Name _____

GPU Programming
EE 4702-1
Final Examination

Friday, 11 December 2015 15:00–17:00 CST

Problem 1	_____	(20 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(15 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(20 pts)
Problem 6	_____	(10 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [20 pts] The code below, taken from Homework 4, performs the rendering pass used to draw the balls. Let n denote the number of balls and let s denote the number of vertices for each ball. It does so using an instanced draw in which a set of s vertices is sent into the rendering pipeline n times, for a total of sn vertices. Procedure `glDrawArraysInstanced(type,0,s,n)` sends a set of s vertices for rendering n times.

```
// CPU code for rendering pass.

  glBindBufferBase(GL_SHADER_STORAGE_BUFFER,1,balls_pos_rad_bo);
  glBindBufferBase(GL_SHADER_STORAGE_BUFFER,2,balls_color_bo);
  glBindBuffer(GL_ARRAY_BUFFER,sphere.points_bo);
  glVertexPointer(3,GL_FLOAT,0,0);
  glEnableClientState(GL_VERTEX_ARRAY);
  glDrawArraysInstanced(GL_TRIANGLE_STRIP, 0, num_vertices, num_balls );

// Vertex Shader Code for rendering pass.

void vs_main_instances() {
  vec4 center_o_rad = balls_pos_rad[gl_InstanceID];
  float rad = center_o_rad.w;
  vec3 to_surface = rad * gl_Vertex.xyz;
  vec4 position_o = vec4(center_o_rad.xyz + to_surface, 1f);

  gl_Position = gl_ModelViewProjectionMatrix * position_o;
  vertex_e = gl_ModelViewMatrix * position_o;
  normal_e = gl_NormalMatrix * gl_Vertex.xyz;
}
```

(*a*) In the shader code above indicate which variables are uniforms and which are vertex shader inputs. *Hint: This question can be answered without understanding what an instanced draw is.*

☐ Put a `U` next to all uniform variables.

☐ Put an `I` next to all vertex shader inputs.

(*b*) For each rendering pass indicate how much data must be sent from the CPU to the GPU due to this vertex shader. Assume that **all uniform variables** need to be **re-sent each rendering pass**. But, make an intelligent determination on whether vertex shader inputs and other data needs to be re-sent.

☐ Amount of CPU to GPU data for uniform variables per pass in terms of s and n.

☐ Amount of CPU to GPU data for vertex shader inputs per pass in terms of s and n.

☐ Amount of CPU to GPU data for other data used by vertex shader per pass in terms of s and n.

Problem 1, continued:

(*c*) Appearing below is the vertex shader used for spheres in an ordinary rendering pass, followed by the vertex shader for the instanced rendering. Both render the same balls, but using different CPU code.

```
void vs_main() {                // Vertex Shader for Ordinary Rendering
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  vertex_e = gl_ModelViewMatrix * gl_Vertex;
  normal_e = gl_NormalMatrix * gl_Vertex.xyz;
}

void vs_main_instances() {      // Vertex Shader for Instanced Rendering
  vec4 center_o_rad = balls_pos_rad[gl_InstanceID];
  float rad = center_o_rad.w;
  vec3 to_surface = rad * gl_Vertex.xyz;
  vec4 position_o = vec4(center_o_rad.xyz + to_surface, 1f);

  gl_Position = gl_ModelViewProjectionMatrix * position_o;
  vertex_e = gl_ModelViewMatrix * position_o;
  normal_e = gl_NormalMatrix * gl_Vertex.xyz;
}
```

Because it accesses `balls_pos_rad`, `vs_main_instances` *looks like* it's accessing more data than `vs_main`, but in reality less data is being sent to render all the balls.

☐ What data was sent from CPU to GPU in the original code that substituted for `balls_pos_rad`?

☐ How did this data compare in size to `balls_pos_rad`?

(*d*) Explain why the instanced draw gives higher performance than ordinary rendering.

☐ Higher performance because ...

Problem 2: [15 pts] CUDA kernel A launched with 10 blocks of 1024 threads on a GPU with 10 streaming multiprocessors (abbreviated SMs or MPs) takes 27 s to run. Consider several scenarios in which the kernel is launched on a GPU with 9 SMs. The only difference between the GPUs is the number of SMs. In both GPUs the maximum number of threads per SM and per block is 1024.

(a) Suppose that kernel A is launched again with 10 blocks but this time on the GPU with 9 SMs.

☐ Run time for A in a 10-block launch on the 9 SM GPU? ☐ Explain.

(b) How long would A take to run on the 9-SM GPU if launched with 9 blocks of 1024 threads, but doing the same amount of work as the 10-block launch. Kernel A was written by a skilled programmer.

☐ Run time for A in a 9-block launch on the 9 SM GPU? ☐ Explain.

(c) CUDA kernel C launched with 10 blocks of **32 threads** takes 72 s on the 10-SM GPU. How long would the 10-block launch take on the 9-SM GPU?

☐ Run time for C in a 10-block launch on the 9 SM GPU?

☐ Explain how the low thread count makes part c (this question) different than part a.

