A GPU Implementation of Connected Component Labeling

Sean M. O'Connell, soconn1@lsu.edu

Abstract

The connected component labeling algorithm, central to many image processing applications, can be efficiently implemented on the GPU. This work demonstrates a GPU framework suitable for real-time image processing applications and potentially for high resolutions images, a task currently ill-suited for CPU based methods. The implementation utilizes the OpenGL API and the GLSL shading language. Novel approaches are taken in order to solve several steps of connected component labeling on the GPU. Drawbacks and limitations are presented throughout the paper. The performance is measured against a variety of benchmarks and finally compared to the popular OpenCV library.

1. Introduction

Connected component labeling is an algorithm used in computer vision applications that allows for the extraction of pertinent features from binary images. These features are extracted and uniquely identified on the basis of pixel connectivity, which is defined by a specific heuristic. While often referred to by other names, such as region labeling, region extraction, blob extraction, or blob discovery, connected component labeling's sole purpose is to establish connectivity relationships between pixels. The process takes a binary image and groups connected pixels into equivalence classes in order to uniquely identify each segmented object. This is crucial in many areas of computer vision that require extraction of individual blobs or objects, such as in the processing of medical images, aerial photography, video surveillance, and motion tracking applications.

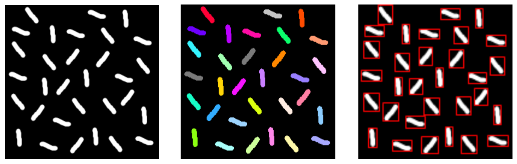


Figure 1.1: Example of connected component labeling.

While many CPU based novel methods exist for solving this connected component labeling, namely [1], [2], [3] and [10], we are motivated to consider a GPU implementation for the following reasons:

- GPUs have been widely adopted for use in image and video processing applications due to tremendous speed increases that can be had from the parallel architecture.
- No GPU implementation of connected component labeling currently exists at the present time.

2. Existing GPU techniques

At the present time only two related works can be found pertaining to a GPU implementation of connected component labeling, namely Fung's method from GPU Gems 2 [11] for tracking a single object and Hansen's gpuTracker software [14] for blob tracking. Neither of these methods is a full implementation of connected component labeling since both only produce rough approximations of extracted objects via *centroids*.

GPU Gems 2. Fung's method assumes the input image contains a single object to be tracked and only produces a single *centroid* (x, y) value. While only requiring 3 rendering passes, the efficiency is less than desirable since given an NxM image, it performs N number of texture samples in a single pass for a single pixel. This occurs for all M rows of the image. In the final pass, M additional texture samplings occur.

Figure 2.1: Fung's method computing the centroid (red) of the detected hand.

The centroid is found by calculating 3 *moments* for the entire iamge and then computing M_{10}/M_{00}, and M_{01}/M_{00} as shown below in figure 2.2.

$$M_{00} = \sum_{x=0}^{x=W} \sum_{y=0}^{y=H} E(x,y)$$

$$M_{10} = \sum_{x=0}^{x=W} \sum_{y=0}^{y=H} xE(x,y)$$

$$M_{01} = \sum_{x=0}^{x=W} \sum_{y=0}^{y=H} yE(x,y)$$

Figure 2.2: Fung's moment equations.

Fung's method is ill-suited for the purpose of connected component labeling since it (a) only detects single objects, (b) does not label individual pixels and (c) performs expensive operations in the fragment shader.

GPUTracker. Hansen's gpuTracker, like Fung's method, computes the centroid of detected objects, but it also able to extract multiple blobs. Vertical and horizontal axis lengths are computed for every pixel in the image that is contained with a blob. The pixels with the largest axis lengths are classified as centroids.

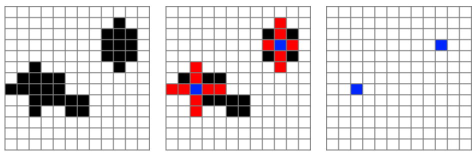


Figure 2.3: (Left) Original image is processed to find longest axis lengths (middle). (Right) Final results consist of centroids.

GPUTracker's implementation requires in excess of 800 texture samplings per pixel in the fragment shader to compute axis lengths, which is far from desirable. Furthermore, once the centroids are found, a geometry shader is used to locate and extract the values. Geometry shaders have yet to be efficient on current GPU architectures and so proves to be an for processing the results. Like Fung's method, gpuTracker does not label each individual pixel in the image.

3. Implementation overview

This work's GPU connected component labeling implementation utilizes a 3x3 connectivity kernel and produces (a) an output image correctly labeled pixels based on connectivity and (b) a set of bounding boxes for each connected component. The process consists of several discrete processing stages. First, each pixel in the input image is assigned a unique ID containing (x, y) information for their particular 2D location in the image. Next, a process similar to run-length encoding [3] is performed to create series of connected pixels down each column of the image. Each of these *vertical runs* is then merged with adjacent ones across the image, from right to left, to form the final equivalence classes, which produces a set of connected components with unique IDs. Finally the results are downloaded from the GPU to the CPU. These steps are listed below.

