Compressed Grids for GPU Ray Tracing of Large Models

Vasco Costa
INESC-ID/IST
Rua Alves Redol, 9
1000-029 Lisboa,
Portugal
vasco.costa@ist.utl.pt

João M. Pereira
INESC-ID/IST
Rua Alves Redol, 9
1000-029 Lisboa,
Portugal
jap@inesc-id.pt

Joaquim A. Jorge
INESC-ID/IST
Rua Alves Redol, 9
1000-029 Lisboa,
Portugal
jaj@inesc-id.pt

ABSTRACT

Ray tracing on GPUs is an area of ongoing research. GPUs are well suited for this parallel rendering algorithm. GPU computing devices typically have characteristics which make them quite different from CPUs: increased data parallelism, increased memory bandwidth, smaller caches, lower memory capacity. Presently it is difficult to visualize large scenes with tens of millions of triangles in these memory constrained platforms. In this paper we present a compressed grid data structure, capable of state of the art rendering performance, using up to $6\times$ less memory than conventional grid storage schemes. The compressed grid is built and traversed on the GPU.

Keywords

Ray-tracing, gpu, grid, compression.

1 INTRODUCTION

Display devices have been increasing in resolution at a more rapid pace than in the past. This means scenes with low polygon counts are no longer suitable as users can perceive the large polygons therein thus reducing their level of immersion. In addition the ray tracing rendering algorithm has been gathering increased attention. It is possible to ray trace complex scenes at real-time frame rates on a single GPU of the latest generation.

The ray tracing algorithm is more amenable for photorealism as it is a global illumination algorithm which can easily display shadows, reflections, or refractions. It is possible to extend it for diffuse interreflections as well using path tracing, or photon mapping albeit at much reduced frame rates. In this paper we focus on solving the basic ray shooting algorithm which is used for all these cases.

To provide real-time ray tracing performance an acceleration structure must be employed in order to reduce the number of ray/polygon intersection tests required to render the scene.

Existing work for GPU ray tracing includes [LGS+09, PL10] which focuses on bounding volume hierarchy (BVH) acceleration structures, [HSHH07] which describes kd-tree acceleration structures, and [KBS11] which focuses on grid acceleration structures. Other interesting developments on CPU ray tracing include: [Áfr12] which implicitly stores a BVH acceleration structure with reduced space requirements (still it uses temporary storage for bounding boxes and other auxiliary data), [LD08] which employs row displacement compression to reduce grid memory storage requirements.

Figure 1: The Lucy model (28 Mtri) rendered at 1024×1024 resolution. The grid acceleration structure is compressed to 210.45 MB. An uncompressed uniform grid of the same dimensions uses 1258.48 MB of memory.

Our work implements row displacement compressed grids in streaming computing architectures such as GPUs.

The organization of this paper is as follows: we describe a novel parallel algorithm for construction of row displacement compressed grids on the GPU, next we analyze the algorithm's performance compared to state of the art grid implementations.

2 GRIDS

Uniform grids are spatial partitioning structures which divide space into identical cubically shaped cells also named voxels in the literature. Typically a grid construction algorithm first computes the scene bounding box then uses an heuristic to compute the number of grid split planes along each major axis of the scene bounding box. These heuristics commonly attempt to use an amount of space directly proportional to the number of primitives in the scene. Thus we arrive at the following heuristic common in grid literature:

$$M_i = S_i \sqrt[3]{\frac{\rho N}{V}} \quad i \in \{x, y, z\}$$

Where ρ is the grid density parameter which in our case is equal to 5. The number of cells M is equal to the grid resolution $M_x \times M_y \times M_z$. N is the number of objects in the scene. S_i is the scene bounding box size in dimension i. V is the bounding box volume.

Ray shooting is implemented by traversing the grid cells intersected by a ray [AW+87] from its point of entrance to its point of exit.

For typical scenes most uniform grid cells will be empty. This means some form of sparse matrix compression scheme is desirable. In our case we implemented the row displacement compression scheme, represented in Figure 2, described in [LD08].

