Mutable array expressions for efficient parallel raytracing in Accelerate

Hugo Peters

v1.1

1 Abstract

Raytracing, like many other techniques in the domain of computer graphics, is an embarrassingly parallel algorithm. It produces an image by finding the closest intersection between a ray, derived from the pixel location in the render-target, and the scene. As scene complexity grows it quickly becomes unfeasible not to employ an acceleration structure that reduces the process of finding an intersection from linear to logarithmic time. The constant cost of traversal of such structures can be greatly reduced by utilizing a small in memory stack. In this paper we will explore how such a traversal procedure may be implemented in Accelerate, and propose an extension to the embedded language to better facilitate this.

2 kd-tree vs BVH

For raytracing it is common to use a kd-tree [5] or a BVH (bounding volume hierarchy) [6]. The main benefit of a kd-tree is the fact that it natively supports a stateless traversal algorithm. Although its performance can be improved by using a short stack this is not strictly necessary [2]. On the other hand, a BVH can in principal only be traversed using a local stack. Conversely, BVHs are much faster to traverse on parallel hardware as well as having a significantly lower memory footprint. [3] With the growing availability of fast on chip memory on modern GPUs, this has made a BVH the defacto standard for realtime raytracing.

3 Other approaches

These approaches were considered first but deemed unsuccessful.

3.1 Stack as an extra dimension in the array

Given a scene, a logarithmic upper bound can be derived on the size of the stack. This allows for a preallocation of memory (albeit rather permissive), of $|pixels| * bound(scene)$ bvh nodes in global memory. However, this method is a rather terrible idea considering performance. Namely, the need for many global memory fetches leads to orders of magnitude longer stalling times than its local memory counterpart. As we know, memory bandwidth is the bottleneck for raytracing on modern hardware [1], and as such we cannot hope to hide the stalling behind computational tasks exhaustively.

3.2 Using nested tuples

Using a type family, it is possible to express a statically sized array as a nested tuple:

```
-- Simplified representation, an actual haskell implementation
-- requires a bit more finess
Array 0 a => ()
Array n a => (a, Array (n-1) a)
```

However, this causes n different variables to be defined, without the guarantee that they will live in contiguous memory. Furthermore, the push and pop operations must require branching code. All of the above promises a non competitive running time. To top it all off, the running time of simplifier stage of the accelerate compiler is heavily impacted by these nested entities, leading to compile times in the order of tens of seconds for stacks of only size 10. Do not forget that with this embedded language that cost is incurred at every run, not during the compilation of the program.

4 Adding Exp local arrays to the Accelerate expression language

Albeit the most daunting solution, it is the most obvious one. After all it is the only way to leverage the full hardware potential. More specifically, it's the lack of the ExtractElement and InsertElement LLVM instructions in the generated code that is withholding the full potential. It seems reasonable to expose map these instructions simply as is the front end with two new AST entries:

```
VecIndex      :: (KnownNat n)
              => VectorType (Vec n s)
              -> IntegralType i
              -> OpenExp env aenv (Vec n s)
              -> OpenExp env aenv i
              -> OpenExp env aenv s
```

```
VecWrite      :: (KnownNat n)
              => VectorType (Vec n s)
              -> IntegralType i
              -> OpenExp env aenv (Vec n s)
              -> OpenExp env aenv i
              -> OpenExp env aenv s
              -> OpenExp env aenv (Vec n s)
```

Of course this also requires an implementation in the various back-ends but is omitted because whilst complicated in practice, have no other effect than simply emitting said instructions in the LLVM IR.

It is not hard to imagine how to implement a stack on top of this functionality.

