Ray Reordering Techniques for GPU Ray-Cast Ambient Occlusion

Vasco Costa João M. Pereira Joaquim A. Jorge
Department of Information Systems and Computer Science
INESC-ID/Instituto Superior Técnico, University of Lisbon
Rua Alves Redol, 9, 1000-029 Lisboa, Portugal
vasco.costa@ist.utl.pt, jap@inesc-id.pt, jaj@inesc-id.pt

Abstract

Global illumination techniques, such as ambient occlusion, can be performed in a physically accurate way via ray casting. However ambient occlusion rays are incoherent. This means their computation is divergent causing a degradation of rendering performance. This problem is particularly acute on the GPU stream computing architectures which have performance issues with thread divergence. We reorder the rays, prior to the rendering step, to reduce this thread divergence issue. Rays which traverse the same region of space are reordered in bundles in order to increase memory coherency. We demonstrate that ray reordering techniques enhance performance while rendering scenes with ambient occlusion rays. The question is how to best perform this ray reordering. Ray reordering for ambient occlusion requires the classification of millions of rays. Spending too much time reordering these rays can negate any rendering performance benefits. Our work surveys and tests several techniques for ray reordering. We achieved the best performance results using a compress-sort-decompress technique, which sorts hashed rays, where the hash key has 32 bits of size.

Keywords

ray casting,ray reordering,gpu,ambient occlusion

1. INTRODUCTION

The availability of high performance hardware, namely GPUs, enables the use of more realistic rendering schemes. However GPUs have some limitations. These hardware architectures feature small sized caches and have simplified branch prediction hardware. Thus they are less tolerant of algorithms which feature divergent program paths or that have poor memory coherency. Unfortunately this is the case for several global illumination rendering algorithms.

In our case we are interested in performing ray-cast ambient occlusion. This technique is used, for example, to generate the baked shadow textures used in computer games. Other global illumination techniques have the same issues with lack of coherency and branch divergence. The processing of secondary rays while performing distributed ray tracing [Cook 84] is one such example.

In order to mitigate coherency issues while performing ray-casting we reorder the rays. Rays which traverse the same region of space are processed together minimizing branch divergence and improving memory coherency. This way we can improve the overall rendering performance. This work examines the performance of several techniques in regards to the ray-casting of ambient occlusion rays.

We can determine the ambient occlusion term, for a given point in the surface of an object, by ray-casting N random ray samples (see Figure 1) across the hemisphere centered around that point p oriented towards the same direction as the surface normal at that same point. These random ray samples can be generated with the aid of an R^2 Halton quasi-random number sequence.

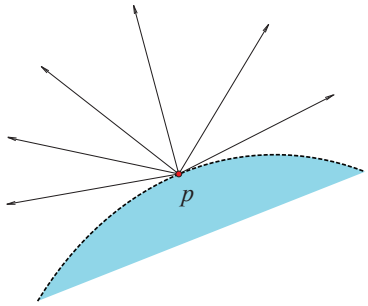

Figure 1. Ambient occlusion sampling.

A simple and expedient way to order these rays would be to group together rays with the same origin point. However as can be easily understood this does not guarantee the rays will not diverge significantly further along their path. Hence we need to take into account the ray directions as well while performing ray reordering.

To determine the ambient occlusion term for a 1024×1024 image with 16 samples per pixel we need to process over 16 million rays. Performing reordering with sorting on

