Topics

• Sorting
  • Sorting networks

• Search
  • Binary search
  • Nearest neighbor search
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
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
Bitonic Merge Sort

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

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
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
Bitonic Merge Sort

Sort the bitonic list
Bitonic Merge Sort

Sort the bitonic list
Bitonic Merge Sort

Sort the bitonic list
Bitonic Merge Sort

3  3  3  3  3  2  1
7  7  4  4  4  1  2
4  8  8  7  2  3  3
8  4  7  8  1  4  4
6  2  5  6  6  6  5
2  6  6  5  5  5  6
1  5  2  2  7  7  7
5  1  1  1  8  8  8

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
## Grouping Computation

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>8</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

**GPGPU**
Implementation Details

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

[Govindaraju 05]
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

Initialize

Step 1

v0 >= v0 so search left half of sub-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

Initialize
Step 1
Step 2

v0 >= v0 so search left half of sub-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

Initialize 4
Step 1 2
Step 2 1
Step 3 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></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>
</tr>
</thead>
<tbody>
<tr>
<td>v0</td>
<td>GPGPU</td>
<td>GPGPU</td>
<td>GPGPU</td>
<td>GPGPU</td>
<td>GPGPU</td>
<td>GPGPU</td>
<td>GPGPU</td>
<td>GPGPU</td>
</tr>
<tr>
<td></td>
<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>

Initialize

Step 1

Step 2

Step 3

Step 4

Done!
Binary Search

- Search for v0 and v2

Initialize

Search starts at center of sorted array

Both searches proceed to the left half of the array
Binary Search

- Search for v0 and v2

Initialize

Step 1

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>
Binary Search

- Search for v0 and v2

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>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

Initialize: 4, 4

Step 1: 2, 2

Step 2: 1, 3

We’ve found the proper v2, but are still looking for v0

Both searches continue
Binary Search

- **Search for v0 and v2**

  Initialize: 4 4

  Step 1: 2 2

  Step 2: 1 3

  Step 3: 0 2

  Now, we’ve found the proper v0, but overshot v2
**Binary Search**

- **Search for v0 and v2**

  **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></td>
<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>

  - **Initialize**
    - Step 1: 4 4
    - Step 2: 2 2
    - Step 3: 1 3
    - Step 4: 0 2

  **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

- \( kNN\)-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
• 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