The process of computing the offsets for each row can be parallelized in the GPU as can the other steps of uniform grid construction. Hence we arrive at Algorithm 1.

3 METHOD

The row displacement compression method stores grid rows, in an overlapped fashion, inside a 1D array L. A 2D array O stores the offsets to the start of each grid row inside L. Prior to accessing this hashed grid a 3D
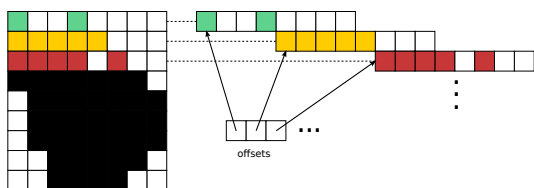

Figure 2: Row displacement compression.

Algorithm 1 Parallel compressed grid build.

1: **function** BUILDCOMPRESSEDGRID($M, objects$)
2:     $D \leftarrow$ DOMAINBITS($M, objects$)
3:             ▷ population count followed by reduce
4:     $nonEmpty \leftarrow$ NONEMPTYCELLS(D)
5:     $NO, NbH \leftarrow M_y \times M_z, 2 \times nonEmpty$
6:     $O, last \leftarrow$ FILLOFFSETS(NO, NbH, M, D)
7:     $NH \leftarrow last + 1$
8:     $H \leftarrow$ COMPUTEOFFSETS($NH, M, O, objects$)
9:                             ▷ inclusive scan
10:     $NL \leftarrow$ COMPUTEPREFIXSUM(H)
11:     $L \leftarrow$ INSERTINDICES($NL, M, O, H, objects$)
12:     **return** D, O, H, L
13: **end function**

---

bit array D, also known as the domain bits array, is consulted to determine if that particular cell is occupied.

In our implementation the domain bits, which state if a grid cell is empty or not, are stored as a linear bit array. Internally the bit array is composed of unsigned ints with 32 bits each. Domain bit computation, as other steps in the algorithm, is made in parallel: for all objects in the scene we determine which cells they overlap, then insert then into the domain bits with atomic memory operations. The computation of the number of non-empty cells is done by a population count pass, followed by a scan pass.

Row displacement compression offsets are computed in the next step and stored in the O array. This step of the algorithm is computationally expensive since it computes a mapping of the grid rows into a compressed 1D array using a find-first-fit scheme. The H array is computed by storing the number of objects which overlap each cell with atomic memory operations. The prefix sum is then computed so each cell points to the tail of its item list. Finally object indices are inserted into the item list with atomic operations. The atomic locks have a fine granularity in order not to constrain parallelism.
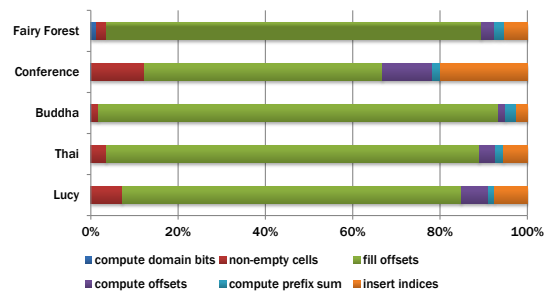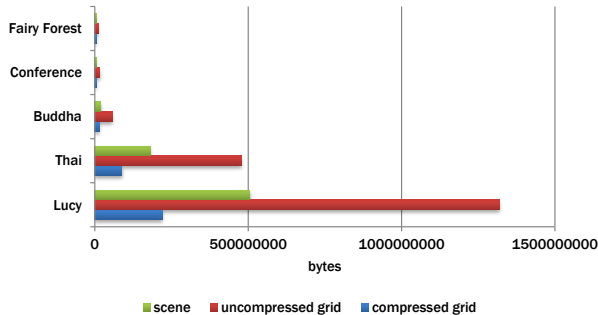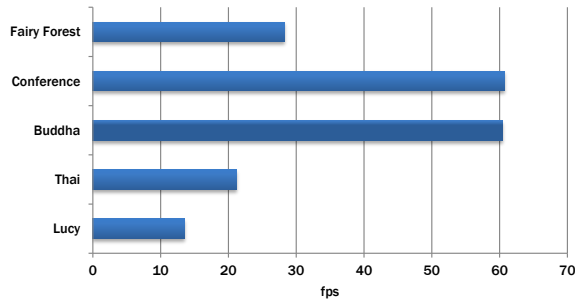

Figure 3: Percentage of time spent in each step of GPU grid construction.