- Compute vertical runs
- Column processing
- Post column processing
- Compute bounding boxes
- Retrieve results

The OpenGL API and GLSL shading language are utilized in this work and all processing is performed via the processing of vertices and fragment rasterization. No general purpose libraries are used, such as NVIDIA's CUDA, ATI's Stream SDK, or OpenCL.

Shaders. OpenGL vertex and fragment shaders are used throughout this implementation. Various stages of this work's algorithm operate primarily on vertices in the vertex shader, while other stages perform all processing in the fragment shader. A variety of different shaders are used for each distinct stage, as well as different shaders within sub step of each stage.

Textures. We use several two texture formats in this work for storing pixel ID values as well as bounding box information. For the pixel IDs the GL_LUMINANCE32F format was used. Only the red channel is utilized for storing pixel IDs. This limits us to utilizing only the 23-bit mantissa portion of each floating point number since this work stores (x, y) information in each value and also relies on blending operations. More details on this are discussed in chapter 4. But it is worth noting here that due to this limited number of usable bits, this work is only able to process image resolutions up to 2047x2047. The GL_ALPHA32F format would have been preferable for storing pixel IDs since this format consumes 25% of the memory, but testing showed the NVIDIA 8 and 9 series GPUs to be inefficient at rendering to single channel alpha texture targets.

We also utilize the GL_RGBA16F texture format for storing bounding boxes, where each (x1, y1, x2, y2) bounding box coordinate is stored in the 4 RGBA channels.

Limitations. This work does have a few limitations and drawbacks due to the current generation of GPU architectures as well as assumptions made in this work's algorithm. First, current GPUs do not support blending operations on integer texture attachments, which is why we chose the GL_ALPHA32F format. If blending operations on the GL_ALPHA32UI texture formats are eventually supported, then image resolutions greater than 2047x2047 could be processed. Another architectural drawback is the excessive number of state changes that occur during processing, namely due to the use of the ping-pong technique [12] and the binding and unbinding of different shaders. However, it is shown that this work is still able to attain reasonable frame rates.

The second limitation is this work's inability to properly classify certain connected component shapes and therefore produces N+M connected components when only N are present. Since this only occurs under a very specific condition, is it therefore not detrimental to the usability of this work. More details on this matter are discussed in chapter 5.

Moving forward. We now examine each of the different stages of this work's implementation. Only a few of the OpenGL specifics are discussed, but every stage's GLSL vertex and fragment shaders are shown in their entirety. We begin the discussion with the *computing vertical runs* stage.

4. Computing vertical runs

In order to form connected vertical runs, ID values are first assigned to all foreground pixels in the input image. Each ID stores a pixel's (x,y) location within the image, thereby assigning the lowest ID to pixel in the top left corner of the image, and the highest ID to the bottom right corner. In this work, we encode the location in a 32-bit floating point value, specifically the lower 23 bit mantissa portion. The (x, y) coordinates are stored in the first 22 bits, where x is placed in the lower 11 bits, and y in the next set of 11 bits. This scheme was chosen over a 1D index since it provides an easy way to decode a pixel's location within the image without knowing the image's dimensions.

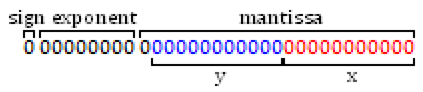


Figure 4.1: 32 bit floating point memory layout of (x, y) locations.

The processing is accomplished by rendering a full screen quad to the frame buffer and executing a fragment shader which calls the functions shown in figure 4.3. In the encode function, we increment each (x, y) component by 1 in order to differentiate between background pixels with IDs of 0 and a foreground pixel located at (0, 0).

```
float encode(ivec2 position)
{
    int ret = (position.x + 1) | ((position.y + 1) << 11);
     return float(ret);
}

ivec2 decode(float value)
{
     int i = int(value);
     int x = (i & 0x000007FF) - 1;
     int y = ((i & 0x003FF800) >> 11) - 1;
     return ivec2(x, y);
}
```

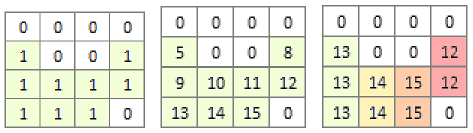
Figure 4.2: GLSL functions for encoding and decoding (x,y) locations and ID values.