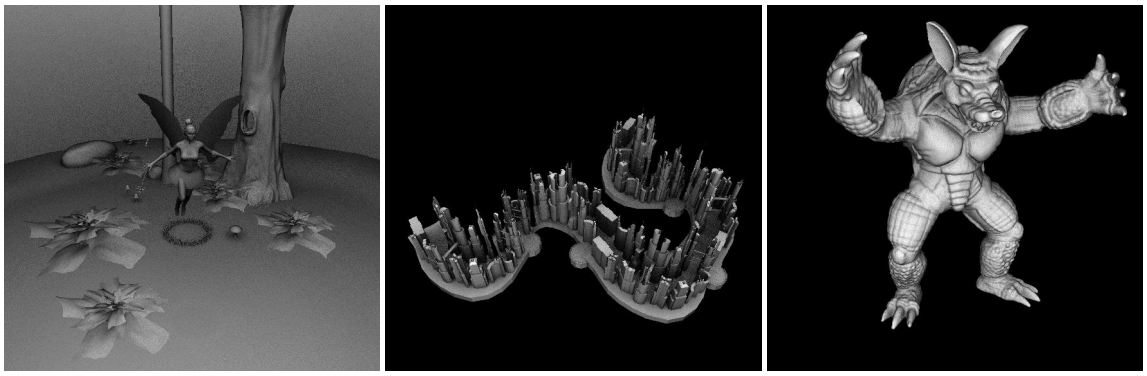

Figure 2. Fairy Forest, Serpentine City, Armadillo test scenes with ambient occlusion.

such huge lists, even in a GPU platform, can take tens or hundreds of milliseconds depending on the sorting algorithm and how accurate we want the ray sorting to be done.

Rays have five degrees of freedom. Three degrees of freedom for the origin point. Two degrees of freedom for the direction normal using spherical coordinates. If we represent each of these degrees of freedom with a 32-bit number then sorting these rays at full precision would at best require sorting large 160-bit keys. Since we are doing the sort operation just to maximize coherency it is possible to sort these rays at less than full precision using an hashing scheme. Thus speeding up the sorting process. To increase sorting speed even further it is possible to employ a compress-sort-decompress scheme, where adjacent rays with the same hash values are bundled together and sorted as a single unit, decreasing the size of the list which needs to be sorted.

Our contributions include the evaluation of ray reordering techniques in the context of GPU ray-casting. We have measured a 1.73x speedup on the rendering of scenes, with ambient occlusion rays, in our tests when using these techniques. The ray reordering techniques discussed in this work are independent of the ray tracing acceleration structure but should provide more of a performance boost in acceleration structures which degrade less gracefully as the ray coherence decreases.

The organization of this paper is as follows: we survey previous related work in detail, then we describe ray reordering techniques suitable for stream computing architectures. Next, we describe the rendering pipeline, the testing methods we adopted and present performance figures. Finally we discuss our results and present ideas for future work.

2. RELATED WORK

Arvo and Kirk [Arvo 87] described a ray tracing acceleration scheme employing *ray classification*. In this scheme a scene is partitioned in 5D ray space. To determine the primitives intersected by a ray using this scheme we simply consult the primitives list of the subspace which contains that ray.

Moon et al [Moon 10] improved ray tracing performance on huge out-of-core scenes by reordering rays according to their hit point location. The technique computes an approximation of the hit point of each ray, i.e. the hit point heuristic, against a decimated mesh which has a quarter of the amount of triangles in the original mesh. This decimated mesh can fit into main memory unlike the original mesh at full resolution. These tentative hit points are then reordered in Z-curve order. Since this technique requires the construction of a decimated mesh it is hard to integrate with existing real-time or interactive applications.

Boulos et al [Boulos 08] used a breadth-first bounding volume hierarchy (BVH) ray packet traversal scheme. Multiple rays can be processed at a time in order to make use of SIMD parallelism. Their algorithm reorders rays when the packet utilization, i.e. the amount of active rays in a packet, drops below a certain threshold. Their algorithm also reorders shading requests by grouping together rays which intersect the same materials.

Barringer and Möller [Barringer 14] traverse a BVH using a stream based approach where the traversal loop also can process multiple rays at a time. Rays are reordered according to their directions prior to processing by sorting them into eight bins. One bin for each possible x, y, z ray sign direction.

Both of the previous techniques are limited in that that they are restricted to applications which use a bounding volume hierarchy ray tracing acceleration structure. While these approaches can be adapted for the traversal of other tree based acceleration structures, namely kd-trees, they do not map well to the traversal of grids and other non-tree based acceleration structures. The focus of our work is on acceleration structure *agnostic* ray reordering. These particular techniques are also highly tuned to the specific hardware configuration. They require extensive recoding work in order to map onto other compute architectures such as GPUs.

