Name Solution

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 Methane?

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 ⟸ I ];
  float rad = center_o_rad.w;
  vec3 to_surface = rad * gl_Vertex.xyz ⟸ I ;
  vec4 position_o = vec4(center_o_rad.xyz + to_surface, 1f);

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

(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.

Uniform sizes: `gl_ModelViewProjectionMatrix` and `gl_ModelViewMatrix`: since each is a 4×4 matrix, 16 floats each; `gl_NormalMatrix`: since it is a 3×3 matrix, 9 floats. Total: $4(32 + 9) = 164\,\mathrm{B}$.

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

Because `gl_InstanceID` is a consecutive numbering of instances it is something that the GPU can easily generate, and so there is no need to send values for this input from the CPU to the GPU. Input `gl_Vertex` is sourced from a buffer object. Recall that the buffer object contains the vertex coordinates of a sphere of radius 1 centered at the origin. We expect that this buffer object would be sent from the CPU to the GPU just once, and used as many times as needed. So in a typical pass the amount of data sent would be zero.

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

Lets assume that the ball positions change each rendering pass but their colors remain the same. Amount of data: $16s\,\mathrm{B}$.

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`?

Short Answer: The modelview matrix.

Explanation: In all of the classroom examples and homework assignments, the vertices for a sphere are computed for a sphere of radius 1 with its center at the origin. These vertex coordinates and associated normals and texture coordinates are usually placed in buffer objects once, during initialization.

To show the sphere in the desired location at the desired size the usual way of doing things in class was to set the modelview matrix. This is the approach used in `vs_main`. One rendering pass is needed for each sphere rendered, and the modelview matrix would be set before each pass.

In contrast, with an instanced draw one can render multiple spheres in a single rendering pass. The pre-defined variable `gl_InstanceID` indicates which sphere (numbered starting at 0) is being rendered. Because the same modelview matrix is used during an entire rendering pass another method is needed to specify the size and position of each sphere. The code in `vs_main_instances` gets the size and position from the array `balls_pos_rad` and uses that to transform the local sphere coordinate in `gl_Vertex` (by doing a vector add rather than a matrix/vector multiply). Note that this method does not rotate the sphere, but that should not be a problem since texture coordinates are presumably not being used.

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

The size of the modelview matrix is $4 \times 4 = 16$ floats, the size of the normal matrix is $3 \times 3 = 9$ floats. They are sent once per sphere. The size of an element of `balls_pos_rad` is 4 floats. So the original code uses $\frac{16+9}{4} = 6.25\times$ more data.

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

☑ Higher performance because ...

. . . the overhead of performing a rendering pass is incurred once per n balls rather than n times per n balls (spheres).

Long Answer: The original method does send $6.25\times$ more data, but that's still just 25 bytes per ball so the amount of data will not have a significant impact. The instanced draw does more computation: it needs to perform three multiplies (`rad *` gl_Vertex.xyz) that the original code does not have to do. That's in the vertex shader and is dwarfed by the amount of computation needed to do lighting computations in the fragment shader. So the additional computation needed for the instanced draw is also not a significant factor. Also, each vertex shader invocation needs to read the 16 bytes from balls_pos_rad, but the invocations sharing an instance ID will read the same element, and so this will add little time. It's likely that the time to set up a rendering pass will be much larger than these other differences.

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 the maximum number of threads 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.

Short Answer: It would take $2 \times 27 = 54$ s because one SM would be assigned two blocks and those blocks would run sequentially.

Long Answer: In this particular case, because of the thread limit, only one block at a time can run on an SM. When the kernel is launched each of the first 9 blocks will be sent to an SM. The tenth block will have to wait for one of the first nine blocks to finish. Assuming that all blocks take about the same amount of time, that should occur in 27 s. The tenth block will then finish 27 s later, for a total time of 54 s.

(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.

Short Answer: The run time would be $\frac{10}{9} 27\,\mathrm{s} = 30\,\mathrm{s}$.

