OpenCL Implementation Of A Heterogeneous Computing System For Real-time Rendering And Dynamic Updating Of Dense 3-d Volumetric Data

Andrew Miller
Computer Vision Group
Research Developer
3-D TERRAIN RECONSTRUCTION FROM IMAGES
## APPLICATIONS

<table>
<thead>
<tr>
<th>Augmented Reality</th>
<th>Change Detection</th>
<th>Geo Referencing</th>
<th>Geo Positioning</th>
<th>Visibility</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>app</strong></td>
<td><img src="image" alt="app" /></td>
<td><img src="image" alt="Geo Reference" /></td>
<td><img src="image" alt="Geo coordinates" /></td>
<td><img src="image" alt="Visibility" /></td>
</tr>
<tr>
<td><strong>2-d</strong></td>
<td><img src="image" alt="2-d" /></td>
<td><img src="image" alt="Change Detection" /></td>
<td><img src="image" alt="2-d model" /></td>
<td><img src="image" alt="2-d Visibility" /></td>
</tr>
<tr>
<td><strong>3-d</strong></td>
<td><img src="image" alt="3-d" /></td>
<td><img src="image" alt="3-d model" /></td>
<td><img src="image" alt="3-d Visibility" /></td>
<td><img src="image" alt="3-d Visibility" /></td>
</tr>
</tbody>
</table>

- Geo Reference the input image
- Geo coordinates (Long=?, lat=?, ele=?)
- Vantage Point
- Vantage point
- Vantage point
GEO-POSITIONING

Input

Geo-coordinates
(Long=?, lat=?, ele=?)

Output

3-d model

Planar model
APPLICATIONS

- Example: Change Detection:

  Change detection on planar model

  Change detection on 3-d model
WHY IS IT CHALLENGING?

- occlusion
- rapidly varying spatial scale
- shadows
- featureless surface

3-d texture

two views of a surface

the resulting ambiguity
### EXISTING METHODS

<table>
<thead>
<tr>
<th>3-d Method</th>
<th>Pros</th>
<th>Cons</th>
<th>References</th>
</tr>
</thead>
</table>

**Furukawa 2008**

**Crispell 2009**
VOLUMETRIC MODELING
VOLUMETRIC MODELING

- Basics
  - Voxel data and probabilistic framework
  - Ray tracing
    - Updating
    - Rendering
- Memory Footprint
  - Spatial subdivisions
  - Efficient spatial encoding and Data storage
  - Refining
- GPU Implementation
  - Ray Tracing
  - Updating in parallel, with GPU optimizations
VOLUMETRIC MODEL

- Given a volume, divide into “volumetric pixels”, called voxels
Given a volume, divide into “volumetric pixels”, called voxels
**VOXEL DATA**

- Two probability distributions stored in each voxel:
  - Occupancy probability (alpha)
  - Appearance model (Mixture of 3 Gaussians)

\[ P(X \in S) \]

Probability that voxel \( X \) is in a surface

\[
P(X \in S) = \sum_{i=0}^{N} w_i \pi_i
\]

Appearance modeled as a mixture of

\[
I(\text{intensity})
\]
MODEL MANIPULATION

- Divide volume into voxels that store two distributions
- Need to learn the correct (best) parameters for these two distributions
- Two main procedures to manipulate our model
  - Update (model reconstruction)
    - Given a set of observations and cameras, learn distributions in each voxel
  - Render (expected images)
    - Given a novel viewpoint, render expected image of the volume
- Both of these procedures are accomplished by casting rays
RAY CASTING

- Cast ray through the image plane into volume
- Sum relevant statistics as rays intersect voxels
**MODEL ALONG A RAY**

- Assume one voxel along a ray produces observed intensity
  - Must be un-occluded, and have high surface probability
  - How likely some voxel $X_\alpha$ is responsible for observation at $r$:

$$P(V_r = X_\alpha) = P(X_\alpha \in S) \prod_{\alpha' < \alpha} [1 - P(X_{\alpha'}, \in S)]$$

*Surface likelihood*  Visibility of voxel alpha
**UPDATING**

- Use an online algorithm to update surface probabilities:

\[
P^{N+1}(X \in S) = P^N(X \in S) \frac{p^N(I^{N+1} | X \in S)}{p^N(I)}
\]

