Name ______________________

EE 4702

Take-Home Pre-Final Questions

Due: 9 December 2013

Please work on this exam alone. You may use references for OpenGL, CUDA, and similar topics.

Problem 1	________	(20 pts)
Problem 2	________	(30 pts)
Problem 3	________	(15 pts)
Problem 4	________	(20 pts)
Problem 5	________	(15 pts)
Exam Total	________	(100 pts)

Alias ______________________

Good Luck!

Problem 1: [20 pts]The code below is the `data_unpack` routine from the Homework 3 solution. Recall that the code in Homework 3 requires that the number of threads must be no smaller than the number of balls (the value of `chain_length`).

Re-write the routine so that it works correctly even if the number of balls is greater than the number of threads.

☐ Re-write so number of threads can be fewer than `chain_length`.

☐ Compute the number of threads using CUDA-provided variables such as `blockDim`.

☐ Array `dc.d_pos` must be accessed efficiently.

```
__global__ void data_unpack()
{




  const int bi = threadIdx.x + blockIdx.x * blockDim.x;
  if ( bi >= dc.chain_length ) return;




  Ball* const ball = &dc.d_balls[bi];
  ball->position = dc.d_pos[bi];




}
```

Problem 2: [30 pts]Recall that the code for Homework 3 simulated a chain of balls (or string of beads). In this problem we'll extend that code by giving the balls electric charge, either positive or negative. The balls are sealed in a special coating that retains the charge.

The CUDA kernel below applies the force due to these charges. Like the penetration routine it must consider nearly all possible pairs. The code is written to optionally use shared memory when accessing the "b" object.

```
__global__ void time_step_gpu_sol_part_m(float delta_t) {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int threads_per_object = num_threads / dc.chain_length;

  const int a_idx = tid % dc.chain_length;
  const int b_idx_start = tid / dc.chain_length;

  __shared__ float4 c_pos[1024];

  float4 pos_a = dc.d_pos[a_idx];                                       // Global Access
  float3 force = make_float3(0,0,0);

  for ( int j=b_idx_start; j < dc.chain_length; j += threads_per_object )
    {
      if ( use_cache )
        {
          __syncthreads();
          if ( a_idx == 0 ) c_pos[b_idx_start] = dc.d_pos[j];           // Global Access
          __syncthreads();
        }

      if ( a_idx != j )
        {
          float4 pos_b = use_cache ? c_pos[b_idx_start] : dc.d_pos[j];  // Global Access
          pNorm a_to_b = mn(pos_a,pos_b);
          const bool repel = ( a_idx & 1 ) == ( j & 1 );
          force += ( repel ? -0.15f : 0.15f ) / a_to_b.mag_sq * a_to_b;
        }
    }

  const float3 delta_v = delta_t * dc.d_balls[a_idx].mass_inv * force;
  atomicAdd(&dc.d_vel[a_idx].x,delta_v.x);
  atomicAdd(&dc.d_vel[a_idx].y,delta_v.y);
  atomicAdd(&dc.d_vel[a_idx].z,delta_v.z); }
```

Questions are on next page.

Problem 2, continued:

(*a*) In this part assume that `use_cache` is false, and so shared memory won't be used. Compute the amount of data requested due to accesses to `d_pos` and compute how much of that data is actually used. Do this for the following configuration: `chain_length=256`, a block size of 1024 threads, and a grid size of 16 blocks. The code runs on an NVIDIA Kepler GPU which has memory request sizes of 32, 64, and 128 bytes. For your convenience, comments show where global memory can be accessed. (Array `c_pos` is not in global memory, and is anyway won't be needed until the next subproblem.)

☐ Total size of requests (in bytes) to `d_pos` considering all threads.

☐ Amount of data in those requests that was needed.

(*b*) Repeat the problem above for the case when `use_cache` is true. Remember that memory requests are not generated when reading or writing `c_pos` itself.

☐ Total size of requests (in bytes) to `d_pos` considering all threads.

☐ Amount of data in those requests that was needed.

(*c*) The `d_pos` accesses that are used to fill shared memory, `c_pos[b_idx_start] = dc.d_pos[j];`, are wasteful (though the waste is small in proportion to the total amount of data accessed). Explain why the accesses are wasteful and fix the problem.

☐ Fix problem.

☐ Why are these accesses wasteful?

(*d*) The amount of memory read when assigning `pos_a` can be reduced by using shared memory. Modify the code to do so.

☐ Use shared memory to reduce redundant global accesses to initialize `pos_a`.

Problem 3: [15 pts]Consider the following excerpt from the Cone code from the Homework 2 solution. (File `hw2-sol.cc` or visit `http://www.ece.lsu.edu/koppel/gpup/gpup/2013/hw2-sol.cc.html`).

```
void render_probb(pCoor base, float radius, pVect to_apex) {
    if ( opt_lod != bo_lod )  // If needed lod != stored lod.
      {
        bo_lod = opt_lod;

        /// Code for computing coordinates and normals omitted.


        if ( !buffer_obj_coords ) {
            glGenBuffers(1,&buffer_obj_coords);
            glGenBuffers(1,&buffer_obj_norms);  }

        glBindBuffer(GL_ARRAY_BUFFER, buffer_obj_coords);
        glBufferData(GL_ARRAY_BUFFER, coords.occ() * sizeof(float),
                     coords.get_storage(), GL_STATIC_DRAW);

        glBindBuffer(GL_ARRAY_BUFFER, buffer_obj_norms);
        glBufferData(GL_ARRAY_BUFFER, norms.occ() * sizeof(float),
                     norms.get_storage(), GL_STATIC_DRAW);
      }

    // Transform computation and modelview update omitted.

    glColor3fv(color);

    // Tell OpenGL to get coordinates and normals from buffer objects.
    //
    glBindBuffer(GL_ARRAY_BUFFER, buffer_obj_coords);
    glVertexPointer( 3, GL_FLOAT, 0, NULL);
    glEnableClientState(GL_VERTEX_ARRAY);

    glBindBuffer(GL_ARRAY_BUFFER, buffer_obj_norms);
    glNormalPointer(GL_FLOAT,0,NULL);
    glEnableClientState(GL_NORMAL_ARRAY);

    // Draw the cones. (There will be minor flaws since 1 strip used.)
    glDrawArrays(GL_QUAD_STRIP,0,num_coords);

    // Cleanup code omitted.
```

Questions are on next page.

