GPU Programming
EE 4702-1
Take-Home Final Examination
5 December - 16:00, 10 December 2009

Please do not discuss this exam with any other students. If you have any question E-mail me or stop by my office Wednesday or later. Do not attempt to find solutions to similar problems, but it’s okay to otherwise seek out references beyond those used in class.

Please drop off solution at my office, slide it under the door if I’m not in. Solutions can also be E-mailed.

Problem 1 ________ (20 pts)
Problem 2 ________ (20 pts)
Problem 3 ________ (20 pts)
Problem 4 ________ (20 pts)
Problem 5 ________ (20 pts)

Alias ____________________________ Exam Total ________ (100 pts)

Good Luck!
Problem 1: (20 pts) In the balls simulation the balls cast shadows on the platform but they do not cast shadows on each other. The `vs_main_bshadow` routine below is a vertex shader that will show the shadows that one ball casts on another. It is called when rendering balls; if there are no shadows then it computes the lighting normally, otherwise it accounts for the lack of light from one or both light sources.

For each ball, the host code finds up to four other balls that might possibly shade it, the eye-space coordinates of these are passed to the vertex shader in `float4` variable `neighbor_ball_X`.

Routine `find_shadow(vertex_e, neighbor_ball_X, light)` determines whether ball `neighbor_ball_X` does indeed leave `vertex_e` in the shadow of light source `light`. That is, if it returns true the vertex is shaded from the light, otherwise it is illuminated.

```cpp
void vs_main_bshadow() {
    vec4 vertex_e = gl_ModelViewMatrix * gl_Vertex;
    vec3 normal_e = normalize(gl_NormalMatrix * gl_Normal);
    float4 neighbor_ball[4] = 
        { neighbor_ball_0, neighbor_ball_1, neighbor_ball_2, neighbor_ball_3 };
    bool shadow_light[2] = {false, false};

    for ( int light=0; light<2; light++ )
        for ( int n = 0; n < 4; n++ )
            bool in_shadow = find_shadow(vertex_e, neighbor_ball[n], light);
            shadow_light[light] |= in_shadow;
            // if ( in_shadow ) break; // Bad idea.

    generic_lighting(vertex_e, gl_Color, normal_e, shadow_light);
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

(a) Modify `generic_lighting` to account for the shadows. *Hint: This is easy.*

```cpp
void generic_lighting(vec4 vertex_e, vec4 color, vec3 normal_e, bool sh[2])
{
    vec4 new_color = vec4( color.rgb * gl_LightModel.ambient.rgb, color.a);
    for ( int i=0; i<2; i++ )
        {
            vec4 light_pos = gl_LightSource[i].position;
            vec3 v_vtx_light = light_pos.xyz - vertex_e.xyz;
            float phase_light = dot_pos(normal_e, normalize(v_vtx_light).xyz);
            vec3 ambient_light = gl_LightSource[i].ambient.rgb;
            vec3 diffuse_light = gl_LightSource[i].diffuse.rgb;
            float dist = length(v_vtx_light);
            float distsq = dist * dist;
            float atten_inv =
                gl_LightSource[i].constantAttenuation +
                gl_LightSource[i].linearAttenuation * dist +
                gl_LightSource[i].quadraticAttenuation * distsq;
            new_color.rgb += color.rgb *
                ( ambient_light + phase_light * diffuse_light ) / atten_inv;
        }
    gl_FrontColor = new_color;
}
```
Problem 1, continued:

(b) Indicate whether each storage type below can be used for `neighbor_ball_X`, and how well that would work.

- [ ] uniform float4 neighbor_ball_0 Possible? Good?
- [ ] attribute float4 neighbor_ball_0 Possible? Good?
- [ ] varying float4 neighbor_ball_0 Possible? Good?

(c) Ordinary execution of the loop nest (the for loops around the call of `find_shadow`) would involve executing 13 branches. Improper use of branches can sometimes hurt shader code performance. Why does this not apply to the loop nest here?

(d) The `vs_main_bshadow` routine includes a commented-out loop exit, labeled “Bad idea.” That loop exit would reduce execution time in CPU code, but in GPU code it would likely have no impact on performance. Why?
Problem 2: (20 pts) For this problem, use the code and description from the previous problem. (This is essentially a continuation of the previous problem.)

(a) Complete the find_shadow routine so that it returns true if vertex_e is blocked from light light_num by the ball at center neighbor_e.

