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 _________ (30 pts)
Problem 2 _________ (20 pts)
Problem 3 _________ (50 pts)

Alias ____________________________

Exam Total _________ (100 pts)

Good Luck!
Problem 1: [30 pts] Consider the following schemes to reduce the resources needed to render a sphere. A buffer object will be used to hold the coordinates of triangles forming the entire sphere (similar to the method used in the classroom demos), organized as a single triangle strip. The coordinates are for a sphere of radius 1 with the center at the origin. Note that with this positioning the vertex coordinate (before transformation) is the same as the vertex normal. The modelview matrix (\texttt{gl\_ModelViewMatrix}), which in this problem is assumed to be already set, will correctly position the sphere in eye space. (Note that \texttt{gl\_NormalMatrix} (the transpose of the inverse of the modelview matrix) must be used to transform the vertex into a correct eye-space normal.) Unlike the classroom demos, there is no texture and so there is no need to orient (rotate) the sphere in any particular way.

(a) A shader will determine whether the item it is working on (vertex, triangle, or fragment, depending on the shader) is on the side of the sphere facing the user (and so potentially visible) or on the side away from the user (and so not visible). If it is on the side away from the user the shader will take some action to avoid rendering the object (if the shader allows that).

For each shader below:

- Write shader code that writes \texttt{true} to variable \texttt{visible} if the item is visible. If necessary, a shader can use information provided by an upstream shader. (E.g., the fragment shader can be helped by the vertex or geometry shader). An attached sheet or file can be used for the shader code.

- Indicate whether the shader can stop the rendering of the item, and if so, show how.

- Estimate the amount of computation saved by stopping the rendering in the shader.

Afterward indicate the best shader(s) to use to minimize computation.

- Vertex Shader.  Visibility Code  Stop Rendering  Saved Computation

- Geometry Shader.  Visibility Code  Stop Rendering  Saved Computation

- Fragment Shader.  Visibility Code  Stop Rendering  Saved Computation

- The best shader(s) to save computation time for invisible part of sphere. Explain.
Problem 1, continued: Continuing with the sphere problem.

One way to reduce the size of the buffer object is to have it store information on only half the sphere, say the north hemisphere. Given coordinate \((x, y, z)\) in the north hemisphere one can trivially compute \((-x, -y, -z)\), the coordinate in the south hemisphere, if that coordinate were needed.

\((b)\) Show shader code for rendering the sphere using the hemisphere buffer object. Show only the shaders that are needed and assume a routine \texttt{lighting(pos,norm)} is available that sets the correct lighting for a coordinate at \texttt{pos} with normal \texttt{norm}.

For this sub-problem do not rotate the sphere so that its north pole faces the user. That’s a good idea, but it won’t work when the buffer object stores only one eighth of a sphere.

As with the previous problem, the shader code should not emit triangles if they are not visible.

☐ Which is the most appropriate shader for this code?

☐ Show the shader code. (Or use attached sheet or file.)
Problem 1, continued: Still continuing with the sphere problem.

(c) Compare the performance of the approach used in part (a) to the approach used in part (b).

Compare execution time for the two approaches, making reasonable assumptions.
Problem 2: [20 pts] The code below computes a cross product and stores the normalized result. Assume that the code runs on an NVIDIA GPU of compute capability 2.0.

```c
__global__ void prob_cross_prod(int num_elements)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int num_threads = blockDim.x * gridDim.x;
    for ( int idx = tid; idx < num_elements; idx += num_threads )
    {
        float3 a = arraya[idx]; // Vector load. (One add (for addr) + load insn)
        float3 b = arrayb[idx]; // Vector load. (One add (for addr) + load insn)
        float3 vec = cross(a,b); // Six multiplications, three additions.
        float3 mag_sq = dot(a,b); // Three multiplications, two additions.
        float3 length_inv = 1.0f / sqrtf(mag_sq); // See CUDA manual.
        float3 nvec = length_inv * vec; // Three multiplications.
        arrayc[idx] = nvec; // Vector store. (One add (for addr)+store insn)
    }
}
```

(a) Assuming a very large number of threads, determine the number of clock cycles needed per element per multiprocessor. Base your answer on the information in the CUDA Programmers Guide Version section 5.4.1 (linked to [http://www.ece.lsu.edu/koppel/gpup/refs.html](http://www.ece.lsu.edu/koppel/gpup/refs.html)) and on the comments in the code above. Be sure to figure out the instructions needed for the line assigning `length_inv`.

(For example, suppose it took only two floating-point adds for an the entire loop iteration (which computes one element). Since a multiprocessor, based on Table 5-1, can execute 32 FP adds per clock cycle, a multiprocessor requires just \( \frac{2}{32} \) clock cycles per element.)

- [ ] Clock cycles per element assuming a large number of threads.

(b) Suppose a global memory load had a latency of 400 cycles. Based on your answer to the previous part, what would be the minimum number of threads needed to make full utilization of a multiprocessor. (Hint: Compute the latency (beginning to end time) for one iteration of thread \( X \), then determine how many threads can run in those times when thread \( X \) is waiting for memory.)