5 Results

The results can be found in fig. 1

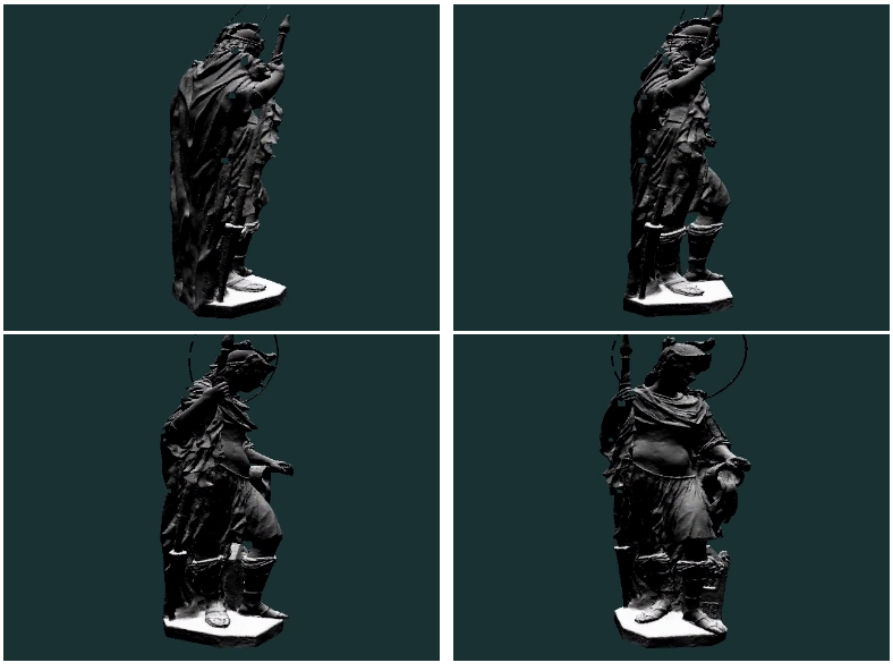

Figure 1: The resulting raytracing program. Easily running in realtime a RTX 3070M at a resolution of 640×480. The scene consists of 278502 triangles, contained in a BVH with 32767 nodes.

6 Current limitations

The resulting program is unlikely to successfully scale to much larger resolutions or more complex scenes for two major reasons.

The first is the lack of interop between the cuda texture and the opengl framebuffer, causing cpu hungry textures transfers back and forth between the host and the device.

The latter is the suboptimal BVH construction technique. For simplicity a very simple split on largest axis until $|triangles| < n$ where n is some target number of triangles in a leaf (e.g. 30) was used. More sophisticated techniques such as the SAH heuristic have been shown to lead to much more desirable performance whilst still being efficient to construct (albeit greedily) [4].

7 Moving forward

It is likely that a larger performance gain may be won by closer analysis of the memory layouts and anticipation of the memory needed by the trace kernel.

However, with the rise of RTX and similar hardware accelerated raytracing capabilities of modern graphics cards, it seems unwise to continue the raytracing specific efforts of this project.

Instead, one may consider how `Exp` local arrays and its emergent capabilities may benefit, or even facilitate, the implementation and consequent performance of other parallel algorithms using the Accelerate library.

References

[1] Timo Aila and Tero Karra. Architecture considerations for tracing incoherent rays, 2010. URL: https://research.nvidia.com/sites/default/files/pubs/2010-06_$Architecture-Considerations-for/aila2010hpg_paper.pdf$.

[2] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree gpu raytracing. *Proceedings of the 2007 symposium on Interactive 3D graphics and games - I3D 07*, 2007. URL: https://graphics.stanford.edu/papers/i3dkdtree/gpu-kd-i3d.pdf, https://doi.org/10.1145/1230100.1230129 doi:10.1145/1230100.1230129.

[3] Marek Vinkler, Vlastimil Havran, and Jiří Bittner. Bounding volume hierarchies versus kd-trees on contemporary many-core architectures. *Proceedings of the 30th Spring Conference on Computer Graphics - SCCG 14*, 2014. URL: https://dcgi.fel.cvut.cz/home/havran/ARTICLES/sccg2014.pdf, https://doi.org/10.1145/2643188.2643196 doi:10.1145/2643188.2643196.

[4] Ingo Wald. On fast construction of sah-based bounding volume hierarchies. URL: https://www.sci.utah.edu/publications/wald07/fastbuild.pdf.

[5] Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in o(n log n). *2006 IEEE Symposium on Interactive Ray Tracing*, 2006. URL: http://www.sci.utah.edu/publications/SCITechReports/UUSCI-2006-009.pdf, https://doi.org/10.1109/rt.2006.280216 doi:10.1109/rt.2006.280216.

[6] Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in o(n log n). *2006 IEEE Symposium on Interactive Ray Tracing*, 2006. URL: http://www.sci.utah.edu/publications/SCITechReports/UUSCI-2006-009.pdf, https://doi.org/10.1109/rt.2006.280216 doi:10.1109/rt.2006.280216.