```cpp
bool find_shadow(vec4 vertex_e, vec4 neighbor_e, int light_num) {
    float r = opt_ball_radius;
    // Solution goes here.
}
```
Problem 3: (20 pts) The CUDA routine below computes the vertex positions for a ball based on the ball’s center position (a homogeneous coordinate) and orientation (a quaternion). It reads vertex positions for a unit-radius sphere centered on the origin, and transforms them using a transformation matrix that it computes. The sphere coordinates in \texttt{sphere} are precomputed by the CPU and never change. The vertex coordinates in \texttt{vertices} are used by OpenGL code to render the sphere.

The host code will launch one block per ball, each thread in a block computes vertices for a different part of a ball. The block size determines how many vertices each thread computes. Assume that \texttt{sphere.size} is 1024 or even larger.

```c
__device__ xf_ball_per_block()
{
    const int idx = blockIdx.x; // This is the same for all threads in a block.
    if ( idx >= ball_count ) return;
    const int tid = threadIdx.x;

    float4 ball_position = balls_x.position[idx];
    float4 ball_orientation = balls_x.orientation[idx];

    __shared__ pMatrix transform_local_to_world;

    bool all_compute_matrix = false;
    if ( tid == 0 || all_compute_matrix )
        transform_local_to_world = compute_transf(ball_orientation, ball_position);

    __syncthreads();

    const int thr_per_block = blockDim.x;
    const int my_part = sphere_size / thr_per_block;
    const int start = tid * my_part;
    const int stop = start + my_part;

    for ( int i=start; i<stop; i++ )
    {
        float4 sphere_local = sphere[i];
        float4 sphere_world = transform_local_to_world * sphere_local;
        vertices[ idx * sphere_size + i ] = sphere_world;
    }
}
```

(a) Notice that if \texttt{all_compute_matrix} is false then only thread 0 (in each block) computes the transformation matrix. Explain why setting \texttt{all_compute_matrix} to true would only make a difference when the block size is larger than 32 threads (the warp size).

(b) Access to the global array \texttt{sphere} is not coherent, and so the GPU will make many more trips to memory than are necessary. Explain why the incoherence forces the GPU to make seemingly unnecessary trips to memory.
Problem 3, continued:

(c) Modify the code so access to \texttt{sphere} is coherent, but without modifying what the code does. \textit{Hint: the code will have to access \texttt{sphere} in a different order.}
Problem 4: (20 pts) Answer each question below.

(a) A GPU has \( m \) multiprocessors, and a CUDA application needs to run a launch kernel with \( n \) threads. What range of block sizes will ensure that all multiprocessors are used?

(b) Suppose a GPU has two multiprocessor and a CUDA application needs to launch a kernel of just 30 threads. The threads do not share memory. Explain the difference in performance between launching one block of 30 threads and two blocks of 15 threads each.

(c) Access to CUDA memory takes hundreds of cycles and there is no cache that can reduce that number to only a few cycles. Why does a properly written CUDA program not lose performance despite the long wait for data?

(d) CUDA shared memory has no counterpart in the OpenGL shader language. Explain what shared memory might be used for.
Problem 5: (20 pts) Answer each question below.

(a) The variables below are available to OpenGL vertex shaders. Show how the value for each of these variables is set in host OpenGL code. That is, show a line or two of OpenGL code that provides a value for the variable. Don’t show more than necessary.

- gl_ModelViewMatrix
- gl_Vertex
- gl_LightSource[2].position

(b) Suppose one wanted a strip of triangles, each of a different color, and the strip was to be rendered as a GL_TRIANGLE_STRIP.

- Explain why it’s impossible to give each triangle its own color without using a geometry shader.

- Describe how one could give each triangle its own color with a geometry shader.
(c) Describe what the z-buffer and z test are for.

(d) The equation used to compute lighting is complex, as it must consider at least eight light sources, plus account for the four types of lighting. Why then would the amount of computation needed for texture processing be much larger, even when only one texture is used?