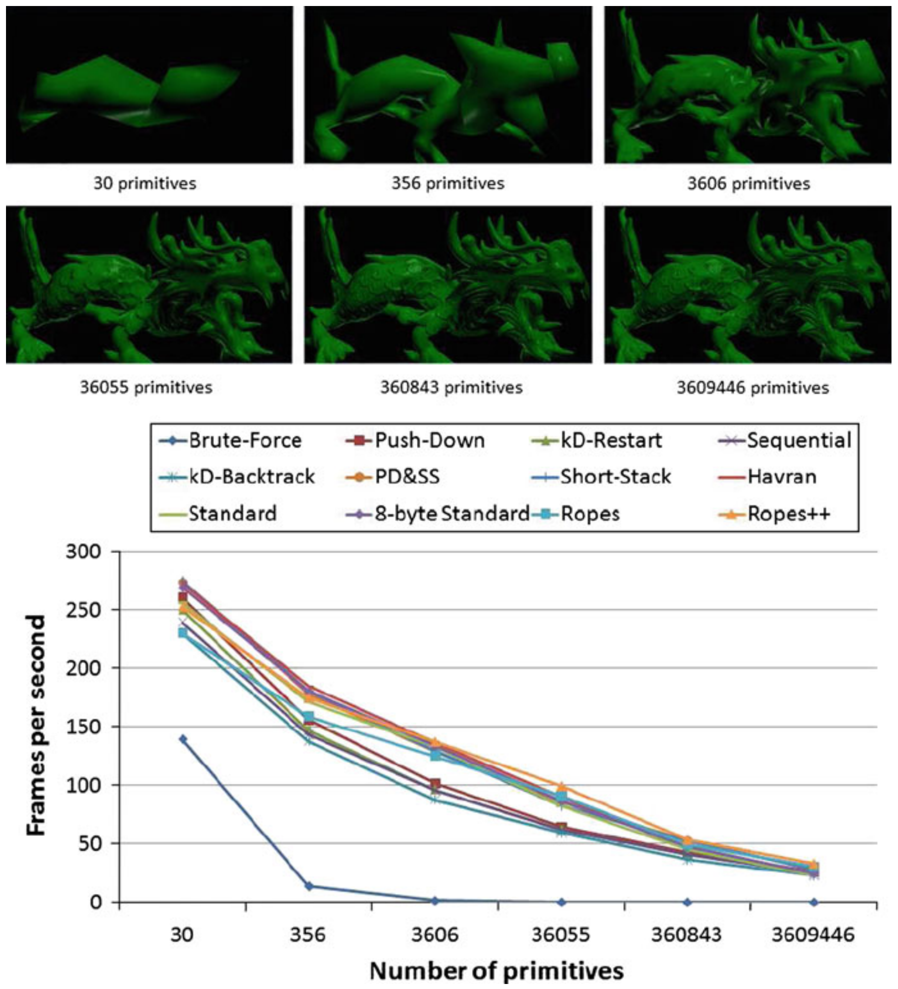

Fig. 14 Traversals' performance degradation according to the increase of the number of primitives (1,408 × 768 resolution, single GPU)

interleaved lines vs interleaved blocks. Using four devices (two GTX 295 cards), Fig. 15 shows that interlacing blocks leads to a better performance than interlacing lines, since the former presents a higher amount of ray coherence cases, as explained in Sect. 4.4.

Finally, the NVIDIA OPTIX CUDA ray tracing engine [5] and our Ropes++ traversal algorithm were compared regarding performance. Figure 16 shows that our traversal implementation reached speedup gains up to 3× when compared to NVIDIA OPTIX. It is important to notice that OPTIX supports kD-trees (Short-Stack traversal [5]) and BVHs. Using the OPTIX kD-tree, the Stanford Asian Dragon caused an out of memory exception, and therefore we were not able to compare this scene with our implementation.