After all pixels are assigned IDs, every connected vertical strip of foreground pixels is assigned the bottom most pixel's ID that can be reached within the same strip. As stated in chapter 3, this operation is similar to run length encoding [3]. First, each pixel samples its neighbor to the south's ID value. Then in each successive pass, the stored ID value at each pixel is used to retrieve the location for the next ID value. By executing this operation log(h) times, where h is the height of the image, all vertical connected strips are correctly assigned the bottom most ID value that can be reached within the same column. The pixel at the bottom of a vertical run is termed the *root*, all other pixels are the *children*. The fragment shader for this operation is shown in figure 4.3. A uniform variable, *passIndex*, is used to specify which pass is being executed. If passIndex is equal to 0, then each foreground pixel samples its south adjacent neighbor. For every other pass, each pixel uses its current ID to sample the next value to be assigned to itself. Note that only in the first pass do we check for background pixels. After this initial pass, there is no need to check further.

```
uniform sampler2D tex;
uniform int passIndex;

void main()
{
     ivec2 uv_coord = ivec2(gl_FragCoord.xy);
     float pixel = texelFetch2D(tex, uv_coord, 0).x;
     if (passIndex == 0) {
          ivec2 uv = uv_coord + ivec2(0,1);
          float neighbor = texelFetch2D(tex, uv, 0).x;
          gl_FragColor.x = neighbor * clamp(pixel, 0, 1);
     } else {
          float next = texelFetch2D(tex, decode(pixel), 0).x;
          gl_FragColor.x = next;
     }
}
```

Figure 4.3: Fragment shader for generating vertical runs.

Figure 4.4 demonstrates this entire process. In this example, we do not label pixels using the scheme described above, but rather assign them their 1D index within the image for the sake of readability.



0	0	0	0
1	0	0	1
1	1	1	1
1	1	1	0

0	0	0	0
5	0	0	8
9	10	11	12
13	14	15	0

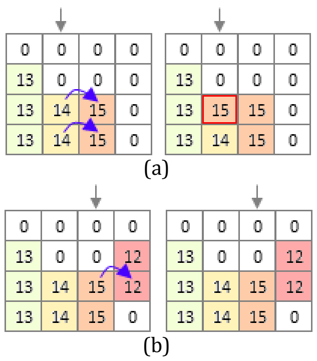
0	0	0	0
13	0	0	12
13	14	15	12
13	14	15	0

Figure 4.4: An example of vertical runs being formed from the original image(left).

5. Column processing

After all vertical runs are formed, we sequentially merge each run from right to left across the image. We move from right to left in order to assign each connected component the highest ID value possible (i.e. the *root* pixel nearest the bottom right corner of the image). For each column in the image we perform 3 steps: *gather neighbor, update column,* and *scatter back*. For the first and last step, each pixel within the column is treated as a vertex and processing is performed in the vertex shader. The *update column* step processes every pixel within the column in image space, hence operations are performed in the fragment shader. The overall idea is to merge adjacent vertical runs by linking the *roots*, where a root may change its ID if an adjacent vertical run has a higher ID value. This is accomplished via scattering in the vertex shader. By enabling OpenGL blending and setting the operation to GL_MAX, a scattered ID value is only rasterized to the frame buffer is it is greater than the current value already present at that location. If we chose to use the GL_MIN blending operation, then pixel IDs of 0 would need to be checked and discarded at various times throughout processing.

Gather neighbor. First, every pixel within the current column samples its right adjacent neighbor. If the neighbor is a foreground pixel, the ID is scattered to the current pixel's root. Since OpenGL blending is enabled as previously described, the decision to write this ID is handled automatically (i.e. the greater of the current root pixel value and the scattered ID is written). In figure 5.1, (a) demonstrates when a root's ID is replaced and (b) shows when it is not.



0	0	0	0
13	0	0	0
13	14	15	0
13	14	15	0

0	0	0	0
13	0	0	0
13	15	15	0
13	14	15	0

(a)

0	0	0	0
13	0	0	12
13	14	15	12
13	14	15	0

0	0	0	0
13	0	0	12
13	14	15	12
13	14	15	0

(b)

Figure 5.1: In (a) the root ID 14 is replaced with 15. In (b) root ID 15 retains its value after the gather neighbor step.

Again, a vertex is submitted for every pixel in the current column and and processing is performed in the vertex shader. Figure 5.2 lists the vertex shader for the *gather neighbor* step. The fragment shader is trivial and simply writes the incoming scatterID to the frame buffer .

```
uniform sampler2D tex;
flat out float scatterID;

void main()
{
     ivec2 uv_coord = ivec2(gl_Vertex.xy);
     ivec2 uv = uv_coord + ivec2(1, 0);
     float neighbor = texelFetch2D(tex, uv, 0).x;
     float curPixel = texelFetch2D(tex, uv_coord, 0).x;
     ivec2 root_xy = decode(curPixel);
     scatterID = neighbor;
```

```
        gl_Position = gl_ModelviewProjectionMatrix *
                          vec4(vec2(root_xy), 0.0, 1.0);
    }
```

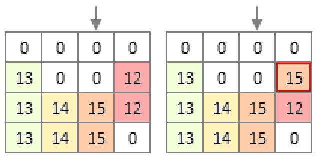
Figure 5.2: Fragment shader for the gather neighbor step.