Problem 3, continued:

(*a*) The code uses buffer objects for vertex coordinates and normals. Why doesn't it also use a buffer object for color?

☐ Why doesn't color need a buffer object?

(*b*) Modify the code so that it uses a buffer object for color, even if that's not a good idea.

☐ Modify code so color *does* use a buffer object for color.

(*c*) Suppose that each time the routine above was called `opt_lod` had a different value. Also suppose that `if ( !buffer_obj_coords )` were somehow changed to `if ( true )`.

☐ What would eventually happen?

Problem 4: [20 pts]Consider another excerpt from the Cone code from the Homework 2 solution. (File `hw2-sol.cc` starting at line 266. or visit
`http://www.ece.lsu.edu/koppel/gpup/gpup/2013/hw2-sol.cc.html`).

```
// Compute transform moving and positioning code from local to global
// space.
//
const float to_height = to_apex.mag();
pVect from_apex(0,0,1);
pNorm rn(from_apex,to_apex);
const float rot_angle = pangle(from_apex,to_apex);
pMatrix_Translate trans_transl(base);
pMatrix_Rotation trans_rot(rn,rot_angle);
pMatrix_Scale trans_scale(radius);
trans_scale.rc(2,2) = to_height;
pMatrix xform = trans_transl * trans_rot * trans_scale;

// Specify our transformation to OpenGL.
//
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glMultTransposeMatrixf(xform.a);

if ( !dont_set_color ) glColor3fv(color);

glBindBuffer(GL_ARRAY_BUFFER, buffer_obj_coords);
glVertexPointer( 3, GL_FLOAT, 0, NULL);
glEnableClientState(GL_VERTEX_ARRAY);

glBindBuffer(GL_ARRAY_BUFFER, buffer_obj_norms);
glNormalPointer(GL_FLOAT,0,NULL);
glEnableClientState(GL_NORMAL_ARRAY);

glDrawArrays(GL_QUAD_STRIP,0,num_coords);
```

The code at the top of the excerpt computes a transform matrix that will position and scale the cone to the desired location. The transform is computed in terms of coordinate `base`, scalar `radius`, and vector `to_apex`. The command `glMultTransposeMatrixf` takes the transform matrix we computed and multiplies it by the existing modelview matrix. The resulting matrix is written as an OpenGL Shading Language uniform with name `gl_ModelViewMatrix`.

In this problem we'll consider computing the transform on the GPU instead. The GPU code will compute the transform in terms of `base`, `radius`, and `to_apex`.

Questions are on next page.

Problem 4, continued:

(*a*) Declare `base`, `radius`, and `to_apex` as uniforms as they would be in an OpenGL SL source file. *Hint: Use the demo-10 (Vertex and Geometry Shaders) code as an example, and look for* `wire_radius`*. The demo-10 code can be accessed from the repo or*
`http://www.ece.lsu.edu/koppel/gpup/code/gpup/demo-10-shader.cc.html` *(CPU code),*
`http://www.ece.lsu.edu/koppel/gpup/code/gpup/demo-10-shdr-simple.cc.html` *(simple shaders), and*
`http://www.ece.lsu.edu/koppel/gpup/code/gpup/demo-10-shdr-geo.cc.html` *(more elaborate shaders).*

(*b*) Write code to send the uniforms from the CPU to the GPU. *Hint: See the previous hint.*

(*c*) Suppose that `xform` is computed in a vertex shader making use of the enormous floating-point capability of the mighty GPU. Do you expect execution to be faster or slower than using the CPU to compute xform? Explain.

(*d*) Suppose that `xform` is computed in a geometry shader. Will execution be faster or slower than using the vertex shader when we are rendering quad strips? (That is, the systems we are comparing both use quad strips, one uses the vertex shader to compute the xform, the other uses the geometry shader.) Explain.

(*e*) Suppose that `xform` is computed in a geometry shader. Will execution be faster or slower than using the vertex shader when we are rendering individual quads? (That is, the systems we are comparing both use individual quads, one uses the vertex shader to compute the xform, the other uses the geometry shader.) Explain.

Problem 5: [15 pts]Answer each question below.

(*a*) Texture access is performed in the fragment shader. Suppose texture access was performed by the vertex shader and the texel values were interpolated, the same way other attributes such as color were.

☐ What impact would that have on appearance?

☐ What impact would that have on performance?

(*b*) Inputs to a fragment shader have interpolation qualifiers, they are `flat`, `noperspective`, and `smooth`. Why are they not needed for the inputs to the geometry shader?

☐ Interpolation qualifiers not needed for geometry shader inputs because:

(*c*) The OpenGL command `glMatrixMode(GL_MODELVIEW)`, used in setting the modelview matrix, is deprecated. (Meaning that its use is discouraged and that it may be removed from the language.) Why was the command once essential but now considered unnecessary?

☐ `glMatrixMode` now unnecessary because: