Efficient Volume Segmentation on the GPU

San Jose (CA) | September 23, 2010
Gernot Ziegler, Devtech-Compute, NVIDIA UK
Allan Rasmusson, University of Aarhus (Denmark)
Agenda

- Introduction / Problem Task
  - Input and Expected Output, Connectivity

- Algorithm
  - Label Setup and Label Propagation
  - Acceleration concepts (Links, Master/Slave)

- Implementation
  - Algorithm mapping to CUDA C
  - Tradeoff comparison for different strategies

- Results

- Conclusion
Introduction / Problem Task

- **Input**
  - 2D array / 3D array of data (typical image/volume data)
  - Connectivity Criterion (when are two elements connected?)

- **Example Input**
  - 2D RGB image
  - Connectivity Criterion: Equal colors, 8-connectivity
Introduction / Problem Task

- **Output**
  - Uniquely labelled regions: 2D Array / 3D Array with all connected regions having the same "label" (usually a 32bit integer value)

- **Example**
  - 2D array of labels

![Diagram of a 2D array of labels](image)
Connectivity Criterion

- When are neighboring cells "connected", become a region?

- Example criterion: Equal RGB values
  - Linked (= 1): [Linked image] symbolized as: [Linked symbol]
  - Not linked (= 0): [Not linked image]

- More useful criterions for noisy input:
  Color gradient thresholding
  e.g. \( \text{Sum}(\text{abs}(p0.rgb - p1.rgb)) < 0.1 \)

- Others: Motion data, n-edged graphs, ...
2D: 4- and 8-connectivity

- Are diagonal neighbors regarded as "connected"?

4-connectivity: Look at vertical and horizontal neighbors

8-connectivity: Also look at Diagonal Neighbors
4- and 8-connectivity

- Affects label propagation!
- Labelling results can differ substantially:

4-connectivity labelling: Upper and lower part separate

8-connectivity labelling: Upper and lower part connected
Algorithm:
Label Setup and Propagation
Label setup

- Each cell has its own label (\( p.rgb = f(p.x, p.y) \))
- Labels are comparable in a strict linear order, e.g. \( L=y\times \text{width}+x \)
- Also, \((x,y)\) can be recovered from label - e.g. Red=X, Green=Y
Simple Label Propagation: 1-gather

- Larger labels propagate to connected cells with smaller labels
- Cells gather from their neighbours: 1-gather
- Completely data-parallel with double-buffering and gather

**Finish**: When no more updates occur!
Algorithm Optimization: Links and max-gather
Problem of 1-gather algorithm: SLOW
(Each pass, labels propagate only one cell further)

Can we make labels propagate faster?

Observation: Connectivity between cells is \textit{static}!

Precompute the furthest connected cell along each connectivity direction (e.g. $x, y, z$)

$\log_2(\text{width} | \text{height} | \text{depth})$ steps

(Similarities with Horn's data-parallel algorithm for prefix sum, GPU Gems 1)
Links: Precomputation Algorithm

- Initialize with local connectivity.

- Repeatedly add cell value that link *points to*.

- Example shown: Computing furthest connected cell to the right.
Links: Directions

- One entry for each cell and each direction
- Example: 4-connectivity links for a cross of connected cells:
Labels: Faster Gathering

- Link Precomputation stage permits **far-away label gathering**

  - **1-gather**

  "Black" = Irrelevant Label

  Links result in faster label propagation

- **Max-gather (via Links)**

  Links result in faster label propagation
Max-gather doesn't suffice

- One might assume that 1-gather is not necessary anymore.
- **BUT:** there are cases where max-gather doesn't fill all cells!

**Links Data:**
Cross of connected cells

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Green Label is largest - Attempted max-gathering**

- Label result (incomplete)

**Black label color = Smaller/Irrelevant Label**
Max-gather doesn't suffice

- 1-gather is still necessary to fill in the unlabelled holes!
Algorithm Optimization: Master/Slave
Master cells

- In each region, one cell keeps its original label
- All other cells: Their label originates from this one cell
- Thus, each labelled region has a master cell

Label Init: Lower/Right values are larger

Labelled result
M = master cell
Master cells: Label propagation

- If master cell changes label, all slave cells can change label
- Hence: Always gather current label from master cell!
- Purpose: Commonly labelled regions flip “at once”.

Pass 0: Three regions: Masters $M_n$, Slaves $S_n$

Pass 1: Region $M_0$ "captures" Masters $M_1$, $M_2$

Pass 2: Master cell lookup makes $S_0$'s and $S_1$'s flip!
Pseudo-Code: Simple Algorithm

// Step I - Label Init
for (all pixels) {
    pixel.label = encodeLabel(pixel.x, pixel.y);
}

// Step II - Propagate Labels
while (AnyLabelChanges) {
    for (all pixels) {
        for (all directions) {
            neighborLabel = gather(neighbor, direction);
            pixel.label = max(pixel.label, neighborLabel);
        }
    }
}
Pseudo-Code: Optimized Algorithm