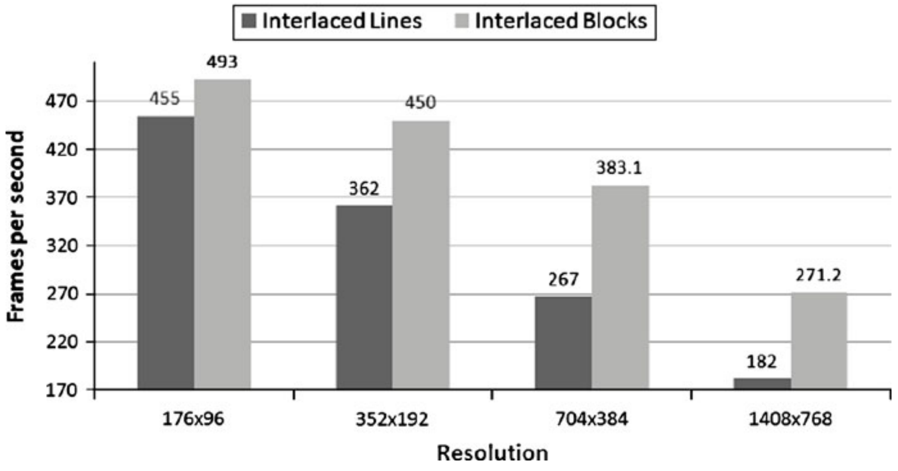

Fig. 15 Traversals' performance degradation according to the increase of resolution using the Bunny model (69k triangles, multi-GPU using 4 devices)

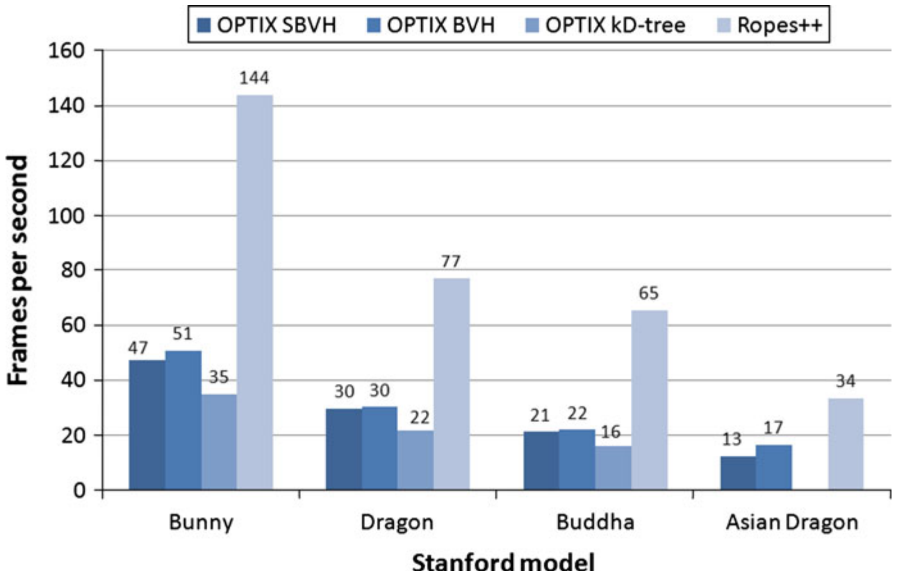

Fig. 16 Comparison between the proposed implementation against NVIDIA OPTIX ray tracing engine for 4 different Stanford models (1,408 × 768 resolution, single GPU)

6 Conclusion and Future Works

This work presented a compilation of different techniques applied to the development of ray tracers, focusing real time rendering. A considerable number of kD-tree traversal approaches were compared regarding performance, in order to obtain an algorithm capable of offering the lowest execution time. The best result was achieved using our proposed traversal approach (Ropes++), based on the technique of Havran et al. [17]. The second best algorithm was our modified Standard traversal, the 8-byte one. Although slightly slower than Ropes++, the 8-byte Standard traversal requires approximately three times less memory, since the Ropes++ needs to store six ropes for each leaf node.

CUDA architecture favors implementations that recently ran only efficiently on CPUs, due to their efficient data structures stored in primary memory. The Standard traversal, which uses stacks, was difficult to efficiently implement using shaders. It must be highlighted that all implemented techniques using CUDA architecture are capable of rendering high definition images in real time in a scalable way, which was not the case in the majority of the original implementations discussed in this work.

We also presented some results regarding multi-GPU processing. Our approach of interleaved blocks achieved better performance than interleaved lines. However, the scalability factor of this approach was below expected, since using 4 devices we only achieved a 2.3$\times$ speedup when compared to a single device. As future work, we intend to research better ways of balancing the tasks between multiple CUDA devices.

Another future work is to implement and compare different acceleration structures, such as BVH and BIH, regarding traversal performance and construction time for both of them.

Acknowledgments The authors would like to thank Jacco Bikker for sharing his kD-tree construction code, previously presented on Arauna ray tracer [23]. Many thanks to Brian Budge at UC Davis, for sharing his knowledge of ray-traversal optimizations on CUDA. The authors also would like to thank CNPq for the financial support received.