Update column. Next, each foreground pixel in the current column is updated to reflect any changes made to its root. This is necessary so that subsequent columns sample correct IDs during the *gather neighbor* step. A single line is drawn down the image column and each pixel is processed in the fragment shader. A simple texel lookup is performed by decoding each pixel's ID to retrieve the (x, y) location of the root. The root's value is then copied to each child. The fragment shader for this operation is shown in figure 5.3.

```
uniform sampler2D tex;

void main()
{
    ivec2 uv_coord = ivec2(gl_FragCoord.xy);
    float curPixel = texelFetch2D(tex, uv_coord, 0).x;
    ivec2 root_xy = decode(curPixel);
    gl_FragColor.x = texelFetch2D(tex, root_xy.xy, 0).x;
}
```

Figure 5.3: Fragment shader for update column step.

Scatter back. Last, each pixel in the current column scatters its ID to the previously processed adjacent neighbor's root location. This is necessary in instances where the previous column contains IDs less than the current, as shown in figure 5.4.

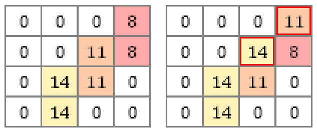


0	0	0	0
13	0	0	12
13	14	15	12
13	14	15	0

0	0	0	0
13	0	0	15
13	14	15	12
13	14	15	0

Figure 5.4: Root ID 8 is scattered to the root location of ID 10 during the scatter back step.

Similar to the *gather neighbor* step, a vertex is submitted for every pixel in the column and processed in the vertex shader. But here, for every root ID in a previous column that is modified, we mark it by setting its depth value to 1. This is necessary since these columns will not be visited again for the duration of the *column processing* stage. Furthermore, multiple scatter backs can occur causing a set of *roots* to change as well as creating a chain where each links to the next, as is the case in figure 5.5.



0	0	0	8
0	0	11	8
0	14	11	0
0	14	0	0

0	0	0	11
0	0	14	8
0	14	11	0
0	14	0	0

Figure 5.5: Example where multiple root IDs are modified during the scatter back step. (Left) original image. (Right) image after process columns stage.

The depth information will be utilized later in the *post column processing* stage. We also call glDepthMask to enable and disable writing to the depth buffer during. We do not want to overwrite any values in the depth buffer during the previous 2 steps, hence depth writes are disabled right after *scatter back*. The vertex shader is shown in figure 5.6.

```
uniform sampler2D tex;
flat out float scatterID;

void main()
{
    ivec2 uv_coord = ivec2(gl_Vertex.xy);
    float curPixel = texelFetch2D(tex, uv_coord, 0).x;
    ivec2 uv = uv_coord + ivec2(1,0);
    float neighbor = texelFetch2D(tex, uv, 0).x;
    ivec2 xy = decode(neighbor);
    scatterID = curPixel;
    gl_Position = gl_ModelviewProjectionMatrix *
                      vec4(vec2(xy), 1.0, 1.0);
}
```

Figure 5.6: Vertex shader for scatter back step.

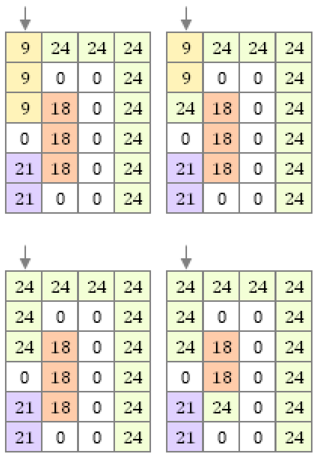
Ping-pong. During each of the 3 previously discussed steps, we must ping-pong [12] between read and write textures, which incurs a penalty due to GPU state changes. Furthermore, since each step is distinct, the ping-pong is performed 2x for each column in the image. Additional state changes include binding and unbinding of each step's fragment and vertex shaders. While intuitively this may seem quite expensive and become a severe bottleneck, we are still able to attain real-time performance.

We only need to ping-pong between read and write textures 2x for all 3 steps since changes made during the *scatter back* are not necessarily required during subsequent processing of columns in the *gather neighbor* step, nor will they be overwritten. Hence, 2 ping-pongs and 3 shader bindings are performed for each column in the image, as shown in the code below.

```
glDrawBuffer(GL_COLOR_ATTACHMENT_0);
glBindTexture(texture_1);
for (int col = image_width - 2; col >= 0; col--)
{
        glUseProgram(gatherNeighborShader);
        RenderVertices();

        glDrawBuffer(GL_COLOR_ATTACHMENT_1);
        glUseProgram(updateColumnShader);
        glBindTexture(texture_0);
        RenderLine();

        glDrawBuffer(GL_COLOR_ATTACHMENT_0);
        glUseProgram(scatterBackShader);
        glBindTexture(texture_1);
        glDepthMask(GL_TRUE);
        RenderVertices();
        glDepthMask(GL_FALSE);
}
```

Figure 5.7: Rendering code for process columns stage.

Limitations. As previously stated, there is a specific scenario where the *column processing* stage inaccurately labels connected components. This occurs in the instance shown below in figure 5.8.