- [ ] Minimum number of threads.
Problem 3: [50 pts] The CUDA code below and on the next few pages operates on pairs of objects stored in shared memory, much like the collision code for the balls demo. Like the balls demo, each object might appear in about five or six pairs, and there is stable pattern of pairs. (Don’t expect, say, pair.x to be numerically close to pair.y.)

The code in planA loads all objects from global memory. The code in planB first loads the objects into shared memory, then operates on them. The code in planC uses shared memory only to re-arrange the second of each object pair loaded. Read all parts of this problem before attempting to answer the questions below.

```c
__global__ void planA(int num_pairs)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int num_threads = blockDim.x * gridDim.x;
    for ( int i=tid; i<num_pairs; i+=num_threads )
    {
        int2 pair = pairs[i];
        float3 pos_1 = pos_global[pair.x];
        float3 vel_1 = vel_global[pair.x];
        float3 pos_2 = pos_global[pair.y];
        float3 vel_2 = vel_global[pair.y];
        float3 f = resolve(pos_1,vel_1,pos_2,vel_2);
        pairs_f[i] = f;
    }
}
```

(a) The performance of planA depends upon the array pairs. Suppose array pairs contains the correct pairs of objects to compare but nothing else was taken into account. Why might performance be poor? Hint: This is due to what could be called the first rule of global memory access.

- Reason for poor performance if pairs ill-chosen.

(b) For planA, what can be done to the pairs array to improve performance? That is, starting with an existing array old_pairs compute a new array of the same size, pairs, such that if element i of old_pairs is (a, b) (old_pairs[i].x=a and old_pairs[i].y=b) then there is some element j of pairs which includes (a, b) or (b, a). Hint: The solution is simple but not 100% effective.

- Explain how pairs should be constructed.

- Estimate the improvement in performance, be as specific as possible.
Problem 3, continued:

(c) By copying object data to shared memory the code in _planB_ potentially solves two problems related to global memory access. The array _prefetch_idx_ stores indices of objects to prefetch into shared memory, the array _pairsB_ indicates which pairs of objects to _resolve_, but refers to them using shared memory indices (rather than global memory indices).

```c
__device__ void planB(int num_pairs)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int num_threads = blockDim.x * gridDim.x;
    __shared__ float3 pos_shared[OBJ_PER_BLOCK], vel_shared[OBJ_PER_BLOCK];

    // Prefetch Loop
    for ( int j=threadIdx.x; j<OBJ_PER_BLOCK; j += blockDim.x )
    {
        int gidx = j + OBJ_PER_BLOCK * blockIdx.x;
        int bidx = prefetch_idx[gidx];
        pos_shared[j] = pos_global[bidx];
        vel_shared[j] = vel_global[bidx];
    }

    // Resolve Loop
    for ( int i=tid; i<num_pairs; i += num_threads )
    {
        short2 pair = pairsB[i];
        float3 pos_1 = pos_shared[pair.x];
        float3 vel_1 = vel_shared[pair.x];
        float3 pos_2 = pos_shared[pair.y];
        float3 vel_2 = vel_shared[pair.y];
        float3 f = resolve(pos_1, vel_1, pos_2, vel_2);
        pairs_f[i] = f;
    }
}
```

- Insert the minimum number of _syncthreads_() calls in the code above needed for correct operation.

- Given the use of _prefetch_idx_, how much more efficiently does the code in _planB_ _Prefetch Loop_ access global memory than the code in _planA_? Assume that _prefetch_idx_ and _pairsB_ are properly prepared (but the list of pairs is still uncontrollable). Is access ideal, or does it still depend on something?

- When ordering _pairsB_ one can emphasize reducing the total access to global memory or the efficiency of global memory access. Explain how each might be emphasized.
Problem 3, continued: The code below is for planC. It uses less shared memory than planB.

```c
__global__ void planC(int num_pairs)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int num_threads = blockDim.x * gridDim.x;

    for ( int i=tid; i<num_pairs; i+=num_threads )
    {
        __shared__ float3 rearrange[BLOCK_SIZE];
        int3 pair = pairs[i];
        float3 pos_1 = pos_global[pair.x];
        float3 vel_1 = vel_global[pair.x];
        rearrange[pair.z] = pos_global[pair.y];
        float3 pos_2 = rearrange[threadIdx.x];
        float3 vel_2 = rearrange[threadIdx.x];
        float3 f = resolve(pos_1,vel_1,pos_2,vel_2);
        pairs_f[i] = f;
    }
}
```

- Insert the minimum number of `syncthreads()` calls so that the code above runs correctly.
- Compare the efficiency of global memory access in planC to the other two versions. Which is more efficient?

Assume that, in planB’s favor, an object appears in multiple pairs (and so planB’s use of shared memory has potential benefit). By using less shared memory then planB, planC can potentially run faster than planB. Explain why, indicating a disadvantage planB has and how planC is faster despite the fact that planB caches objects.