Aila and Laine [Aila 09] reorder secondary rays using 192-bit keys in the CPU in order to improve memory coherency and minimize divergence while performing the rendering on the GPU.

Hoberock et al [Hoberock 09] showed how to reorder shading operations in order to avoid divergence in GPU archi-

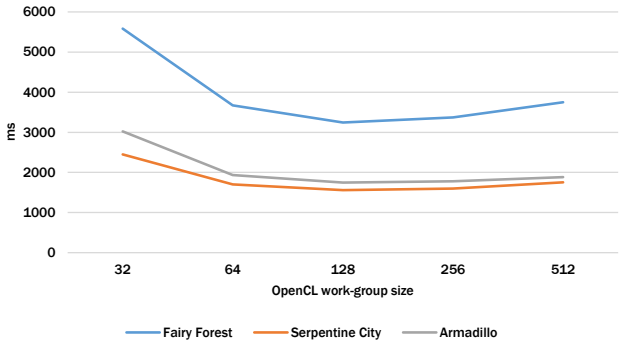

Figure 3. Time required to render the test scenes depending on the work group size.

tectures. The shading operation reordering is done with the aid of prefix sum and radix sort operations.

Garanzha and Loop [Garanzha 10] describe a breadth-first packet traversal algorithm for ray tracing which employs fast ray sorting using a compress-sort-decompress scheme to generate the ray packet bundles.

Blelloch [Blelloch 90] describes uses for the prefix sum operation in parallel architectures. We use prefix sum in our work in order to perform the compression and decompression steps. Harris et al [Harris 07] describe how to implement the prefix sum operation efficiently on GPU architectures.

Efficient GPU parallel sorting algorithms include bitonic sort [Lang 10] and radix sort [Satish 09]. The bitonic sort algorithm implementation we used has $O(N \log^2 N)$ time complexity. Radix sort implementations have $O(kN)$ complexity where k is the key size and N is the number of elements in the list to be sorted. Bitonic sort has the advantage that it can sort any items which have a comparison operator whether they are numbers or not. Radix sort is faster while sorting large lists of numbers with small key sizes.

Our work tests ray reordering techniques for ambient occlusion rays with different key sizes. From the full precision 192-bit keys to the hashed 32-bit keys. We also examine the benefit of the use of the compress-sort-decompress technique.

3. RAY REORDERING TECHNIQUES

In order to examine the performance advantage of ray reordering techniques which employ ray sorting we established a baseline set of techniques which do not employ ray sorting. Ray reordering in that case is only done with the use of the OpenCL [Munshi 11] work group construct which states which and how many compute threads should be executed in the same stream computing unit.

We devised a taxonomy for ray reordering techniques which is as follows: *static ray reordering* techniques apply a strictly fixed ray ordering scheme using the work group construct; *dynamic ray reordering* techniques sort rays according to their origin and direction values.

The ray reordering techniques we use are described in detail in the next subsections.

3.1. Static

We use two different static ray reordering techniques. These are named according to their work group arity.

The *static 1D* technique uses a 1D work group size. The optimum work group size for running a compute kernel is dependent on the characteristics of the hardware used and how many machine registers the kernel uses. In order to determine the optimum size of the work group for our hardware we conducted tests where we ran the ambient occlusion term computation algorithm with different work group sizes. As can be seen in Figure 3 the optimum work group size is 128 for all the test scenes as this provides the smallest ambient occlusion computation time. Rays with the same origin are processed together.

The *static 3D* technique uses the notion that rays with similar directions should be processed at the same time in order to maximize performance. This ray reordering technique does not require any ray sorting to be done either because the ray directions are pseudo-randomly generated using the same Halton sequence. We used a work group size of $8 \times 16 \times 1$ for a set of ambient occlusion rays of dimensions $width \times height \times nsamples$. This technique does not order the rays perfectly, since the pseudo-random generator does not always generate rays in the same overall direction for each nth sample as the object surface varies, but in practice this provides better results than just grouping together rays with the same origin as we did in the previous technique.

