General Algorithm Primitives

Tim Purcell
NVIDIA

GPGPU
Topics

• Sorting
  - Sorting networks

• Search
  - Binary search
  - Nearest neighbor search

• Stream Filtering
Assumptions

- Data organized into 1D arrays
- Rendering pass == screen aligned quad
  - Not using vertex shaders
- PS 2.0 GPU
  - No data dependent branching at fragment level
Sorting
Sorting

- Given an unordered list of elements, produce list ordered by key value
  - Kernel: compare and swap
- GPUs constrained programming environment limits viable algorithms
  - Bitonic merge sort [Batcher 68]
  - Periodic balanced sorting networks [Dowd 89]
Bitonic Merge Sort Overview

• Repeatedly build bitonic lists and then sort them
  - Bitonic list is two monotonic lists concatenated together, one increasing and one decreasing.
    • List A: (3, 4, 7, 8) monotonically increasing
    • List B: (6, 5, 2, 1) monotonically decreasing
    • List AB: (3, 4, 7, 8, 6, 5, 2, 1) bitonic
8x monotonic lists: (3) (7) (4) (8) (6) (2) (1) (5)
4x bitonic lists: (3,7) (4,8) (6,2) (1,5)
Sort the bitonic lists
Bitonic Merge Sort

4x monotonic lists: (3,7) (8,4) (2,6) (5,1)

2x bitonic lists: (3,7,8,4) (2,6,5,1)
Sort the bitonic lists
Sort the bitonic lists
Bitonic Merge Sort

Sort the bitonic lists
Bitonic Merge Sort

2x monotonic lists:  (3,4,7,8) (6,5,2,1)
1x bitonic list:  (3,4,7,8, 6,5,2,1)
Bitonic Merge Sort

Sort the bitonic list
Bitonic Merge Sort

Sort the bitonic list
Sort the bitonic list
## Bitonic Merge Sort

### Sort the bitonic list

<table>
<thead>
<tr>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>7</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
Sort the bitonic list
Bitonic Merge Sort

Done!
Bitonic Merge Sort Summary

- Separate rendering pass for each set of swaps
  - $O(\log^2 n)$ passes
  - Each pass performs $n$ compare/swaps
  - Total compare/swaps: $O(n \log^2 n)$

- Limitations of GPU cost us factor of $\log n$ over best CPU-based sorting algorithms
Making GPU Sorting Faster

• Draw several quads with similar computation instead of single quad
  - Reduce decision making in fragment program

• Push work into vertex processor and interpolator
  - Reduce computation in fragment program

• More than one compare/swap per sort kernel invocation
  - Reduce computational complexity
Grouping Computation
Implementation Details

• Specify interpolants for smaller quads
  - ‘down’ or ‘up’ compare and swap
  - distance to comparison partner

• See Kipfer & Westermann article in GPU Gems 2 and Kipfer et al. Graphics Hardware 04 for more details
GPU Sort

- Use blending operators for comparison
- Use texture mapping hw to map sorting op.

[Govindaraju 05]
Searching
Types of Search

- Search for specific element
  - Binary search
- Search for nearest element(s)
  - k-nearest neighbor search

- Both searches require ordered data
Binary Search

• Find a specific element in an ordered list

• Implement just like CPU algorithm
  - Assuming hardware supports long enough shaders
  - Finds the first element of a given value \( v \)
    • If \( v \) does not exist, find next smallest element > \( v \)

• Search algorithm is sequential, but many searches can be executed in parallel
  - Number of pixels drawn determines number of searches executed in parallel
    • 1 pixel == 1 search
Binary Search

- Search for v0

Initialize 4

Search starts at center of sorted array

v2 >= v0 so search left half of sub-array
Binary Search

- Search for v0

Sorted List

<table>
<thead>
<tr>
<th></th>
<th>v0</th>
<th>v0</th>
<th>v0</th>
<th>v2</th>
<th>v2</th>
<th>v2</th>
<th>v5</th>
<th>v5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

Initialize

Step 1

v0 >= v0 so search left half of sub-array
Binary Search

• Search for v0

Initialize

Step 1

Step 2

v0 >= v0 so search left half of sub-array

Sorted List

GPGPU
Binary Search

• Search for v0

Sorted List

GPGPU

4 2 1 0

At this point, we either have found v0 or are 1 element too far left
One last step to resolve
Binary Search

- Search for v0

Sorted List

<table>
<thead>
<tr>
<th>Step 1</th>
<th>Step 2</th>
<th>Step 3</th>
<th>Step 4</th>
<th>Initialize</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