As can be seen in Figure 3 the most time consuming operation is the FILLOFFSETS step where row offsets are computed using the first-fit method.

(a) Memory used to store the scene and the grid acceleration structure with and without compression.

(b) Framerates for selected scenes.

Figure 4: Memory consumption and rendering frame rates for the test scenes.

The singled-threaded CPU implementation of this algorithm has better performance for scenes with small grid row M_x sizes since these are mostly serial workloads where it is not possible to extract enough row parallelism for the GPU to prevail. However as can be seen in Figure 5 the GPU implementation dominates for the larger scenes with more geometry and correspondingly larger row sizes.

4   RESULTS

Our test platform is 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 660 Ti GPU with 2 GB of RAM.

The implementation language is ANSI C++ for the host code and OpenCL running on the GPU for the compute kernels. During rendering the local work group size is set to 16x16 blocks in order to maximize cache locality and take advantage of pixel parallelism. The application runs on the Linux operating system. All images were rendered at 1024×1024 resolution using one ray per pixel and diffuse shading.

Ray/triangle intersection is done with the Möller-Trumbore algorithm [MT97] since it does not require the usage of any additional memory. Each triangle uses 12 bytes of memory to store the vertex indices and each vertex uses 12 bytes of memory. For scenes with normals each vertex normal also uses 12 bytes of memory.

The Fairy Forest and Conference scenes are representative of the scenes you can typically find in a computer game with irregular polygon density i.e. high polygon count objects inside a lower polygon count environment with walls. The Buddha, Thai Statue, and Lucy models represent scanned scenes with triangles of similar area. These scanned scenes feature larger total polygon counts than the first two.

As can be seen in Table 1 our algorithm has good rendering performance compared to previous work on GPU single-level grids [KS09] and two-level grids

Scene	Grid GTX 280	2lvl Grid GTX 470	Compressed Grid GTX 660 Ti CPU	GPU
Fairy Forest	24 ms 3.5 fps	8 ms 21 fps	20 ms	65 ms 28 fps
Conference	27 ms 7.0 fps	17 ms 26 fps	48 ms	78 ms 61 fps
Thai Statue	417 ms -	257 ms -	537 ms	375 ms 21 fps

Table 1: Performance comparison of our Compressed Grid implementation, with the Grid from [KS09], and the 2lvl Grid from [KBS11]. The table lists grid build times and rendering frame rates. The Thai Statue scene frame rate performance was not specified in those articles. For the Compressed Grid the CPU and GPU implementations of FILLOFFSETS were tested on grid construction.

[KBS11]. This is probably due to our algorithm having improved cache coherence. Our algorithm requires less memory bandwidth per cell traversal. The GPU we are using, the GTX 660 Ti, has similar bandwidth compared to the earlier GTX 480. However the GTX 660 Ti has much improved peak floating point performance making it hard to judge the improvement of the work based on the strengths of a software implementation alone. Two-level grids typically have better render time performance than single-level grids. However our single-level grid implementation on a GTX 660 Ti has better rendering performance than the previously
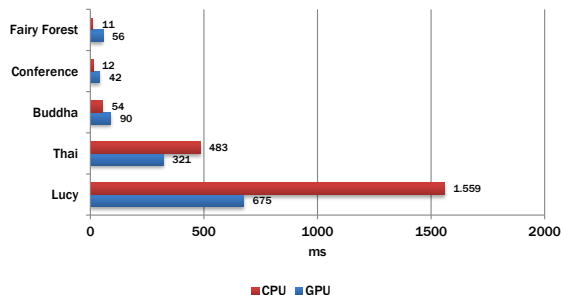