Updated surface likelihood  
Previous surface likelihood  
Bayes’ update factor
**RENDERING**

- Marginalize appearance across ray
  - Calculate visibility of each cell along ray
  - Keep running sum of the expected intensity:

\[
E[I_r] = \sum_\alpha P(V_r = X_\alpha) E[p(I | X \in S)]
\]

Expected image sequence of downtown Providence, RI
MEMORY FOOTPRINT
VOLUMETRIC MODELING

- Basics
  - Voxel data and probabilistic framework
  - Ray tracing
    - Updating
    - Rendering
- Memory Footprint
  - Spatial subdivisions
  - Efficient spatial encoding and Data storage
  - Refining
- GPU Implementation
  - Ray Tracing
  - Updating in parallel, with GPU optimizations
MEMORY FOOTPRINT

- Existing scenes (downtown and capitol) are 1536 by 1536 by 512 ≈ 1.2 billion voxels
- Each voxel carries at least 12 bytes of information:
  - 4 bytes for occupancy probability
  - 8 bytes for appearance model
- Would require about 14 GB of information!
- Reducing Memory in two ways
  - Spatial subdivision
  - Efficiently encoded octree structure
SPATIAL SUBDIVISION

- Fixed grid of shallow octrees
  - Trees have depth limit 4 (512 total voxels)
  - Designed for optimal performance with GPU architecture

An octree is a data structure in which each internal node has exactly eight children. Octrees are used to partition a three dimensional space by subdividing it into eight octants.
**BIT-ENCODED OCTREE**

- Represent tree structure as implicit octree
  - String of bits, each bit indicates if cell has children
  - 73 bits are sufficient for depth four tree (10 bytes)
  - 4 reserved for data pointer (and align to 16)

<table>
<thead>
<tr>
<th>Tree bits [0-9]</th>
<th>Data Ptr [10, 11, 12, 13]</th>
<th>[]</th>
<th>[]</th>
</tr>
</thead>
</table>

- Simple formulas to find child and parent bits:
  
  \[
  parent\ (i) = \left\lfloor \frac{i - 1}{8} \right\rfloor \quad \text{child}\ (i) = 8i + 1
  \]

- Tree structure can be stored in local memory (shared memory) per thread
  - Accelerates tree traversal
**BIT-ENCODED OCTREE**

- Data stored in BFS order, tree points to root cell
- Simple sum calculate data index:

\[
data\_index(i) = 8 \left( \sum_{j=0}^{\text{parent}(i)-1} \text{tree}\_bit(j) \right) + 1\]

- Binary example:

(a) Bits encoding octree found in (b).
(b) Binary tree where each node’s data is a letter
(c) Location of each letter in memory
DATA STORAGE

- Modular data storage
  - Voxel data stored separately from tree structure
  - Minimizes data transfer for kernels that only need some data (i.e. rendering, update passes)
  - Indexing identical across data buffers
REFINE

- Once the surface probability passes a certain threshold, cell is refined into 8 octants
REFINE

- Refining one tree

Surface Probability Buffer

New data cells
REFINE

- Difficult to parallelize
  - Cannot refine all trees in parallel in place as race conditions may occur (data written over other new data)
- Refine in three steps:
  - Compute and save new tree structure, size of each new tree
  - Run a prefix sum on tree sizes to determine new location of each tree's data
  - Allocate new, sufficiently large data buffer, swap data into new locations
### REFINE

1. Compute new Tree Structure, sizes

2. Compute cumulative sum of size buffer using parallel prefix sum (determines new data indices)

3. Swap data into new location (initializing new cells)
GPU IMPLEMENTATION
VOLUMETRIC MODELING

- Basics:
  - Voxel data and probabilistic framework
  - Ray tracing
    - Updating
    - Rendering

- Memory Footprint
  - Spatial subdivisions
  - Efficient spatial encoding and Data storage
  - Refining

- GPU Implementation
  - Ray Tracing
  - Updating in parallel, with GPU optimizations
**GPU IMPLEMENTATION**

- Ray tracing is expensive on volumetric models
  - Each voxel along the path of a ray needs to be intersected and examined
- Very parallel algorithm
  - Can use GPU to process multiple rays simultaneously
- Note that now we are ray tracing a hybrid grid-octree structure
RAY CASTING REVISITED