References

1. Appel, A.: Some techniques for shading machine renderings of solids. In: AFIPS ’68 (Spring): Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference, pp. 37–45. ACM, New York (1968)
2. Glassner, A. (ed.): An Introduction to Ray Tracing. Academic Press, London (1989)
3. Havran, V.: Heuristic ray shooting algorithms. Ph.D. dissertation, Czech Technical University, Praha, Czech Republic, Apr. 2001, available from http://www.cgg.cvut.cz/havran/phdthesis.html
4. NVIDIA, NVIDIA CUDA Programming Guide 3.0, 2010. [Online]. Available: http://www.nvidia.com/object/cuda
5. NVIDIA, Nvidia optix ray tracing engine, 2010. [Online]. Available: http://developer.nvidia.com/object/optix-home.html
6. Whitted, T.: An improved illumination model for shaded display. Commun. ACM **23**(6), 343–349 (1980)
7. Jansen, F.: Data structures for ray tracing. In: Kessener, L.R.A., Peters, F.J., van Lierop, M.L.P. (eds.) Data Structures for Raster Graphics, pp. 57–73. Springer, Berlin (1986)

8. Rubin, S.M., Whitted, T. (1980) A 3-dimensional representation for fast rendering of complex scenes. In: SIGGRAPH '80: Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques, pp. 110–116. ACM, New York
9. Wächter, C., Keller, A.: Instant ray tracing: The bounding interval hierarchy. In: Akenine-Möller, T., Heidrich, W. (eds.) Eurographics Workshop/ Symposium on Rendering, pp. 139–149. Eurographics Association, Nicosia, Cyprus (2006) [Online]. Available: http://www.eg.org/EG/DL/WS/EGWR/EGSR06/139-149.pdf
10. Wald, I., Slusallek, P., Benthin, C., Wagner, M.: Interactive rendering with coherent ray tracing. Comput. Graph. Forum **20**(3) (2001)
11. Boulos, S., Edwards, D., Lacewell, J.D., Kniss, J., Kautz, J., Wald, I., Shirley, P.: Packet-based whitted and distribution ray tracing. In Proceedings of Graphics Interface (2007)
12. Horn, D.R., Sugerman, J., Houston, M., Hanrahan, P.: Interactive k-d tree GPU raytracing. In: Gooch, B., Sloan, P.-P.J. (eds.) SI3D, pp. 167–174. ACM, London (2007) [Online]. Available: http://doi.acm.org/10.1145/1230100.1230129
13. Popov, S., Günther, J., Seidel, H.-P., Slusallek, P.: Stackless KD-tree traversal for high performance GPU ray tracing. Comput. Graph. Forum **26**(3), 415–424 (2007) [Online]. Available: http://dx.doi.org/10.1111/j.1467-8659.2007.01064.x
14. Günther, J., Popov, S., Seidel, H.-P., Slusallek, P.: Realtime ray tracing on GPU with BVH-based packet traversal. In: Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007, pp. 113–118 (2007)
15. Kaplan, M.: Space-tracing: a constant time ray-tracer. In: SIGGRAPH '85: Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques. ACM, New York (1985)
16. Foley, T., Sugerman, J.: KD-tree acceleration structures for a GPU raytracer. In: Meißner, M., Schneider, B.-O. (eds.) Graphics Hardware, pp. 15–22. Eurographics Association, Los Angeles (2005). [Online]. Available: http://www.eg.org/EG/DL/WS/EGGH/EGGH05/015-022.pdf
17. Havran, V., Bittner, J., Sára, J.: Ray tracing with rope trees. In: Kalos, L.S. (ed.) 14th Spring Conference on Computer Graphics, pp. 130–140. Comenius University, Bratislava (1998)
18. Aila, T., Laine, S.: Understanding the efficiency of ray traversal on gpus. In: HPG '09: Proceedings of the Conference on High Performance Graphics 2009, pp. 145–149. ACM, New York (2009)
19. Harris, M., Harris, M.: Parallel prefix sum (scan) with cuda (2007). [Online]. Available: http://beowulf.lcs.mit.edu/18.337/lectslides/scan.pdf
20. Wald, I., Havran, V.: On building fast kd-trees for ray tracing, and on doing that in O(N log N). In: Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, pp. 61–69 (2006)
21. Brodal, G.S., Fagerberg, R., Jacob, R.: Cache oblivious search trees via binary trees of small height. In: SODA '02: Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 39–48. Society for Industrial and Applied Mathematics, Philadelphia (2002)
22. The stanford 3d scanning repository. [Online]. Available: http://graphics.stanford.edu/data/3Dscanrep/
23. Bikker, J.: Arauna real time ray tracing (2009). [Online]. Available: http://igad.nhtv.nl/~bikker/