Both of these static techniques should also be applicable to shadow rays. The dynamic ray reordering techniques which use sorting should be much better suited for rendering reflection and refraction rays. We describe these dynamic techniques in the next subsection.

3.2. Dynamic

In the *dynamic 32-bit* technique [Garanzha 10] the 5D rays are hashed into 32-bit unsigned integer keys. These keys are then subsequently sorted. This reduces the amount of time spent doing comparisons. Since sort precision is reduced with this hashing scheme the sorted rays will be less coherent than using an exact ray sorting scheme. The HASHKEY32 function maps the 3D origin coordinates into the lower 24-bits of the key and the 2D spherical coordinates of the ray direction into the upper 8-bits of the key as can be seen in Algorithm 1.

The *dynamic 192-bit* technique [Aila 09] sorts the rays at a greater precision. The floating point ray coordinates are converted into a 192-bit key which is then sorted. The HASHKEY192 function first maps the ray origin and direction into the range $[0, 1]$, then scales the 3D origin coordinates into 24-bits each, the 3D ray direction coordinates into 21-bits each, and finally it interleaves the bits of each component with COLLECTBITS into a 6D array of 32-bit unsigned integer values. As we shall see later this tech-

Algorithm 1 Ray hashing functions.

```
function TRANSLATE(aabb, point)
    return ← (point − aabb_max) / (aabb_max − aabb_min)
end function
function HASHKEY32(aabb_rays, origin, direction)
    o_{x,y,z} ← TRANSLATE(aabb_rays, origin) × 256          ▷ 8 bits × 3
                ▷ translates ray origin into ray bounding box coordinates [0,1]
    θ ← ACOS(direction_z)
    φ ← ATAN2(direction_y, direction_x)
    d_x, d_y ← (φ+π)/2π × 8, θ/π × 8                        ▷ 4 bits × 2
    return ← HASH3(o_x, o_y, o_z) ∨ (HASH2(d_x, d_y) ≪ 24)
end function
function COLLECTBITS(hash, idx, x)
    for all i ∈ 0 . . . 31 do
        k ← (idx + i × 6) ≫ 5
        hash[k] ← hash[k] ∨ ((x ≫ i) ∧ 1) ≪ ((idx + i × 6) ∧ 31)
    end for
end function
function HASHKEY192(aabb_rays, origin, direction)
    o_{x,y,z} ← TRANSLATE(aabb_rays, origin)
    d_xy ← (direction + 1.0) × 0.5
    hash[0 . . . 5] ← 0
    COLLECTBITS(hash, 0, o_x × 256.0 × 65536.0)
    COLLECTBITS(hash, 1, o_y × 256.0 × 65536.0)
    COLLECTBITS(hash, 2, o_z × 256.0 × 65536.0)
    COLLECTBITS(hash, 3, d_x × 32.0 × 65536.0)
    COLLECTBITS(hash, 4, d_y × 32.0 × 65536.0)
    COLLECTBITS(hash, 5, d_z × 32.0 × 65536.0)
    return ← hash
end function
```

nique gives better rendering performance at a higher sort time cost.

3.3. Compress-Sort-Decompress

The compress-sort-decompress technique [Garanzha 10] compresses the rays using run-length-encoding. Adjacent rays with the same hash values are grouped together. This reduces the amount of elements in the list to be sorted. Afterwards compressed ray bundles are decompressed into a final sorted list. This technique uses 32-bit ray hash keys. If there is too much variance between the ray keys the run-length-encoding scheme will not compress the rays well.

Prior to running the compress-sort-decompress algorithm ray hashes are computed using HASHKEY32. The algorithm works in the following order: compression via run-length-encoding (RLE), sorting of the compressed rays, decompression of the compressed rays. The overhead of the compression and decompression steps is minor as we can see in Figure 4. Once again this algorithm reduces the time spent to sort the rays since the total number of RLE blocks to sort is smaller than the total number of rays.

3.3.1. Compression

This is the run-length-encoding step. Here the $header$ array is computed. Each array element is an RLE block which states the $start$ position of the first ray of the block in coordinates of the initial array of rays, the $size$ which stores the number of adjacent rays with an identical hash, as well as the actual $hash$ value for that ray block.