Camera Center

Image Plane

Workgroups are 8x8 patches of image (64 rays)
RAY CASTING

- Tracing through a fixed grid of octrees
- Two approaches:
  - Two nested while loops (one traversing the grid, the other traversing the current octree)
  - Single while loop, each ray caches its own current tree
Double Loop Method

Outer Loop: Iterating over regular-grid
Double Loop Method

Outer Loop: Iterating over regular-grid
Double Loop Method

Outer Loop: Iterating over regular-grid

Inner Loop: Iterating over Octree

Each ray reads octree (16 bytes) into local memory
Double Loop Method

Outer Loop: Iterating over regular-grid

Each ray reads octree (16 bytes) into local memory

Inner Loop: Iterating over Octree
Double Loop Method

Outer Loop: Iterating over regular-grid

Each ray reads octree (16 bytes) into local memory

Inner Loop: Iterating over Octree
Double Loop Method

Outer Loop: Iterating over regular-grid

Each ray reads octree (16 bytes) into local memory

Inner Loop: Iterating over Octree
Double Loop Method

Outer Loop: Iterating over regular-grid

Inner Loop: Iterating over Octree

Each ray reads octree (16 bytes) into local memory
Double Loop Method

Outer Loop: Iterating over regular-grid
Double Loop Method

Outer Loop: Iterating over regular-grid

New octree loaded into local memory

Inner Loop: Iterating over Octree
Double Loop Method

Outer Loop: Iterating over regular-grid

New octree loaded into local memory

Inner Loop: Iterating over Octree
Double Loop Method

Outer Loop: Iterating over regular-grid

Inner Loop: Iterating over Octree

New octree loaded into local memory

Note how the two upper rays must wait for the lower ray to finish (due to OpenCL branching)
Outer Loop: Iterating over regular-grid

Note how the two upper rays must wait for the lower ray to finish (due to OpenCL branching)
Single Loop Method

Rays traverse fixed grid and octrees
Single Loop Method

Rays traverse fixed grid and octrees
Each ray reads octree (16 bytes) into local memory

Single Loop Method

Rays traverse fixed grid and octrees
Bottom ray does not wait, loads next block into local memory
Single Loop Method

Bottom ray does not wait, loads next block into local memory

Rays traverse fixed grid and octrees
Single Loop Method

Bottom ray does not wait, loads next block into local memory

Rays traverse fixed grid and octrees
Single Loop Method

Bottom ray does not wait, loads next block into local memory

Rays traverse fixed grid and octrees
**RAY CASTING**

- Single while loop tracing performs better
  - Threads do not have to wait for neighbors to continue processing
  - Global memory transfer time is hidden by scheduler
  - Operations on tree in local memory are fast
**UPDATING**

- Complications with update:
  - Multiple rays can pass through the same cell; creates race conditions
  - Need atomic operations to update global memory
    - Global writes (especially atomics) are very expensive

Two rays (in red) update the same cell – need atomic (serial) operations.
Determine connected components in two passes:

Example: 4x4 work group: Matrix of voxel IDs in local memory. One element per work-item (thread).

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
Determine connected components in two passes:

1) Each thread in the work group connects to the right

Example: 4x4 work group:
Matrix of voxel IDs in local memory. One element per work-item (thread)
Determine connected components in two passes:

1) Each thread in the work group connects to the right
2) Each tail looks at the next row, continues linked list

Example: 4x4 work group: Matrix of voxel IDs in local memory. One element per work-item (thread)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

0 → 0 → 1 → 1
0 → 2 → 2 → 2
2 → 2 → 3 → 3
3 → 3 → 3 → 4
The head of each list sums up their contribution to their cell.

The head of each list then makes an atomic call to update global memory.

Example: 4x4 work group: Matrix of voxel IDs in local memory. One element per work-item (thread)
SPEED AND MEMORY ANALYSIS
SPEED AND MEMORY ANALYSIS

- Tested multiple processes:
  - Render top down (fewer intersections), Render oblique (more intersections), Update empty scene, Update near converged scene, Refine
- Processes tested in OpenCL on:
  - multi-core (Gulftown) CPU
  - 200 series NVIDIA card
  - Fermi generation NVIDIA card
  - ATI 5870 series
- OpenCL code is currently optimized for NVIDIA Fermi generation cards
  - Unrolled 3d rays/points in float4 vectors to three floats to save register space/boost occupancy
  - Using 8x8 workgroup size
**RENDER VS OCCUPANCY**

Render time vs kernel occupancy for two generations of NVIDIA cards.

