Simpler and Faster HLBVH with Work Queues

Kirill Garanzha NVIDIA
Jacopo Pantaleoni NVIDIA Research
David McAllister NVIDIA
Short Summary

- Full GPU implementation
- Simple work queues generation
- Simple middle split hierarchy emission
- Efficient and straightforward top-level SAH tree emission
- Enabled by fast atomic instructions
3 improvements over HLBVH 2010

• Fast work queues
• Spatial-median BVH splits based on binary search
• Top-level SAH BVH build on GPU
What is work queue for GPU

Problem: Generates variable # of work items

Target: Generate the output queue with no holes
And get full GPU utilization

Input Queue data

Output Queue data

# output elems

GPU parallel processing

And get full GPU utilization
work queue generation

- Option 1: output item address = atomic increment on global counter for each element of input queue
  - Pros: no need to store temp results
  - Cons: many conflicting writes
work queue generation

• Option 2: output item address = prefix sum
  • Pros: no conflicting writes
  • Cons: need to store results in temp memory and propagate them to output queue (global memory is limited on GPU)
work queue generation

• Option 3: hybrid approach
  (local prefix sum + atomic / warp)
  • Pros: unites the pros of both methods and remove their cons;
  • Pros: fastest!
work queue generation

• Option 3: hybrid approach

- Every prefix sum is computed in shared memory.
- No need to spend limited global memory.
- Temp results stored in each thread memory

Input Queue data

<table>
<thead>
<tr>
<th># output elems</th>
<th>2</th>
<th>0</th>
<th>0</th>
<th>2</th>
<th>0</th>
<th>2</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>warp exclusive prefix sum (warp)</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Output item offset (global)</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

output counter += warp[0] sum
output counter += warp[1] sum

- Atomic increments are fast on Fermi
- Reduce conflicting writes when used once per warp
WQ application: HLBVH pipeline

1. Morton Codes for each primitive
2. DuaneSort
3. Distribute the primitives into clusters
4. Emit sub-tree spatial-median BVH for each cluster
5. Emit binned SAH top-level BVH over clusters
6. Refit AABBs

Work queues used in iterative processes
Distribute primitives into clusters

• Sample 4bit morton codes
• 2bit clusters are determined with 2 most significant bits of each prim morton code
• Compression from CSD is used to extract the segments of clusters
• 15bit clusters are good for high-quality BVH
Spatial median BVH emission

Each morton code expanded into bits shown as row per bit level

<table>
<thead>
<tr>
<th>Primitives</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>mortoncode[bit3]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>mortoncode[bit2]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>mortoncode[bit1]</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>mortoncode[bit0]</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

• Primitives sorted according to Morton Codes
• Find the split in the range of primitives using trivial binary search per thread: log2(n) memory fetches and comparison
• Very simple implementation of binary search merges well with simple work queues
Binned SAH top-level BVH

- 15bit clusters are good for high-quality BVH...
- Up to 32K clusters, each represent median-split BVH with a bounding box
- We have implemented [Wald IRT 2007] binned BVH builder using CUDA
- Atomic instructions of Fermi provide with good results and simple implementation
- Pros: Everything stays in GPU memory, no transfer costs

1. Map clusters to bins
2. Compute SAH using bins, then split
Results

- 10x faster than original HLBVH 2010
- Uses 4x less GPU device memory for BVH emission
- Quality of HLBVH is 10-15% lower than the quality of full SAH BVH
- Can build PowerPlant (12.7M triangles) in-core on GTX480 in 62ms
Results

- GTX 480:
  - Fairy Forest BVH (174K triangles) – 4.8ms
  - Turbine Blade BVH (2M triangles) – 10.5ms
  - Power Plant (12M triangles) – 62ms
Results

- Fast work queues enable a very high speed GPU based BVH build
- We believe they will enable a large class of parallel algorithms to be more straightforward on GPUs