The $header$ array is computed as follows. We create an array H with size equal to the number of rays to process. $H[i]$ is initialized as 1 when the ray at position i differs from the previous ray at $i-1$ and is initialized as 0 in other cases. We compute the exclusive prefix sum of H. This initializes H with the $start$ positions for each RLE block. It also computes the total amount of blocks. We can now assign the $start, size, hash$ values to each block where $size$ at position i is computed via $H[i+1] - H[i]$. The blocks have now been computed and are ready for the sort step.

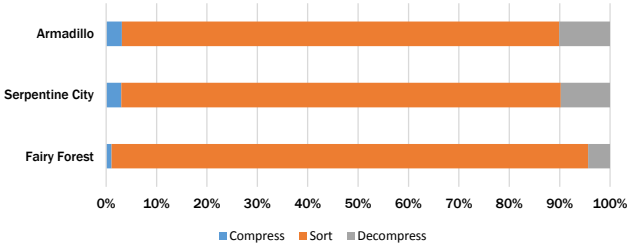

Figure 4. Percentage of total ray reordering time spent on the compress, sort, decompress stages for each test scene.

3.3.2. Sorting

In this step the RLE blocks are sorted by their $hash$ values. Our implementation uses a bitonic sort of $O(N \log^2 N)$ time complexity. Sorting tens of millions of 32-bit key value pairs on our system, with this algorithm, takes hundreds of milliseconds. We also tested the radix sort algorithm with $O(kN)$ linear time complexity. Radix sort takes tens of milliseconds on the same system to sort the same number of key value pairs.

3.3.3. Decompression

This is the final decompression step which returns a sorted list of rays given the intermediate list of sorted RLE blocks. We decompress these RLE blocks by first building an array with the sorted positions for each ray and secondly copying rays from their initial positions in the ray list to their final positions in the sorted ray list.

We compute the sorted positions array as follows. We create an array O with size equal to the number of RLE blocks to process. $O[i]$ is initialized with the $size$ of the block at position i. We compute the exclusive prefix sum of O in order to determine the start positions, in the final ray list, for each block. Then we create an H array which will contain the final ray positions. H has the same size as the number of rays we are processing with all its elements initialized as ∞. We compute the ray positions by initializing the ray positions array H thus. $H[O[i]]$ is initialized with the $start$ ray position of a block. We do this for all the RLE blocks. Then we compute the inclusive segmented prefix sum of H which will generate the final offsets. The segmented prefix sum works in intervals, delimited by the non-∞ elements, where the ∞ elements are treated as if they had a value of 1.

Thus we generate the final ray positions in consecutive order for each block. Once we finish copying the rays from the original list to the sorted list we can render the scene.

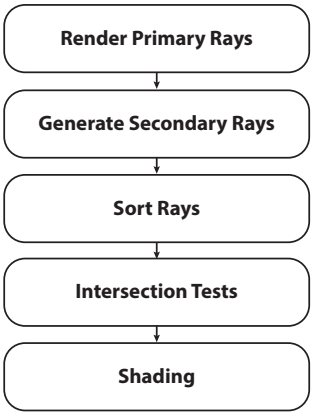

Figure 5. Rendering Pipeline.

4. RENDERING PIPELINE

The traditional ray tracing rendering pipeline works on what is known as a megakernel approach [Laine 13]. It uses an expensive, typically recursive, function aka kernel which computes the color contribution at a given pixel. This leads to an unbalanced rendering load since not all pixels require the same computation time.

Factors which can influence the computation time include: the time to traverse the ray tracing acceleration structure, the time to test the polygons intersected along the ray path, and the time to shade the materials. If the materials are elaborate the shading may take more time to compute than the time required for the polygon intersections task. In addition megakernels increase register pressure which, on a SIMT architecture such as a GPU, leads to problems when trying to issue enough concurrent threads to fully utilize available compute resources.