9	24	24	24
9	0	0	24
9	18	0	24
0	18	0	24
21	18	0	24
21	0	0	24

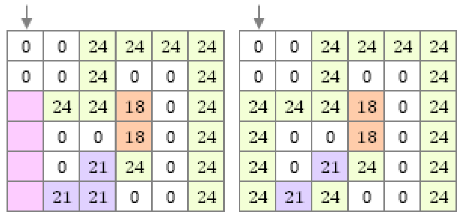
9	24	24	24
9	0	0	24
24	18	0	24
0	18	0	24
21	18	0	24
21	0	0	24

24	24	24	24
24	0	0	24
24	18	0	24
0	18	0	24
21	18	0	24
21	0	0	24

24	24	24	24
24	0	0	24
24	18	0	24
0	18	0	24
21	24	0	24
21	0	0	24

Figure 5.8: (Top left) Original image goes through gather neighbor step (top right), then update column (bottom left) and finally scatter back (bottom right).

The problem here is that vertical run ID 21 does not correctly merge with the rest of the image. All connectivity information between ID 21 and ID 18 is lost since ID 24 is scattered to the *root* of vertical run 18, hence overwriting the 21. While this may seem detrimental, if at any point later in the processing connectivity is reestablished between ID 24 and 21, then this error is corrected, as shown in figure 5.9.



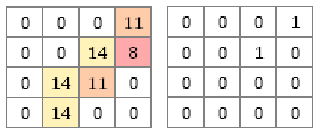
0	0	24	24	24	24
0	0	24	0	0	24
	24	24	18	0	24
	0	0	18	0	24
	0	21	24	0	24
	21	21	0	0	24

0	0	24	24	24	24
0	0	24	0	0	24
24	24	24	18	0	24
24	0	0	18	0	24
24	0	21	24	0	24
24	21	24	0	0	24

Figure 5.9: Correct connectivity is restored when column 0 is processed and performs the scatter back step.

6. Post column processing

At this point, all roots whose IDs were modified via the *scatter back* step, have a value of 1 at their corresponding location in the depth buffer, as shown in figure 6.1.

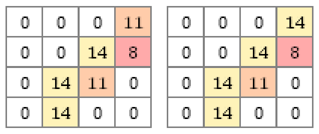


0	0	0	11
0	0	14	8
0	14	11	0
0	14	0	0

0	0	0	1
0	0	1	0
0	0	0	0
0	0	0	0

Figure 6.1: (Right) depth buffer after image (left) is processed.

Update roots. Each of these roots must iteratively update their ID until the decoded (x, y) location is equal to the decoded (x, y) of the new ID value. To accomplish this, a full screen quad is rendered to the frame buffer while setting the depth comparison function to GL_EQUAL and setting the depth value of each quad vertex to 1. This alleviates the fragment shader from processing any pixels whose depth values are not set to 1 (i.e. set to 0). Figure 6.2 shows the result after this operation and the fragment shader is listed in figure 6.3.



0	0	0	11
0	0	14	8
0	14	11	0
0	14	0	0

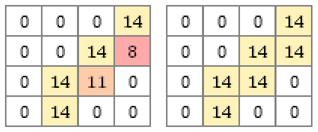
0	0	0	14
0	0	14	8
0	14	11	0
0	14	0	0

Figure 6.2: Roots whose depth value is 1 (left) are iteratively updated to reflect their final root IDs (right).

```
uniform sampler2D tex;

void main()
{
    ivec2 uv_coord = ivec2(gl_FragCoord.xy);
    float curPixel = 0.0;
    while (true) {
        curPixel = texelFetch2D(tex, uv_coord, 0).x;
        ivc2 xy = decode(curPixel);
        if (all(equal(xy, uv_coord))) break;
        uv_coord = xy;
    }
    gl_FragColor.x = curPixel;
}
```

Figure 6.3: Fragment shader for the update roots step.

Update children. Next, all *child* pixels within each vertical run must be updated to reflect any changes made to their roots' IDs. Results are shown below in figure 6.4.



0	0	0	14
0	0	14	8
0	14	11	0
0	14	0	0

0	0	0	14
0	0	14	14
0	14	14	0
0	14	0	0

Figure 6.4: All pixels with each vertical run (left) are updated to reflect their root IDs.

A full screen quad is rendered to the frame buffer and all processing is performed in the fragment shader, shown in figure 6.5.

```
uniform sampler2D tex;

void main()
{
    ivec2 uv_coord = ivec2(gl_FragCoord.xy);
    float curPixel = texelFetch2D(tex, uv_coord, 0).x;
    ivec2 xy = decode(curPixel);
    gl_FragColor.x = texelFetch2D(tex, xy.xy, 0).x;
}
```

Figure 6.5: Fragment shader for update children step.