Problem 3: [15 pts] The CUDA kernel below is based on a classroom example. Notice that the computation of `tid` has been changed. Indicate the change in execution time (relative to the commented out code) and whether the new code is correct (produces the same answer as the commented out code). *Note: The original exam contained a second part.*

(*a*) Consider the simple kernel:

```
__global__ void cuda_thread_start_simple() {
  // const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int tid = threadIdx.x;                  // Changed code. Original line above.

  const int elt_per_thread = array_size / num_threads;
  const int start = elt_per_thread * tid;
  const int stop = start + elt_per_thread;

  for ( int h=start; h<stop; h++ ) {
      float4 p = d_v_in[h];
      d_m_out[h] = dot( p, p ); }}
```

☐ Due to change, execution time is ◯ *longer* ◯ *about the same or* ◯ *shorter* .

☐ Due to change, results are ◯ *incorrect* or ◯ *still correct.*

☐ Important: Explain by showing how array elements are assigned to threads.

Problem 4: [20 pts] Answer the following questions about CUDA.

(*a*) In an execution of the code below an SM reads 32 times more data than it needs to due to the way the code is written.

```
__global__ void some_prob(char *in_data, int *out_data) {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int width = 128;
  const int start_id = tid * width;
  int sum = 0;
  for ( int i=0; i<width; i++ ) sum += in_data[start_id+i];
  out_data[tid] = sum;
}
```

☐ Code reads 32× more data than it needs because ...

Suppose that the type of `in_data` were changed from `char*` to `int*`. Instead of reading 32× more data it would only be reading x× more data than needed.

☐ What is the value of x?

(*b*) The code sample below is based on the shared memory classroom demo. *Note: The second* `syncthreads` *did not appear in the original exam.*

```
__shared__ int sum;

if ( threadIdx.x == 0 )   sum = 0;
__syncthreads();
if ( threadIdx.x == 40 )  sum += 40;
if ( threadIdx.x == 70 )  sum += 70;
if ( threadIdx.x == 200 ) sum += 200;
__syncthreads();
out_data[tid] = sum;
```

☐ Show at least four different possible values for `sum` that the code above can write to `out_data`.

☐ Explain.

(*c*) Consider the use of `__syncthreads()` in the CUDA code below and in general.

☐ In general, what does `__syncthreads` do?

☐ What might go wrong if `__syncthreads` were removed from the code below?

```
__shared__ int our_data[1025];
our_data[threadIdx.x] = my_element;
__syncthreads();
output_data[tid] = my_element + our_data[threadIdx.x + 1];
```

☐ What's wrong with the use of `__syncthreads` below?

```
if ( threadIdx.x != 20 ) __syncthreads();
```

Problem 5: [20 pts] The geometry shader below, from Homework 4, passes a triangle unchanged.

```
layout ( triangles ) in;
layout ( triangle_strip, max_vertices = 3 ) out;

void gs_main_simple() {



  vec3 tnorm =                                  ; // FILL IN.



  for ( int i=0; i<3; i++ )
    {
      normal_e = In[i].normal_e;
      vertex_e = In[i].vertex_e;
      color = In[i].color;
      gl_Position = In[i].gl_Position;
      EmitVertex();
    }









  EndPrimitive();
}
```

(*a*) Add code so that `tnorm` is assigned the triangle's eye-space geometric normal.

☐ Set `tnorm` to eye-space geometric normal of triangle.

(*b*) Modify the shader so that it emits a second triangle of the same shape but displaced one unit in the direction of `tnorm` (see above). (For `normal_e` see next part.) The new triangle should be blue. The following library functions are available: `cross`, `dot`, `normalize`, and `length`. A library of colors **is not** available.

☐ Add code to emit second triangle.

☐ Be sure to assign ☐ `vertex_e`, ☐ `color` to blue, and (important) ☐ `gl_Position`.

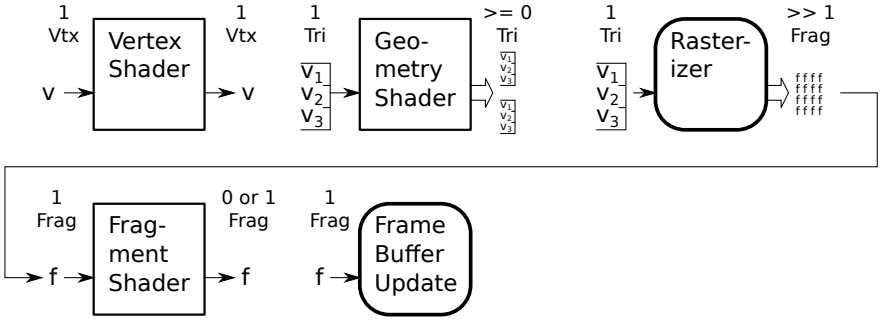
☐ Modify the `layout` declarations, if necessary.

(*c*) What about `normal_e` for the new triangle? Suppose we are rendering a sphere. Compare the appearance of the new triangle with `normal_e` set to `tnorm` to the appearance when `normal_e` is set to `In[i].normal_e`.

☐ Describe difference in appearance for `normal_e = tnorm` versus `normal_e = In[i].normal_e`.

Problem 6: [10 pts] Answer the following questions.

(*a*) The depth test (also called the z test) in an important part of the rendering pipeline.



☐ Where in the rendering pipeline is the depth test performed?

☐ Why would it make no sense to perform the depth test in the vertex shader?

(*b*) Switching from a triangle strip to individual triangles increases the amount of work performed by the vertex shader by a factor of 3, but does not change the amount of work performed by the geometry and fragment shaders.

☐ Why does the vertex shader do 3× more work?

☐ Why do the geometry and fragment shaders do the same amount of work?