In order to mitigate the load balancing issues of a megakernel approach we devised a rendering pipeline which splits the rendering computation into several pipeline stages, as can be seen in Figure 5, at the cost of some additional temporary memory, for storing all the rays, in each of the intermediate stages. This allows us to optimize the load balancing at each of the processing steps required to compute the rendered image in a more fine-grained way leading to improved performance versus a megakernel. In the case of secondary rays we are also interested in applying ray reordering in order to improve coherency. This is not something which can be done using a traditional megakernel approach since, in that case, it is not possible to stop the computation midway to globally reorder all the rays. Each thread executes in a concurrent fashion and any two threads may be in completely different rendering stages at the same time.

To have acceptable rendering quality, in a modern interactive application, we require physically accurate shadows and ambient occlusion term computation. Reflections and refractions are more expensive to compute with more complicated memory access patterns. Their applicability is also much less than the previous global illumination effects in typical scenes so we did not consider these in our current rendering pipeline in order to simplify it and further enhance performance. For example since we do not have refractions we can apply the back-face culling technique.

Our rendering pipeline works by first generating the primary rays, also known as eye rays, traversing the acceleration structure and computing the nearest intersection for each of the primary rays. Then we generate the secondary rays, which in the case of this work consist of ambient occlusion rays. The secondary rays are sorted according to their spatial distribution, where rays that traverse the same region of space are grouped together, and intersection tests are done in order to compute the ambient occlusion term. Finally we compute the shading for all the pixels in the screen taking into account the material, normal, uv coordinates, ambient occlusion term, at each pixel.

Our application supports texture mapping but this was not used for the test in this work since our main concern is profiling the performance of each of the ray reordering techniques and texture mapping would further unbalance the rendering load thus complicating the results analysis.

5. TESTING METHODOLOGY

The test platform uses an AMD FX 8350 8-core CPU @ 4.0 GHz powered machine with 8 GB of RAM. The graphics card includes a NVIDIA GeForce GTX TITAN GPU with 6 GB of RAM. The performance of the CPU is irrelevant, in our case, since all rendering and ray reordering algorithms run on the GPU. The algorithms were implemented in the OpenCL programming language.

All test images were rendered at 1024×1024 resolution with one primary ray per pixel using dot-normal (i.e. Lambertian) shading and sixteen ambient occlusion samples per primary ray. Thus rendering an image requires ray-casting 17.83 million rays.

We selected three test scenes (Serpentine City, Fairy Forest, Armadillo) representative of typical 3D gaming applications which can be seen in Figure 2.

Ray-triangle intersection adopts the Möller-Trumbore [Möller 97] algorithm since it does not have additional memory or precomputation costs. Each triangle needs 36 bytes of memory to store vertexes. Triangles with normals require an additional 36 bytes of memory. This is done to ensure more coherent memory accesses than an indexed vertex scheme would be able to provide. In our experience storing the triangles this way improves the rendering performance by 10% over the indexed method.

Sorting is generally done using the $O(N \log^2 N)$ bitonic sort algorithm. In the cases where we used the $O(kN)$ radix sort this is stated in the text.

6. RESULTS DISCUSSION

The speedup obtained with ray reordering techniques increases as the amount of incoherent rays as part of the total work load increases. As can be seen on the left of Figure 6 the Fairy Forest scene benefited the most from ray reordering. This is because the camera parameters we

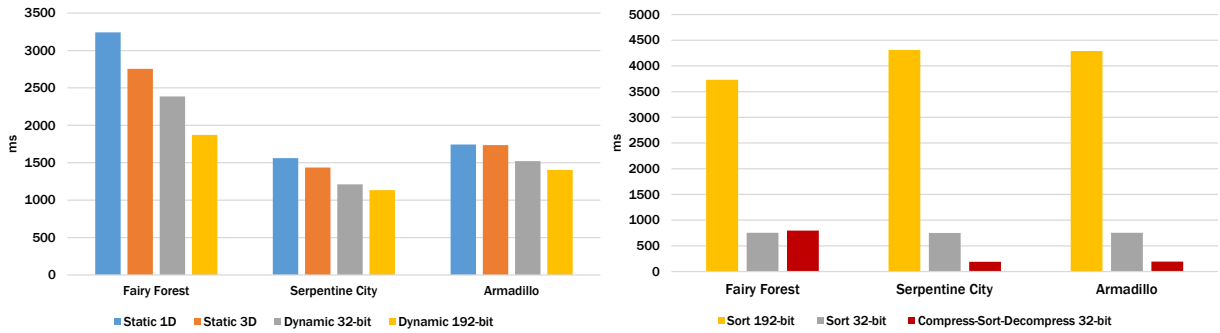

Figure 6. The chart on the left displays the time to render the test scenes for each of the ray ordering methods. The chart on the right displays the time required to reorder the rays for the dynamic reordering methods. The static reordering methods do not require any additional ray reordering time.