7. Computing bounding boxes

At this stage all pixels within the image have been labeled and correctly grouped into their unique groups or connected components. Now, we compute bounding boxes for each of these connected component by once again utilizing the scattering capabilities of the vertex shader as well as the OpenGL blending operation. First, a vertex is submitted to the GPU for every pixel in the image. The vertex shader decodes the pixel's root's (x, y) location and scatters that pixel's (x, y) values to the root. In the fragment shader, every incoming (x, y) value is written to the 4 channels of the bound GL_RGBA16f texture like so:

- red: image_width – 1 - x
- green: image_height – 1 - y
- blue: x
- alpha: y

With OpenGL blending enabled and the operation set to GL_MAX, each root pixel will automatically compute its entire bounding box by accumulating the (x, y) positions of every one of its child pixels. The blue and alpha channels store the maximum (x, y) values of the bounding box. we invert the incoming (x, y) values by subtracting them from the image's dimensions and storing the result. An example of this operation is shown below in figure 7.1.

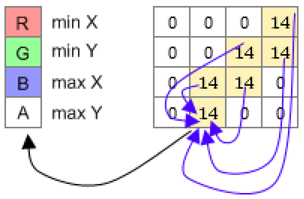


Figure 7.1: An example of bounding box computation via vertex scattering.

The vertex and fragment shaders for this process are shown below in figures 7.2 and 7.3.

```
uniform sampler2D tex;
flat out vec2 xy;
flat out float scatterID;

void main()
{
    ivec2 uv_coord = ivec2(gl_Vertex.xy);
    float curPixel = texelFetch2D(tex, uv_coord, 0).x;
    vec2 run_xy = decode(curPixel);
    scatterID = curPixel;
    xy = uv_coord;
    vec4 out_pos = vec4(run_xy, 0.0, 1.0);
    gl_Position = gl_ModelviewProjectionMatrix *
                    vec4(run_xy, 0, 1);
}
```

Figure 7.2: Computing bounding boxes vertex shader.

```
uniform sampler2D tex;
flat in vec2 xy;
flat in float scatterID;

void main()
{
    ivec2 texSize = textureSize(tex, 0);
    float x1= float(texSize.x - 1) – xy.x;
    float y1 = float(texSize.y- 1) – xy.y;
    float x2 = xy.x;
    float y2 = xy.y;
    gl_FragColor = vec4(x1, y1, x2, y2);
    }
}
```

Figure 7.3: Computing bounding boxes fragment shader.

8. Retrieving components

At this point, we have a set of textures that (a) store correct ID labels for every pixel in the image and (b) bounding box information for each connected component. To retrieve this information, we first must move and store all unique *root* IDs and bounding box information to the upper left corner of their textures. This allows for much more efficient and straightforward data transfer from the GPU to CPU by enabling us to only download a given NxN block of image data via calls to glReadPixels. In order to calculate N, we must first know the number of unique IDs in the image.

Occlusion Query. In order to retrieve the number of unique IDs in the image, we make use of the OpenGL occlusion query extension and a fragment shader that discards all pixels in the image whose ID value does not correspond to their (x, y) locations. Only *root* pixels will pass this test and hence, the occlusion query will return the number of *roots* or unique ID values. If M pixels pass the test, then N = ceil(sqrt(M)). The fragment shader for this operation is shown in figure 8.1.

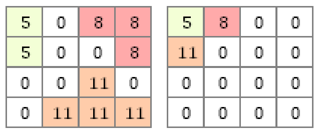
```
uniform sampler2D tex;

void main()
{
    ivec2 uv_coord = ivec2(gl_FragCoord.xy);
    float curPixel = texelFetch2D(tex, uv_coord, 0).x;
    ivec2 xy = decode(curPixel);
    if (all(equal(xy, uv_coord)))
        gl_FragColor.x = curPixel;
    else discard;
}
```

Figure 8.1: Fragment shader for occlusion query pass.

Stencil Routing. Next, we utilize stencil routing in order to move all *root* IDs and their associated bounding box values to the top left corner of the textures. We first initialize an NxN region of the stencil buffer with a series of increasing 8bit values, ranging from 1 to M. The stencil test is then enabled via the following OpenGL calls:

- glEnable(GL_STENCI_TEST)
- glStencilFunc(GL_EQUAL, 1, 0xFF)
- glStencilOp(GL_DECR, GL_DECR, GL_DECR)

These calls result in each value in the stencil buffer being decremented by 1 each time a fragment is written to a location within the NxN region, as well as only rasterizing pixels to the frame buffer where the corresponding location in the stencil buffer is 1. Furthermore, glPointSize(N) is called in order to rasterize each vertex to the entire NxN region of the frame buffer and stencil buffer, as required by [13]. An example of this process is shown below in figure 8.2.



5	0	8	8
5	0	0	8
0	0	11	0
0	11	11	11

5	8	0	0
11	0	0	0
0	0	0	0
0	0	0	0

Figure 8.2: (Left) original image's root IDs are scattered to the top left corner (right) via stencil routing.