Long Answer: Since the programmer is skilled and since the number of blocks matches the number of SMs, we would expect that the run time would be proportional to the amount of work divided by the amount of computing resources. If the amount of computing resources (the number of SMs) changes from 10 to 9 we would expect run time to increase by a factor of $\frac{10}{9}$ yielding a run time of $\frac{10}{9} 27\,\mathrm{s} = 30\,\mathrm{s}$.

(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.

It would still take about 72 s because with 32-thread blocks it's possible for two blocks to run simultaneously on the same SM. Since 32 threads are not nearly enough to fully utilize an SM's resources (for example, the 128 single-precision floating-point units that NVIDIA calls CUDA cores on a CC 4.X device) the two blocks can run at the same time and so they will take the same amount of time to run as they would if each were run on a separate SM, assuming that the code was not data-limited. (We can assume it's not data limited in this case because there are only 32 threads.)

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.

Because of the change every block does the computation that block 0 would do in the original code. Since work is divided evenly, and because the number of blocks hasn't changed, the execution time should not change by much. The only possible reason for an increase in execution time is that each block writes the same array elements. However, the effect of this should be small because the number of writers to a particular location is equal to the number of active blocks (which at most would be on the order of 160 [with a block occupancy of 16 and 10 SMs]) and because for multiple writes to the same location to be a problem those writes would have to occur within cycles of each other, which is unlikely for even two writers because any of a number of factors could cause blocks on different SMs to be at different places in the code such as not starting at exactly the same time.

The results would be incorrect because the computation performed by blocks $1, 2, \ldots$ is not being done.

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 ...

Adjacent threads in a warp read global memory locations that differ by 128 bytes, and so a separate request will be needed for each thread. The minimum request size is 32 bytes and only one byte of the request is used (`in_data` is an array of `char`s, which are one byte). Since only 1 out of 32 bytes are used the code reads $32\times$ more data than is needed.

More Details: To see how adjacent threads are reading global addresses that are 128 bytes apart compute `start_id+i` for two cases: `threadIdx.x` = 0 and for `threadIdx.x` = 1. In both cases set `blockIdx.x` = 0 and `i` = 0. For thread 0: `start_id+i` = 0+0 = 0 and for thread 1 `start_id+i` = 1*128+0 = 128. So the distance between accesses is 128 elements, or 128 B because each element is one byte. It should be easy to convince yourself that the distance between accesses is 128 when comparing any `threadIdx.x` and `threadIdx.x+1`. Keep in mind that within a block (and so a warp) the two threads will have the same `blockIdx.x` and the same `i`. (The same `i` because within a warp they must be at the same point in execution.)

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

☑ What is the value of x?

Suppose that the size of an `int` is four bytes (which is common). Then the thread would be using 4 of 32 bytes, and so it would only be reading $\frac{32}{4} = 8\times$ more data than necessary. That is, $x = 8$.

(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`.

They are: 40, 70, 200, 110, 240, 270, 310.

☑ Explain.

At the first `__synchthreads` we know for certain that `sum` is zero. Statements like `sum += 40;` consist of a shared memory read, an add, and a shared memory write. Let R40 denote the shared memory read performed by thread 40, and define W40, R70, etc. similarly. Perhaps we would like the ordering of these events to be R40, W40, R70, W70, R200, W200. If so `sum` would be set to 310. All we know for certain is that within a thread the write will come after the read. But the ordering between threads is not guaranteed because each of the participating threads is on a different warp. Consider the following possible ordering: R40, R70, R200, W200, W70, R40. In this case all threads read zero, and since 70 is the last to write, `sum` will be set to 70. Consider another possible ordering: R70, W70, R40 R200, W200, W40. Here, threads 40 and 200 will read the 70 written by thread 70, since 40 is the last to write `sum` will be 110.

Here are all possibilities: 40, 70, 200, 110, 240, 270, 310.