used for that scene require more ambient occlusion rays to be traced. In the Fairy Forest scene geometry covers the whole screen space whereas for the Serpentine City and Armadillo scenes a lot of the screen space is an empty background. Ina addition Serpentine City and Armadillo are manifolds so the computation of the ambient occlusion term is much more cache coherent than for the Fairy Forest scene which is more representative of a typical game scene.

As expected the more accurate hashing schemes have better rendering performance. The static techniques, which use a fixed ordering scheme, lead to worse rendering times than the dynamic techniques which compute actual ray hash keys and group together those rays which traverse the same regions of space.

On the right of Figure 6 we can see that the dynamic 192-bit key technique takes a long time to sort the ambient occlusion rays. In fact it takes longer to sort the 192-bit ray keys, using the bitonic sort algorithm, than to render the whole scene using the simplest static technique. When we reduce the key size to a sixth of 192-bits i.e. 32-bits the time required to sort the rays is reduced by a similar amount. The total amount of memory operations and comparisons is much less. If we use radix sort instead of bitonic sort the sort times are reduced from hundreds of milliseconds to tens of milliseconds thus making the dynamic ray reordering methods the most viable choice to improve ambient occlusion ray rendering performance.

The compress-sort-decompress scheme further improves sort performance for the Serpentine City and Armadillo scenes. This is due to it grouping together the background rays which take less time to render. In other words by simultaneously processing bundles of rays which take more or less the same amount of time to finish we are reducing the amount of bubbles in the GPU workload thus improving performance. This is an important fact which should not be underestimated as this can also be applied to other rendering tasks such as shading operations where not all materials have the same computational complexity level.

The performance gap between the 192-bit and 32-bit key dynamic techniques is large enough to see that there is still more work to be done with the hash functions. One thing which could also be improved is the way the ray origins are hashed. The current hashing algorithm uses a virtual grid with the same resolution in all x, y, z axis but the scene geometry is seldom uniformly distributed across all axis. The virtual grid should therefore be able to better adapt to the distribution of geometry in the scene. Instead of using this virtual grid for the ambient occlusion rays we could use the acceleration structure (e.g. bvh, kd-tree, grid) cell id which contains the ray origin instead as this is compact and the acceleration structure cell id has previously been computed when doing the prior rendering pass for the primary rays.

	Serpentine City (138.63 KTriangles)	Fairy Forest (173.98, KTriangles)	Armadillo (345.94 KTriangles)
Render Time			
static			
1D	1561 ms	3242 ms	1744 ms
3D	1436 ms	2757 ms	1736 ms
dynamic			
32-bit	1212 ms	2387 ms	1522 ms
192-bit	1133 ms	1873 ms	1403 ms
Reorder Time			
dynamic			
sort 192-bit	4310 ms	3732 ms	4289 ms
sort 32-bit	752 ms	754 ms	753 ms
c.s.d. 32-bit	189 ms	798 ms	192 ms

Table 1. Performance results of the techniques for the test scenes. The static techniques do not require a reordering step prior to rendering. The dynamic techniques perform ray reordering with bitonic sorting.

As can be seen in Figure 4 most of the time that is spent with the compress-sort-decompress technique is sort time. Despite neighboring rays with the same hash being grouped together. When the hash function has more collisions rays are compressed better resulting in less sort time however hash functions with more collisions are more inaccurate and result in worse ambient occlusion computation time. It is necessary to strike a fine balance between the sort time and the render time to provide the best overall performance.

