Name ______________________________

EE 4702

Take-Home Pre-Final Examination

Tuesday, 29 November 2011 to Friday, 2 December 2011

Work on this exam alone. Regular class resources, such as notes, papers, documentation, and code, can be used to find solutions. Do not try to seek out references that specifically answer any question here. Do not discuss this exam with classmates or anyone else. Any questions or concerns about problems should be directed to Dr. Koppelman.

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

Alias ______________________________

Good Luck!

Problem 1: [20 pts]The advantage of an OpenGL triangle strip over individual triangles is that vertices shared by multiple triangles only need to be specified once.

(*a*) Why is the code below not a good example?

```
for ( int i=0; i<1000; i++ )
  {
    glBegin(GL_TRIANGLE_STRIP);
    glNormal3fv(normal[i]);  glVertex3fv(v0[i]);
    glNormal3fv(normal[i]);  glVertex3fv(v1[i]);
    glNormal3fv(normal[i]);  glVertex3fv(v2[i]);
    glEnd();
  }
```

(*b*) Explain the impact of the proper use of triangle strips on each of the items below. For one of the items below the answer should consider two cases, when a buffer object is being used and when it is not being used. Remember, for this part assume that triangle strips are being used properly.

- Total communication bandwidth.

- Total vertex shader execution time.

- Total fragment shader execution time.

(*c*) A triangle strip is well suited for situations in which triangles are approximating a smooth surface (such as the surface of a sphere). It is less well suited for surfaces that really are composed of triangles. Explain why. *Hint: Consider colors and normals.*

Problem 2: [20 pts]Assume that the lighted color of a triangle depends on conditions at its centroid (a point equidistant from its three vertices), including light and viewer location. Though computed for the centroid, this same lighted color would be used for all fragments.

(*a*) Which would be the most appropriate shader to compute this color: vertex, geometry, or fragment? Explain.

(*b*) Regardless of your answer above, explain how a vertex shader could be used to compute the centroid (a first step in computing the color). This will require some extra help from the CPU, but the burden of computation should be on the shader. (*Note: the challenge here is getting the vertex locations.*)

(*c*) Shown below are three possible declarations for the output of the shader that computes the triangle's lighted color. Each of these will produce the desired result. But one is clearly best, and one is clearly worst. Identify the best and worst to use and explain why for each.

```
flat out vec3 tri_color;
smooth out vec3 tri_color;
noperspective out vec3 tri_color;
```

Problem 3: [20 pts]Suppose an NVIDIA GPU of compute capability 2.0 has 8 multiprocessors. Each multiprocessor has 32 cores, but a single warp can use only 16 cores (it takes two different warps to keep all 32 cores in a multiprocessor busy).

The GPU is to be used to add two arrays element-wise. Suppose that the number of array elements is 2^{24}.

Let t_1 denote the amount of time it takes one thread (yes, just one) to perform the entire calculation on the GPU. The kernel code is shown below:

```
__device__ void prob(int array_size) {
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  for ( int i=tid; i<array_size; i += num_threads )
    result[tid] = a[tid] + b[tid];
}
```

(*a*) What would be the time to perform the calculation for a single block of just two threads (in terms of t_1)?

(*b*) Consider a launch of eight blocks of 64 threads each. Explain why reducing the number of blocks by one will leave some CUDA cores totally unused.

(*c*) Consider a launch of eight blocks of 64 threads each. Explain why reducing the block size by 17 or more will leave some CUDA cores totally unused.

(*d*) Explain why a launch of 16 blocks of 512 threads would take about the same amount of time to run as a launch of 32 blocks of 256 threads. *Note: Up to 8 blocks or 48 warps can run on a multiprocessor.*

(*e*) Explain how memory latency would increase the number of threads needed for good performance.

Problem 4: [20 pts]Consider the CUDA code below in which two alternatives are shown: putting a balls array in shared memory or in global memory. Assume the code runs on GPUs of compute capability 2.0 (though the answers could apply to 1.0 through 2.2).

(*a*) Explain why the Option 1 (shared memory) code below will run slowly. Fix the problem making as few changes as possible.

```
struct Ball {
  float3 position;   float3 velocity;
  float radius;      float mass;      };

__shared__ Ball balls[256]; // Option 1 - Shared memory.

__device__ void
update(float deltat)
{
  int start = threadIdx.x * balls_per_thread;
  int stop = start + balls_per_thread;
  for ( int i=start; i<stop; i++ )
    balls[i].position += deltat * balls[i].velocity;
}
```

(*b*) Explain why the Option 2 code below (using global memory) will run slowly. Fix the problems making as few changes as possible. (There are two major things to be fixed.)

```
struct Ball {
  float3 position;   float3 velocity;
  float radius;      float mass;      };

Ball* balls; // Option 2 - Global memory.

__device__ void
update(float deltat)
{
  int start = threadIdx.x * balls_per_thread;
  int stop = start + balls_per_thread;
  for ( int i=start; i<stop; i++ )
    balls[i].position += deltat * balls[i].velocity;
}
```

Problem 5: [20 pts]Answer each question below.

(*a*) Why is the if/return needed in the CUDA code below?

```
__device__ void prob(int array_size) {
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if ( tid >= array_size ) return;
  result[tid] = a[tid] + b[tid];
}
```

(*b*) The line of code below checks for a special case to avoid a square root. That might make sense for a CPU, but not for necessarily for CUDA code on a GPU. Describe a situation in which it makes sense for CUDA and a different situation when it makes no sense (meaning it would be faster to do the square root all the time). Assume that 50% of the time d is equal to 1.

```
  if ( d == 1 ) s = 1; else s = sqrt(d);
```

(*c*) The loop below is innocent on a CPU, but on a GPU it can execute inefficiently. Identify the problem and fix it.

```
  for ( int i=0; i<32; i++ )
    {
      if ( a[i] == t ) { do_stuff(i); break; }
    }
```