Since we want to move *root* IDs and bounding box information, we bind both corresponding textures to the frame buffer during rendering (i.e. MRT). The vertex and fragment shaders are shown below in figures 8.3 and 8.4.

```
uniform sampler2D tex;
uniform sampler2D boundsTex;
uniform vec2 stencilCenter;
flat out float scatterID;
flat out vec4 bounding box;

void main()
{
    ivec2 uv = ivec2(gl_Vertex.xy);
    scatterID = texelFetch2D(tex, uv 0).x;
    bounding box  = texelFetch2D(boundsTex, uv, 0);
    ivec2 xy = decode(scatterID);
    if (all(equal(uv_coord, xy)))
        gl_Position = gl_ModelviewProjectionMatrix *
                      vec4(stencilCenter, 0.0, 1.0);
    else
        gl_Position = gl_ModelviewProjectionMatrix  *
                      vec4(-stencilCenter, -1.0, 0.0);
}
```

Figure 8.3: Vertex shader bound during stencil routing.

```
flat in float scatterID;
flat in vec4 box;

void main()
{
    gl_FragData[0].x = scatterID;
    gl_FragData[1] = bounding box;
}
```

Figure 8.4: Fragment shader bound during stencil routing.

Downloading results. After stencil routing has been performed, the results are retrieved via calls to glReadBuffer and glReadPixels. First, each unique connected component ID is retrieved, and then their corresponding bounding boxes. Since all the data is stored in the top left NxN corner of the textures, data transfer is minimized.

Moving forward. While the bulk of this work's implementation thus far has been sequential (i.e. the *process columns* stage), we move on to discuss a method to speed up the processing of an input image by dividing it up into a series of N vertical blocks and processing each in parallel.

9. Parallel block processing

By dividing the image into N vertical blocks, the overall performance can be greatly improved. Here, each block is processed independently during the *process columns* and *post process columns* stages. Afterwards, the results in each block must be merged. We do this by iteratively processing pixels on either side of each border. Each pixel samples IDs on the opposite border and then scatters its ID or the neighbor's ID to an appropriate location. If the neighboring ID is less than the current pixel's ID, the current pixel scatters its ID to the neighbor's root location, and vice versa if the neighboring ID is greater than the current pixel's. Once again, vertices are submitted to the GPU for each pixel along the block borders and we mark vertices corresponding to pixels on the left side of a border with z = 1, and pixels on the right with z = -1. This enables us to quickly calculate the (x, y) location of a border pixel's neighbor. This process is performed log(N) times. The vertex shader is shown in figure 9.1.

```
uniform sampler2D tex;
flat out float scatterID;

void main()
{
    ivec2 uv = ivec2(gl_Vertex.xy);
    float curPixel = texelFetch2D(tex, uv, 0).x;

    ivec2 n_uv = uv + ivec2(gl_Vertex.z, 0);
    float neighbor = texelFetch2D(tex, n_uv, 0).x;

    ivec2 xy = decode(curPixel);
    scatterID = neighbor;
    gl_Position = gl_ModelViewProjectionMatrix *
                    vec4(vec2(xy), 1.0, 1.0);
}
```

Figure 9.1: Parallel block processing vertex shader.

After we perform this process, the *post column processing* step must be performed a second time to propagate changes made to any *root* IDs. With this optimization in mind, we now move forward to analyze and discuss this work's performance results.

10. Results and discussion

The primary test hardware used in this work consists of an AMD Phenom II X2 550 @ 3.1Ghz, 4GB DDR2 RAM, and an under clocked NVIDIA 9800GT 512MB video card. The video card clock speeds are 550Mhz core, 900Mhz memory, and 1375Mhz shader clock. A Tesla S1070 was also utilized for performance testing, with the results shown later in this chapter.

Baseline performance. The baseline performance of this work for a 512x512 input image consisting of 31 distinct blobs, see figure 1.1, is shown below in figure 10.1. We were able to achieve average execution times of 9ms and frame rates of 112fps.

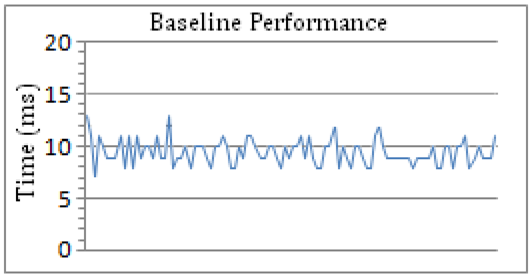
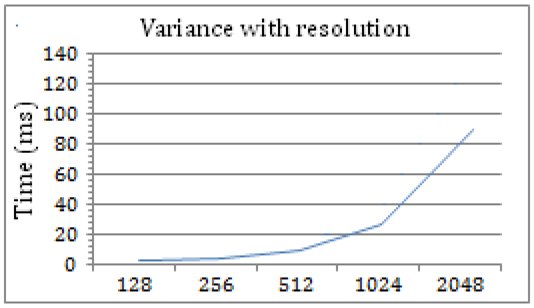
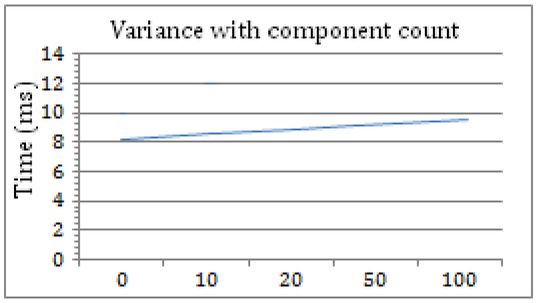