// Step I - Label Init
for (all pixels)
    pixel.label = encodeLabel(pixel.x, pixel.y);
// Precalculate links
precomputeLinks();
// Step II - Propagate Labels
while (AnyLabelChanges) {
    for (all pixels) {
        for (all directions) {
            // Use max-gather
            neighborLabel1 = gather(neighbor, direction);
            neighborLabelMax = gather(neighbor, pixel.maxgather(direction));
            pixel.label = max(pixel.label, neighborLabel1, neighborLabelMax);
            // Master/Slave
            if (pixel.label != pixel.originalLabel) {
                masterRef = decodeLabel(pixel.label);
                pixel.label = max(pixel.label, masterRef.label); }}}}
Implementation
Implementation: Image Storage

- Input: RGBA, 8 bit
Implementation: Label Storage

- 32 bit for x and y
- Max width: 65535
- Max height: 65535
- Label ordering: upper left << lower right
- $L = x \times \text{width} + y$ (!)
- 3D version: 8/10 bit for x, y and z
Implementation: Links Storage

- All directions stored in global memory
- Line-interleaving ensures memory coalescing during links precomputation & label propagation
Implementation: Execution Configuration

- Block Size = (multiple of 32, 1)
- Extra horizontal block for odd-width images
- Exact number of vertical blocks
- Thread config fits image, label and links processing
Results: Simple 1-gather

- Only 1-gather
- Simple and works, but: SLOW!
- Interesting: "Tug-of-war" in lower part of image, until a much larger label from right (large x component) comes along
Results: Master/Slave Principle

- Already-connected regions switch at once, see e.g. video's ending
Results: Links & Master/Slave

- Pre-linked regions switch a lot faster
Example of 8-connectivity

- 8-connectivity: Links in 8 directions are generated and used.
Results: Input Images

- Used in CUDA TopCoder challenge
Impact of Links & max-gather

- Iterations vs. Gather Lengths (pixels)
- Time vs. Gather Lengths (pixels)

- 100x300
- 1Kx768
- 4Kx4K

- 4-Conn
- 8-Conn
Impact of Master/Slave

![Graphs showing impact of Master/Slave]

- **Iterations**
  - 100x300
  - 1Kx768
  - 4Kx4K

- **Time**
  - 100x300
  - 1Kx768
  - 4Kx4K

- **Gather Lengths (pixels)**
  - 0 to 600

- **Connectivities**
  - 4-Conn
  - 8-Conn
Extension to 3D

- Extend algorithm to 3D (cells = voxels)
- Choice of connectivity scheme
- Labels are now a function of x, y, z
- Labels can be converted to and from 3D coordinates
- 8bit x, y, z -> RGB 8bit
3D Connectivity

- Choice of connectivity scheme from three building blocks:
Results: 3D volume (256x256x100)

1-Gather & Master/Slave
Results: 3D volume (256x256x100)

Max-Gather

CU_6CON_3D-MS  Slice 0  ViewMode: Image

Steps: Label 0  Gather: 1  Total: 1
Results: Typical execution times

- Fast enough for video processing!
- 3D volume of 256x256x100: 1500 ms
- Fast enough for interactive connectivity experiments
- Shmem not yet utilized!

Run on Tesla C2050, includes GPU memory transfers

<table>
<thead>
<tr>
<th>Image</th>
<th>Kernel</th>
<th>Gather</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100by300</td>
<td>CU_4CON</td>
<td>32</td>
<td>4.04</td>
</tr>
<tr>
<td>100by300</td>
<td>CU_8CON</td>
<td>64</td>
<td>2.44</td>
</tr>
<tr>
<td>1Kby768</td>
<td>CU_4CON</td>
<td>64</td>
<td>7.48</td>
</tr>
<tr>
<td>1Kby768</td>
<td>CU_8CON</td>
<td>128</td>
<td>10.78</td>
</tr>
<tr>
<td>4Kby4K</td>
<td>CU_4CON</td>
<td>256</td>
<td>343.84</td>
</tr>
<tr>
<td>4Kby4K</td>
<td>CU_8CON</td>
<td>128</td>
<td>356.43</td>
</tr>
<tr>
<td>ctHead</td>
<td>CU_6CON_3D</td>
<td>128</td>
<td>1499.43</td>
</tr>
</tbody>
</table>
Summary

- Links Precomputation from static connectivity is highly beneficial for label propagation.
  - Surprise: Less than maximal gather lengths are just as usable!
- Completely data-parallel algorithm (parallelization over all pixels, no atomic operations)
- Current implementation (gmem-based) already has real-time 2D performance
Future Work

- Efficient usage of shared memory (prototypes exist)
- Label List generation (based on data compaction)
- Distance Field computation might also benefit from Links
Thank you!
Additional Material
Label lists (Sketch)

- **Q**: How can I extract a list of all discovered regions?
- Step 1: Each region has one master cell. Isolate all cells that have retained their own label!
- Step 2: With list of master cells and their labels, each region's cells can be extracted by filtering for that label.
- Both steps can be solved by Data Compaction! (e.g. HistoPyramids, or Scan)
- Future Work!