GPU Programming
EE 4702-1
Final Examination
12 December 2013, 15:00–17:00 CST

Problem 1  ____  (25 pts)
Problem 2  ____  (25 pts)
Problem 3  ____  (15 pts)
Problem 4  ____  (10 pts)
Problem 5  ____  (15 pts)
Problem 6  ____  (10 pts)

Alias  ___________________________

Exam Total  ____  (100 pts)

Good Luck!
Problem 1: (25 pts) The code below computes forces between all pairs of a set of balls, similar to the code from the pre-final and Homework 3. Unlike those prior versions, the code below is written for the case where the number of balls (chain_length) is a multiple of the total number of threads.

Let \( c \) denote the number of balls, \( n \) denote the total number of threads (num_threads), and \( g \) denote the grid size (number of blocks). The size of a float4 is 16 bytes. The code runs on an NVIDIA Kepler GPU, which supports requests of size 32, 64, and 128 bytes.

```c
const int tid = threadIdx.x + blockIdx.x * blockDim.x;
const int num_threads = blockDim.x * gridDim.x;
const int a_idx_start = tid;
const int balls_per_thread = chain_length / num_threads;
for ( int i=0; i<balls_per_thread; i++ )
{
    const int a_idx = a_idx_start + i * num_threads;
    const float4 pos_a = d_pos[a_idx]; // Global Access A
    float3 force = make_float3(0,0,0);
    for ( int b_idx = 0; b_idx < chain_length; b_idx++ )
    {
        const float4 pos_b = d_pos[b_idx]; // Global Access B
        force += force_compute(pos_a,pos_b);
    }
    d_vel[a_idx] += delta_t * d_balls[a_idx].mass_inv * force;
}
```

(a) In terms of \( n \), \( c \), and \( g \) compute the amount of data requested and used for the line marked Global Access A.

- Data requested by Global Access A.  
- Data used by A.

(b) In terms of \( n \), \( c \), and \( g \) compute the amount of data requested and used for the line marked Global Access B.

- Data requested by Global Access B.  
- Data used by B.
Problem 2: (25 pts) The code below is similar to the code from the previous problem except that shared memory is used to hold values needed for pos_b. The value of csize is not well chosen given the way the code is written. As before let c denote the number of balls, n denote the total number of threads, and g denote the grid size (number of blocks). The size of a float4 is 16 bytes. The code runs on an NVIDIA Kepler GPU, which supports requests of size 32, 64, and 128 bytes.

```c
int num_threads = blockDim.x * gridDim.x, tid = threadIdx.x + blockIdx.x * blockDim.x,
int a_idx_start = tid, balls_per_thread = dc.chain_length / num_threads;
const int csize = 8;
__shared__ float4 c_pos[csize];
for ( int i=0; i<balls_per_thread; i++ ) {
    const int a_idx = a_idx_start + i * num_threads;
    const float4 pos_a = dc.d_pos[a_idx];               // Global Access A
    float3 force = make_float3(0,0,0);
    for ( int b_idx = 0; b_idx < dc.chain_length; b_idx++ ) {
        const int c_idx = b_idx % csize;
        if ( c_idx == 0 ) {
            __syncthreads();
            c_pos[loc] = dc.d_pos[b_idx + loc];            // Global Access B
            __syncthreads();
        }
        const float4 pos_b = c_pos[c_idx];
        force += force_compute(pos_a,pos_b);
    }
    dc.d_vel[a_idx] += delta_t * dc.d_balls[a_idx].mass_inv * force;
}
```

(a) In terms of n, c, and g compute the amount of data requested and used for the line marked Global Access B.

- Data requested by Global Access B: 0
- Data used by B: 0

(b) Choose csize to minimize memory access by Global Access B.

- Show new value of csize: 16
- Explain: By increasing csize to 16, we can ensure that the requested memory is aligned with the supported request sizes of the GPU, thus minimizing memory access latency.

