Name ______________________

GPU Programming
EE 4702-1
Final Examination

Tuesday, 9 December 2014 7:30–9:30 CST

Problem 1	______	(20 pts)
Problem 2	______	(15 pts)
Problem 3	______	(20 pts)
Problem 4	______	(20 pts)
Problem 5	______	(25 pts)
Exam Total	______	(100 pts)

Alias ______________________

Good Luck!

Problem 1: [20 pts] Part of the vertex shader from Homework 6 (which is similar to the vertex shader in earlier assignments) appears below. For the questions below let c denote the chain length (the number of balls), let s denote the value of `opt_segments` (the number of iterations of the `t` loop in the CPU code), and let $n = cs$ denote the number of "vertices" sent to the vertex shader.

```
in ivec2 gl_Vertex; // Vertex Shader Inputs

out Data_to_GS {    // Interface block for vertex shader output / geometry shader input.
  vec4 vertex_e[2][2];  // Vertex coordinates in eye space.
  vec4 position[2][2];  // Vertex coordinates in clip space.
  vec2 texCoord[2];
  vec3 normal_e[2];
  vec3 radial_e;  // Normal for edge primitives.
};

void vs_main_lines() {
  const int bidx = gl_Vertex.x;
  const int ti = gl_Vertex.y;
  const int radial_idx = bidx * opt_segments + ti;
  const float delta_t = 1.0 / opt_segments;
  const float t = float(ti) * delta_t;
  const float theta = delta_t * radial_idx * omega;
  vec3 pos1 = balls_pos[bidx-1].xyz;                 // Buffer object access.
  vec3 pos2 = balls_pos[bidx].xyz;                   // Buffer object access.
  vec3 v12 = pos2.xyz - pos1.xyz;
  // ...
```

(a) For an entire rendering pass (execution of `glDrawArrays`) how much data is sent from the CPU to the GPU for this vertex shader? The answer should be in bytes and in terms of c, s, and n.

☐ Amount of data from CPU to GPU:

(b) For an entire rendering pass (execution of `glDrawArrays`) how much data is output by the vertex shader for the geometry shader? The answer should be in bytes and in terms of c, s, and n.

☐ Amount of data output from vertex shader:

(c) For an entire rendering pass (execution of `glDrawArrays`) how much data is read by the geometry shader (not shown) per frame? It's important to remember that the input to the vertex shader is line strips and the input to the geometry shader is lines. The answer should be in bytes and in terms of c, s, and n.

☐ Amount of data input to geometry shader:

Problem 1, continued: Recall that `balls_pos` is an array of ball positions, something that changes each frame. Note that the vertex shader is reading the positions from a buffer object.

It would be possible, though wasteful, to use a vertex shader input for the ball positions.

(*d*) Show the vertex shader input declarations that can be used for ball positions. *Hint: This part is easy, look at the code on the previous page.*

☐ Declarations:

(*e*) Compute the amount of data sent from the CPU to the GPU needed when the ball positions are read as vertex shader inputs. The answer should be in bytes and in terms of c, s, and n.

☐ Amount of data for ball positions as vertex shader inputs:

Problem 2: [15 pts] The fragment shader used in many class demonstrations and homework assignments performs lighting calculations. This yields the best results, but is computationally expensive. The historic location for computing lighting is the vertex shader.

(*a*) With the help of a diagram describe how fragment-shader lighting computation yields a more realistic image than vertex-shader lighting computation.

☐ Fragment shader lighting more realistic because:

(*b*) How many times more work does it take to compute lighting in the fragment shader? **State any assumptions** used for your answer.

☐ Using the fragment shader for lighting requires $x\times$ more computation assuming:

(*c*) Doing lighting calculations in the fragment shader does not always make a big difference in appearance. Consider a hybrid scheme in which a fragment-shader input determines whether the fragment shader should do the lighting calculations or just use the `color` input. Which stage is best suited to determine if the fragment shader should do lighting calculations?

☐ Shader stage best suited to determine where to compute lighting?

☐ Explain.

Problem 3: [20 pts] The CUDA kernel below computes an operation on all pairs of elements of array `d_pos`, the number of array elements is `chain_length`. Let B denote the block size of the kernel launch. The grid size (number of blocks) is (and must be) `chain_length`.

```
__global__ void force_pairs()
{
  const int a_idx = blockIdx.x;
  const float4 pos_a = d_pos[a_idx];                  // Global Access A

  float3 force = make_float3(0,0,0);

  for ( int b_idx = threadIdx.x;  b_idx < chain_length;  b_idx += blockDim.x)
    {
      const float4 pos_b = d_pos[b_idx];
      force += force_compute(pos_a,pos_b);
    }

  d_vel[a_idx] += delta_t * d_balls[a_idx].mass_inv * force; // Error here.
}
```

(*a*) Compute the amount of data read for the entire kernel due to the line Global Access A. The size of a `float4` is 16 bytes and the minimum request size is 32 bytes. Compute this number in terms of B (block size) and c (`chain_length`). *Note: In the original exam n was used for chain length.*

☐ Amount of data read due to Access A.