Figure 10.1: Baseline performance graph

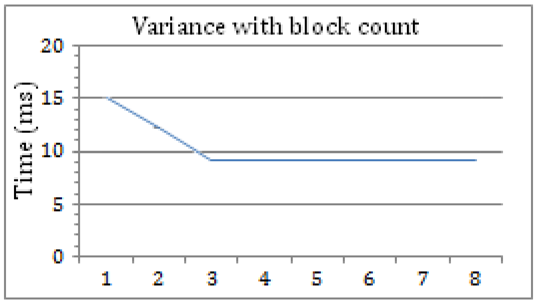
Benchmarks. We tested the performance of this work at varying image resolutions as well as a varying number of connected components. In figure 10.2 (a) we see that execution times increase non-linearly at resolutions above 512x512. But processing a 1024x1024 image still requires only 27ms, thus achieving on average 37fps. In figure 10.2 (b) we see that there appears to be no direct correlation between performance and the number of connected components. We also found that increasing the number of blocks for parallel processing provided no benefit past N = 4, as shown in figure 10.2 (c).



(a)



(b)



(c)

Figure 10.2: Performance graphs

Tesla S1070. Unfortunately the Tesla S1070 produced identical results to the under clocked NVIDIA 9800GT video card, even though the S1070 consists of 2x the number of processing cores and has greater clock speeds. This would seem to indicate the existence of a bottleneck related to state changes, rather than vertex or fragment processing. The performance numbers attained for the Tesla S1070 are shown in table 10-1.

Image Resolution	S1070	9800GT
128x128	3ms	3ms
256x256	4ms	4ms
512x512	9ms	9ms
1024x1024	25ms	27ms
2048x2048	90ms	91ms

Table 10-1: Performance comparison between NVIDIA Tesla S1070 and 9800GT.

OpenCV. Finally we compared this work's performance against OpenCV's current connected component labeling method, the Contour Tracing Technique [10]. We found that this work did not exceed OpenCV's performance, but for image resolutions below 1024x1024, the results were approximately the same. The results are listed below in table 10-2.

Image Resolution	OpenCV	GPU
128x128	3ms	3ms
256x256	5ms	4ms
512x512	9ms	9ms
1024x1024	19ms	27ms
2048x2048	70ms	91ms

Table 10-2: Performance comparison between OpenCV and this work.

11. Summary and future work

Acknowledgements

References

[1] Jung-Me Park, Carl G. Looney, Hui-Chaun Chen. *Fast Connected Component Labeling Algorithm Using a Divide and Conquer Technique*. Computer Science Dept University of Alabama and University of Nevada, Reno. 2004.

[2] Quoc Tuan Pham. *Parallel Algorithms and Architectures for Binary Image Component Labeling.* University of California, M.S. 1987.

[3] Kofi Appiah, Andrew Hunter, Patrick Dickinson, Jonathan Owens. *A Run-Length Based Connected Component Algorithm for*

FPGA Implementation. Field-Programmable Technology. 2008.

[4] Louis Bavoil, Steven Callahan, Claudia Silva. *Multi-Fragment Effects on the GPU using the K-Buffer.* Symposium on Interactive 3D Graphics and Games. 2007.

[5] Christopher Oat. *Efficient Spatial Binning on the GPU*. SIGGRAPH Asia. 2008.

[6] Kevin Myers, Louis Bovoil. *Stencil Routed A-Buffer*. SIGGRAPH. 2007.

[7] Gernot Ziegler, Art Tevs, Christian Theobalt, Hans-Peter Seidel. *GPU Point List Generation through Histogram Pyramids*. VMS 2006, GPU Programming. 2006.

[8] Thorsten Scheuermann, Justin Hensley. *Efficient Histogram Generation Using Scattering on GPUs*. ACM I3D 2007 Conference Proceedings. 2007.

[9] Bingsheng He, Naga K. Govindaraju, Qiong Lou, Burton Smith. *Efficient Gather and Scatter Operations on Graphics Processors*. ACM/IEEE Conference on Supercomputing. 2007.

[10] Fu Chang, Chun-Jen Chen, Chi-Jen Lu. *A Linear-Time Component-Labeling Algorithm using Contour Tracing Technique*. Computer Vision and Understanding, V93, I2, pp 206-220. 2004.

[11] James Fung. *Computer Vision on the GPU.* GPU Gems 2, pp 656-658. 2005.

[12] GPGPU Ping-pong technique, http://www.mathematik.tu-dortmund.de/~goeddeke/gpgpu/tutorial.html#feedback2

[13] Stencil Routing, http://www.tatwood.net/articles/25/stencil-routing

[14] GPUTracker: GPU Accelerated Blob Tracking, http://code.google.comm/p/gputracker

Appendix A: Example images