(c) Modify the code so that the amount of data accessed due to Global Access B matches the previous part when csize is set to the original value, 8. The change should be within the if ( c_idx==0 ) block and it is only one line of code.

- Modify code to reduce waste when csize=8: `c_pos[loc] = dc.d_pos[b_idx + loc];` should be modified to `c_pos[loc] = dc.d_pos[b_idx + loc] % csize;`
Problem 3: (15 pts) Answer the following questions about triangles.

(a) The diagrams below show two possible numberings for vertices for rendering a rectangle as a triangle strip.

Which one is correct?

Show how the triangles would be rendered for the incorrect case.

(b) Several possible strategies are being considered to render a rectangle.

Suppose the left figure above were rendered as a triangle strip (with the vertices correctly numbered).

How many times would the vertex shader be executed?

Suppose the left figure above was rendered as individual triangles.

How many times would the vertex shader be executed?
Problem 3, continued:

(c) Consider the following two ways of tessellating a rectangle.

Suppose that lighting is computed in the vertex shader.

☐ Which tessellation above is better?  ☐ Explain.

(d) Suppose that lighting were being computed in the fragment shader.

☐ Why would either tessellation above be wasteful?
Problem 4: (10 pts) In class we saw how we could obtain the mirror image of a vertex easily if the mirror were on the \(xz\) plane at \(y = 0\). For such a mirror, the mirror image of a vertex at \((x, y, z)\) would be at \((x, -y, z)\).

Appearing below are a vertex shader and geometry shader that do nothing special: The vertex shader computes the clip-space coordinates and lighting, and the geometry shader emits one triangle.

Modify the shader(s) so that for each original triangle two are emitted: one in the original position, and the other in the reflected position. (Don’t worry about stenciling or anything like that.)

*Hint: Don’t forget that the mirror is in object-space (world-space) coordinates.*

- Modify to emit reflected triangle, in addition to regular.
- Think about coordinate spaces.

```c
#pragma once

void vs_main_basic() {
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

    // Compute eye-space vertex coordinate and normal.
    // These are outputs of the vertex shader and inputs to the frag shader.
    vec4 var_vertex_e = gl_ModelViewMatrix * gl_Vertex;
    vec3 var_normal_e = normalize(gl_NormalMatrix * gl_Normal);
    vec4 lcolor = generic_lighting( var_vertex_e, gl_Color, var_normal_e);

    gl_BackColor = gl_FrontColor = lcolor;
}

out Data {
    int hidx; // Not used.
};

#pragma once

void gs_main_helix() {
    int i;
    for ( int i = 0; i < 3; i++ )
    {
        gl_FrontColor = gl_FrontColorIn[i];
        gl_BackColor = gl_BackColorIn[i];
        gl_Position = gl_PositionIn[i];
        EmitVertex();
    }

    EndPrimitive();
}

#pragma once
```

Problem 5: (15 pts) Answer each question below.

(a) Suppose we have a buffer object for vertices corresponding to the object on the lower left.

How can we use that buffer object to render an object that will look like the one on the right?

(b) Describe a situation in which using `glVertex` rather than vertex arrays (`glDraw`) is a good idea.

Situation in which `glVertex` better:

(c) Describe a situation in which using vertex arrays but without buffer objects is a good idea.

Situation in which vertex arrays without buffer objects better:
Problem 6: (10 pts) In class the topic of texture filtering was introduced with a description of what would happen if a texture image of a picket fence (white stripes in front of a green background) were rendered without an appropriate MIPMAP level and without filtering. Either the fence or the grass would disappear.

(a) Illustrate the problem. On your diagram show the fence, some pixels and some texels. It is important to choose the size of the pixels and texels correctly.

☐ Illustration of problem. ☐ Show pixels. ☐ Show texels.

(b) Re-draw the diagram showing how the choice of an appropriate MIPMAP level can avoid the problem of the fence disappearing.

☐ Illustration of how MIPMAP level avoids problem.

(c) Draw a diagram showing an example of how linear filtering can improve the appearance of the fence.

☐ Example of how linear filtering improves appearance.