7. CONCLUSIONS AND FUTURE WORK

In this work we demonstrated that ray reordering speeds up the rendering of incoherent rays on stream computing architectures. We have measured a 1.73x speedup while rendering scenes with a lot of incoherent rays where we had 16 ambient occlusion samples per pixel. Instead of sorting the rays themselves it is much faster and more effective to sort the 32-bit hashes of the rays reducing the time spent on sorting. For interactive applications it is best to use a sorting algorithm such as radix sort since bitonic sort does not have enough performance to render scenes with tens of millions of rays at interactive frame rates.

There is still a lot of work to be done regarding ray hash functions in order to improve ray reordering performance. There is a large rendering performance gap between the 32-bit and 192-bit hash functions which we intend to explore in the future by experimenting with different key sizes and hash functions.

Since ray reordering is a task scheduling technique we should in the future not only take into account memory coherency, where neighboring rays are coalesced, but we should also take into account the ray intersection times into the hash function in order to guarantee better performance.

8. ACKNOWLEDGEMENTS

We thank the NVIDIA Corporation for the donation of the GeForce GTX Titan used for this research.

This work was supported by national funds through FCT - Fundação para a Ciência e Tecnologia, under project PEst-OE/EEI/LA0021/2013.

We would like to thank the Utah Animation Repository (Fairy Forest), Herminio Nieves (Serpentine City), and the Stanford 3D Scanning Repository (Armadillo) for the test scenes.

9. REFERENCES

[Aila 09] Timo Aila and Samuli Laine. Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 145–149, 2009.

[Arvo 87] James Arvo and David Kirk. Fast Ray Tracing by Ray Classification. *ACM SIGGRAPH Computer Graphics*, 21(4):55–64, 1987.

[Barringer 14] Rasmus Barringer and Tomas Akenine-Möller. Dynamic Ray Stream Traversal. *ACM Transactions on Graphics (TOG)*, 33(4), 2014.

[Blelloch 90] Guy E Blelloch. Prefix Sums and Their Applications. Technical report, Carnegie Mellon University, 1990.

[Boulos 08] Solomon Boulos, Ingo Wald, and Carsten Benthin. Adaptive Ray Packet Reordering. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing 2008*, pages 131–138, 2008.

[Cook 84] Robert L Cook, Thomas Porter, and Loren Carpenter. Distributed Ray Tracing. *ACM SIGGRAPH Computer Graphics*, 18(3):137–145, 1984.

[Garanzha 10] Kirill Garanzha and Charles Loop. Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. *Computer Graphics Forum*, 29(2):289–298, 2010.

[Harris 07] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel Prefix Sum (Scan) with CUDA. *GPU gems*, 3(39):851–876, 2007.

[Hoberock 09] Jared Hoberock, Victor Lu, Yuntao Jia, and John C Hart. Stream Compaction for Deferred Shading. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 173–180, 2009.

[Laine 13] Samuli Laine, Tero Karras, and Timo Aila. Megakernels Considered Harmful: Wavefront Path Tracing on GPUs. In *Proceedings of the Conference on High Performance Graphics 2013*, pages 137–143, 2013.

[Lang 10] HW Lang and FH Flensburg. Bitonic sort. http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm, May 2010.

[Möller 97] T. Möller and B. Trumbore. Fast, Minimum Storage Ray-Triangle Intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997.

[Moon 10] Bochang Moon, Yongyoung Byun, Tae-Joon Kim, Pio Claudio, Hye-Sun Kim, Yun-Ji Ban, Seung Woo Nam, and Sung-Eui Yoon. Cache-oblivious ray reordering. *ACM Transactions on Graphics (TOG)*, 29(3):28, 2010.

[Munshi 11] Aaftab Munshi, Benedict Gaster, Timothy G Mattson, and Dan Ginsburg. *OpenCL Programming Guide*. Pearson Education, 2011.

[Satish 09] Nadathur Satish, Mark Harris, and Michael Garland. Designing Efficient Sorting Algorithms for Manycore GPUs. In *Proceedings of IEEE International Symposium on Parallel & Distributed Processing 2009*, pages 1–10, 2009.