Figure 5: Time required to fill the offset table using the GPU vs the CPU.
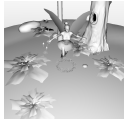
	Fairy Forest	Conference	Buddha	Thai Statue	Lucy
Scene					
Triangles	173.98 K	282.76 K	1.09 M	10.00 M	28.06 M
Memory	4.21 MB	5.15 MB	18.67 MB	171.66 MB	481.60 MB
Uncompressed Grid					
Memory	11.22 MB	15.44 MB	53.50 MB	455.90 MB	1.22 GB
Compressed Grid					
Memory	5.41 MB	5.44 MB	13.84 MB	83.85 MB	210.45 MB
Build Time	64.77 ms	77.83 ms	98.68 ms	375.08 ms	870.15 ms
Frame Rate	28.30 fps	60.72 fps	60.40 fps	21.22 fps	13.52 fps

Table 2: Scene statistics and grid performance results versus an uncompressed grid data structure [KS09].

mentioned two-level grid implementation on a GTX 480.

The advantages of the compressed grid are more debatable on grid construction. While our implementation features better performance than previous work on larger scenes with tens of millions of triangles, like the Thai Statue, it has poorer performance on the lower polygon scenes. This is due to the time spent performing grid compression namely the FILLOFFSETS phase, as seen in Figure 3, where rows offsets are computed using a first-fit method. For such low polygon count scenes a single-threaded CPU implementation has better performance than our GPU kernel as can be seen in Figure 5. While build times are not an issue for static scenes this remains an open problem in this GPU grid implementation.

Our most important objective is the reduction of memory consumption in order to enable the visualization of larger scenes. As can be seen in Table 2 the implemented single-level compressed grids feature much lower memory consumption than uncompressed single-level grids. Thus compressed grids enable the visualization of more complex scenes. In our case we can visualize the Lucy statue using a sixth of the memory required for a grid without compression as can be seen in Figure 4a.

5 CONCLUSIONS AND FUTURE WORK

This work enables the visualization of large scenes, with tens of millions of triangles, on the GPU. The acceleration structure construction and rendering is performed in parallel in the GPU. The algorithm provides real-time frame rates for scenes with millions of triangles.

For dynamic scenes with destructible geometry we require more rapid grid construction times. This may be achieved with alternative hashing or sparse matrix storage algorithms. This work also does not have optimiza-tions for highly coherent rays such as the use of ray bundles.

6 ACKNOWLEDGEMENTS

REFERENCES

[Áfr12] A.T. Áfra. Incoherent ray tracing without acceleration structures. In *Eurographics - Short Papers*, pages 97–100. Eurographics Association, 2012.

[AW+87] J. Amanatides, A. Woo, et al. A Fast Voxel Traversal Algorithm for Ray Tracing. In *Proceedings of Eurographics*, volume 87, pages 3–10. Eurographics Association, 1987.

[HSHH07] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive k-D Tree GPU Raytracing. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, pages 167–174. ACM, 2007.

[KBS11] J. Kalojanov, M. Billeter, and P. Slusallek. Two-Level Grids for Ray Tracing on GPUs. In *Computer Graphics Forum*, pages 307–314. Wiley Online Library, 2011.

[KS09] J. Kalojanov and P. Slusallek. A Parallel Algorithm for Construction of Uniform Grids. In *Proceedings of the Conference on High Performance Graphics*, pages 23–28. ACM, 2009.

[LD08] A. Lagae and P. Dutré. Compact, Fast and Robust Grids for Ray Tracing. In *Computer Graphics Forum*, pages 1235–1244. Wiley Online Library, 2008.

[LGS+09] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH construction on GPUs. In *Computer Graphics Forum*, pages 375–384. Wiley Online Library, 2009.

[MT97] T. Möller and B. Trumbore. Fast, Minimum Storage Ray-Triangle Intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997.

[PL10] J. Pantaleoni and D. Luebke. HLBVH: Hierarchical LBVH Construction for Real-Time Ray Tracing of Dynamic Geometry. In *Proceedings of the Conference on High Performance Graphics*, pages 87–95. Eurographics Association, 2010.