(*b*) Modify the code so that it uses shared memory to reduce the amount of data read due to Global Access A.

☐ Modify to use shared memory to reduce data read for A.

Problem 3, continued: The CUDA code below is the same as the previous page. The line marked "Error here" has two problems.

```
__global__ void force_pairs()
{
  const int a_idx = blockIdx.x;
  const float4 pos_a = d_pos[a_idx];                  // Global Access A

  float3 force = make_float3(0,0,0);

  for ( int b_idx = threadIdx.x;  b_idx < chain_length;  b_idx += blockDim.x)
    {
      const float4 pos_b = d_pos[b_idx];
      force += force_compute(pos_a,pos_b);
    }

  d_vel[a_idx] += delta_t * d_balls[a_idx].mass_inv * force; // Error here.
}
```

(*c*) Due to one problem the wrong value will be written into `d_vel`. Explain why.

☐ Wrong value written to `d_vel` because ....

(*d*) The line marked "Error here" is also inefficient. Explain why and fix the problem. *Hint: Use shared memory.*

☐ Reason for inefficiency.

☐ Fix with shared memory.

Problem 4: Answer each question below.

(*a*) [5 pts] A CUDA application launches a kernel with a grid size of 20 blocks. Describe a situation in which that is inefficient and explain why.

☐ Twenty blocks is inefficient when:

☐ Because:

(*b*) [5 pts] A CUDA application launches a kernel with a block size of 19 threads. Why is that inefficient?

☐ Nineteen threads is inefficient because:

Problem 4, continued:

(*c*) [5 pts] The loop body in the CUDA code below does one of five kinds of operations. Which is the biggest concern for the code below, grid size, block size, or warp size?

```
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  for ( int i = tid; i<size; i += num_threads ) {
    int k = kind[i];
    float4 d = data[i];
    switch ( k ) {
      case 0: operation_x(i,d); break;
      case 1: operation_y(i,d); break;
      case 2: operation_z(i,d); break;
      case 3: operation_a(i,d); break;
      case 4: operation_b(i,d); break;
    }
  }
```

☐ Biggest concern for code is grid, block, or warp. (Circle one.)

☐ Explain:

(*d*) [5 pts] For many classroom CUDA examples a structure is used to hold application related variables, such as the one below. For the GPU, this had been declared in the `__constant__` address space, but it could have been declared in the global (device) address space (shown commented out below). What's the disadvantage of the global space for this?

```
struct App {
  int num_threads, array_size;
  Vector *v_in;
  Vector *d_v_in;
};

App app;  // In host address space.
__constant__ App d_app;  // In device constant address space.
// __device__ App d_app;  // In device global address space.
```

☐ Disadvantage of global address space for the `App` structure?

Problem 5: Answer each question below.

(*a*) [5 pts] The vertex shader below uses the deprecated output `gl_FrontColor` to hold a lighted color. Those setting the OpenGL standard would prefer that you define your own vertex shader output to hold the lighted color. On the other hand, `gl_Position` is not deprecated and it must be written (if not by the vertex shader, then by the geometry shader). Why?

```
void vs_main() {
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

  vec4 vertex_e = gl_ModelViewMatrix * gl_Vertex;
  vec3 normal_e = normalize(gl_NormalMatrix * gl_Normal);
  gl_FrontColor = generic_lighting(vertex_e,gl_Color,normal_e);
}
```

☐ Cannot substitute `gl_Position` with a user-defined output because:

(*b*) [5 pts] It's possible to define user-defined outputs for the vertex and geometry shaders, but not for the fragment or compute shaders.

☐ Why can't there be user-defined fragment shader outputs?

☐ Why can't there be user-defined compute shader outputs?

(*c*) [5 pts] In the OpenGL shader code below `b` is assigned the value of `a`, but with the elements reversed. It does so the hard way. Do it the easy way (using what we called swizzle operations).

```
vec4 a, b;
b.x = a.w;  b.y = a.z;  b.z = a.y;  b.w = a.x;
```

☐ Assign `b` the easy way.

(*d*) [5 pts] The fragment shader input's `smooth` interpolation qualifier (which is the default for floating-point values) indicates that the qualified variable (`color` in the example below) should be linearly interpolated based on object-space coordinates. Linear interpolation is easy to do, so why is `smooth` interpolation expensive?

```
in Data_GF {
  smooth vec4 color;
}
```

☐ `smooth` interpolation is expensive because ...

(*e*) [5 pts] The statement below is correct but does not provide the full justification for GPUs. Complete the statement.

> Executing 3D graphics is computationally intensive and so many computers have GPUs to take this computation load off the CPU.

☐ To justify GPUs the statement should go on to say ....