In our ray trace implementation, lower occupancy yielded faster results, with a steeper climb for the previous generation hardware.
**UPDATE VS WORKSPACE SIZE**

Update time vs workgroup size.

Optimal workgroup size appears to be 8x8 for both current and previous generation NVIDIA cards.
**SPEED AND MEMORY ANALYSIS**

- Render, update and refine benchmarks:

<table>
<thead>
<tr>
<th></th>
<th>CPU (intel i7-980 @ 3.33 GHz)</th>
<th>ATI 5870</th>
<th>NVIDIA GTX 280</th>
<th>NVIDIA GTX 480</th>
<th>NVIDIA GTX 580</th>
</tr>
</thead>
<tbody>
<tr>
<td>Render Top down</td>
<td>1.45s</td>
<td>.15s</td>
<td>.118s</td>
<td>.062s</td>
<td>.039s</td>
</tr>
<tr>
<td>Render Oblique</td>
<td>2.23s</td>
<td>.27s</td>
<td>.213s</td>
<td>.114s</td>
<td>.065ms</td>
</tr>
<tr>
<td>Update Empty</td>
<td>8.11s</td>
<td>4.72s</td>
<td>13.99s</td>
<td>.934s</td>
<td>.790s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.37s</td>
<td>1.81s</td>
<td>.795s</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(cache)</td>
<td>(cache)</td>
<td>(cache)</td>
<td></td>
</tr>
<tr>
<td>Update Converged</td>
<td>18.92s</td>
<td>n/a</td>
<td>n/a</td>
<td>1.363s</td>
<td>.999s</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.990s</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(cache)</td>
<td></td>
</tr>
</tbody>
</table>

Optimized OpenCL code for NVIDIA architecture. Benchmarks run out of the box on ATI card. ATI does vector ops in parallel, our implementation uses scalar operations.
FURTHER ANALYSIS

- Size of GPU memory currently restricts model size. Larger scenes stream blocks in and out of the GPU
  - Current render processing is faster than 5 gb/s (PCIe bandwidth)
  - Block streaming from host to device memory is a bottleneck
  - Larger scenes would benefit from an integrated CPU/GPU with higher bandwidth
FUTURE WORK
FUTURE WORK

- Goals:
  - More accurately model 3-d space
    - Volumetric (cone) ray tracing. Casting 3-d cone rays, instead of 2-d lines
    - Consider spatial continuity. Filtering voxels, accounting for neighboring joint distribution
    - Model underlying physical properties (surface normals, textures, non Lambertian properties) to make models robust to many surface types and lighting conditions
  - Build bigger models
    - Multi resolution scenes (using octree)
    - Streaming cache system (disk to RAM to GPU) for many blocks
**CONE RAY TRACE**

- Volumetric ray trace
  - Goal is to cover the entire volume when updating

Camera Center

Image Plane
More ray coverage removes some aliasing effects:
DATA STREAMING

- Streaming models for larger scenes
  - Begun implementing a streaming model
  - Asynchronous transfers between three levels of memory
QUESTIONS?
CODE PUBLICLY AVAILABLE AT:
http://vxl.sourceforge.net/
Disclaimer & Attribution

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. There is no obligation to update or otherwise correct or revise this information. However, we reserve the right to revise this information and to make changes from time to time to the content hereof without obligation to notify any person of such revisions or changes.

NO REPRESENTATIONS OR WARRANTIES ARE MADE WITH RESPECT TO THE CONTENTS HEREOF AND NO RESPONSIBILITY IS ASSUMED FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

ALL IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE ARE EXPRESSLY DISCLAIMED. IN NO EVENT WILL ANY LIABILITY TO ANY PERSON BE INCURRED FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

AMD, the AMD arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. All other names used in this presentation are for informational purposes only and may be trademarks of their respective owners.

The contents of this presentation were provided by individual(s) and/or company listed on the title page. The information and opinions presented in this presentation may not represent AMD’s positions, strategies or opinions. Unless explicitly stated, AMD is not responsible for the content herein and no endorsements are implied.