Done!
Binary Search

- Search for v0 and v2

Initialize 4 4

Search starts at center of sorted array

Both searches proceed to the left half of the array

Sorted List

<table>
<thead>
<tr>
<th>v0</th>
<th>v0</th>
<th>v0</th>
<th>v2</th>
<th>v2</th>
<th>v2</th>
<th>v5</th>
<th>v5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
Binary Search

- Search for v0 and v2

Initialize

Step 1

The search for v0 continues as before

The search for v2 overshot, so go back to the right
Binary Search

- Search for v0 and v2

<table>
<thead>
<tr>
<th>Initialize</th>
<th>Step 1</th>
<th>Step 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

We’ve found the proper v2, but are still looking for v0
Both searches continue
Binary Search

• Search for v0 and v2

Initialize
Step 1
Step 2
Step 3

Now, we’ve found the proper v0, but overshot v2
The cleanup step takes care of this
Binary Search

• Search for v0 and v2

Sorted List:

Done! Both v0 and v2 are located properly
Binary Search Summary

• Single rendering pass
  - Each pixel drawn performs independent search

• $O(\log n)$ steps
Nearest Neighbor Search
Nearest Neighbor Search

- Given a sample point $p$, find the $k$ points nearest $p$ within a data set

- On the CPU, this is easily done with a heap or priority queue
  - Can add or reject neighbors as search progresses
  - Don’t know how to build one efficiently on GPU

- $k$NN-grid
  - Can only add neighbors...
kNN-grid Algorithm

- sample point
- candidate neighbor
- neighbors found

Want 4 neighbors
kNN-grid Algorithm

- Candidate neighbors must be within max search radius
- Visit voxels in order of distance to sample point

- sample point
- candidate neighbor
- neighbors found

Want 4 neighbors
kNN-grid Algorithm

- If current number of neighbors found is less than the number requested, grow search radius

- sample point
- candidate neighbor
- neighbors found

Want 4 neighbors
kNN-grid Algorithm

- If current number of neighbors found is less than the number requested, grow search radius

- sample point
- candidate neighbor
- neighbors found

Want 4 neighbors
kNN-grid Algorithm

- Don’t add neighbors outside maximum search radius
- Don’t grow search radius when neighbor is outside maximum radius

- sample point
- candidate neighbor
- neighbors found

Want 4 neighbors
kNN-grid Algorithm

- Add neighbors within search radius

- Sample point
- Candidate neighbor
- Neighbors found

Want 4 neighbors
kNN-grid Algorithm

- Add neighbors within search radius

- sample point
- candidate neighbor
- neighbors found

Want 4 neighbors
**kNN-grid Algorithm**

- Don’t expand search radius if enough neighbors already found

- sample point
- candidate neighbor
- neighbors found

Want 4 neighbors
kNN-grid Algorithm

- Add neighbors within search radius

- sample point
- candidate neighbor
- neighbors found

Want 4 neighbors
**kNN-grid Algorithm**

- Visit all other voxels accessible within determined search radius
- Add neighbors within search radius

- sample point
- candidate neighbor
- neighbors found

Want 4 neighbors
kNN-grid Summary

- Finds all neighbors within a sphere centered about sample point
- May locate more than requested $k$-nearest neighbors

- sample point
- candidate neighbor
- neighbors found

Want 4 neighbors
Stream Filtering

- Select a subset of elements from a list, discard the rest
  - Irregular reduction operation
- **Stream compaction [Horn 05]**
  - Combine scan and search to compute filtered list
  - $O(\log n)$ passes
Stream Filtering

Input Stream

Compacted Stream

X - stream element to remove
Stream Filtering

Input Stream:

| p4 | X | p3 | p1 | X | X | p4 | p2 |

Scan:

| 0  | 1 | 1 | 1 | 2 | 3 | 3 | 3 |

Compacted Stream:

| p4 | p3 | p1 | p4 | p2 |

Scan step generates running count of Xs seen so far.
## Stream Filtering

| Input Stream |  
|--------------|---|
| p4 X p3 p1 X X p4 p2 |

| Scan |  
|------|---|
| 0 1 1 1 2 3 3 3 3 |

| Binary Search |  
|---------------|---|
| 0 1 1 3 3 |

| Compacted Stream |  
|------------------|---|
| p4 p3 p1 p4 p2 |

Binary search over scan results.  
Search result is offset to pull record from
Stream Filtering

Input Stream

```
p4  X  p3  p1  X  X  p4  p2
```

Binary Search

```
0  1  1  3  3
```

Compacted Stream

```
p4  p3  p1  p4  p2
```
Stream Filtering Summary

- Combine scan and binary search to compute filtered list
- $O(\log n)$ passes