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

☑ In general, what does `__syncthreads` do?

It causes each thread to wait until all threads in the block reach `__syncthreads`, after which threads can proceed. In class this was described as a room with an entrance door and an exit door. Initially the entrance door is open. Threads enter the room when their execution reaches `__syncthreads`. When all of the threads in the block are in the room the entrance door closes and the exit opens, which means that execution can proceed to the statement after `__syncthreads`.

☑ 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];
```

Thread 31 might read element 32 of `our_data` before thread 32 writes it.

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

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

All threads in the block must execute `__syncthreads`. In the example above thread 20 does not execute it and so the threads that do execute `__syncthreads` will be stuck in `__syncthreads`.

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

```
layout ( triangles ) in;
layout ( triangle_strip, max_vertices = 6 ) out; // SOLUTION 3 -> 6

void gs_main_simple() {

  /// SOLUTION
  vec3 tnorm = normalize( cross                              // Filled In
    ( In[1].vertex_e.xyz - In[0].vertex_e.xyz,
      In[2].vertex_e.xyz - In[1].vertex_e.xyz ) );

  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();

 /// SOLUTION:
  color = vec4(0,0,1,1); // Blue
  normal_e = tnorm;

  for ( int i=0; i<3; i++ ) {
      vertex_e = In[i].vertex_e + tnorm;
      gl_Position = gl_ProjectionMatrix * vertex_e;
      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.

Solution appears above.

(*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.

Solution appears above. Notice that the `color` and `normal_e` shader outputs are only assigned once since they don't change. Also notice that the eye-space coordinate of the new triangle, `vertex_e`, had to be converted to clip space, something which is done in the vertex shader for the vertices of the original triangle. In the layout declaration `max_vertices` was increased from 3 to 6.

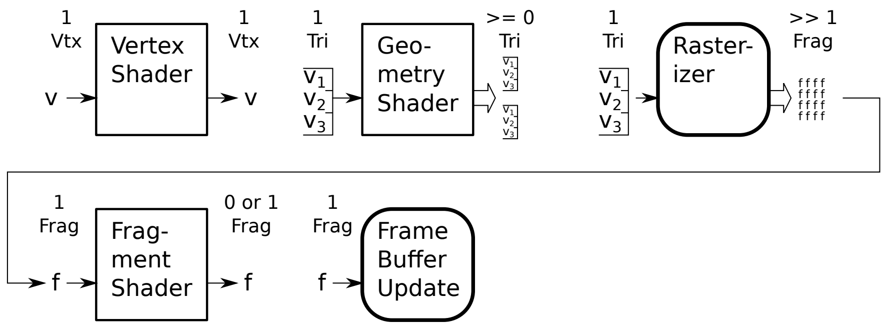
(*c*) What about `normal_e` for the new triangle? Suppose we are rendering a sphere. Compare the appearance of the new triangle with `normal_e` set to `tnorm` to the appearance when `normal_e` is set to `In[i].normal_e`.

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

With `normal_e = tnorm` the sphere will look like it is covered with triangles, that's because two nearby points on different triangles will have different normals and so will be lighted differently, making the triangle boundaries obvious. In contrast, if the normal is based on the sphere normal at those two nearby points, then the two normals will be almost the same and the lighting will be almost the same, making it hard to see the triangles.

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?

It is performed in the frame buffer update stage.

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

Because the vertex shader only has the coordinate of one vertex, not the other vertices (if any) in the primitive. Even if it had this information, that would not help because some fragments might pass the depth test while other fragments would not pass. It would also force the vertex shader to compute the location of each fragment, reproducing the work that the rasterizer does.

(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?

Because in a triangle strip a single vertex, and so the work of a single vertex shader invocation, can be used for up to three triangles.

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

Because the number of primitives, triangles, is the same, and so the number of geometry shader invocations will be the same. Because the number and size of the triangles sent to the rasterizer are the same in both cases the number of fragment shader invocations will be the same.