GPU Programming

EE 4702-1

Final Examination

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

Partial Solution

Problem 1 _______ (20 pts)
Problem 2 _______ (15 pts)
Problem 3 _______ (15 pts)
Problem 4 _______ (20 pts)
Problem 5 _______ (20 pts)
Problem 6 _______ (10 pts)

Alias Methane? __________________________ Exam Total _______ (100 pts)

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 \( \text{glDrawArraysInstanced(type,0,s,n)} \) sends a set of \( s \) vertices for rendering \( n \) times.

```c
// CPU code for rendering pass.

GLuint balls_pos_rad_bo, balls_color_bo, sphere_points_bo;

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;
}
```

(a) In the shader code above indicate which variables are uniforms and which are vertex shader inputs. 

```
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: MVP and MV: 16 floats each, NM: 9 floats. Total: \( 4(32 + 9) = 164 \) B.

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

Assume that \( \text{gl_InstanceID} \) not sent from cpu to gpu ever. Input \( \text{gl_Vertex} \) is sourced from a buffer object, and that doesn’t change. So, 0.

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: \( 16s \) 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.

```cpp
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, an 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 \(16 + 9 = 25\) more data.

**Higher performance because ...**

... the overhead of performing a rendering pass is incurred once rather than once per ball (sphere).

**Long Answer:** The original method does send \(25\times\) 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 (\(\text{rad} * \)
That that's in the vertex shader and is dwarfed by the amount of
computation needed to do lighting computations in the fragment shader. So computation also is not a significant factor. Also, each
vertex shader invocation needs to read the 16 bytes from \texttt{balls.pos.rad}, but the invocations sharing an instance \texttt{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 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.

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

(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.

The run time would be $\frac{10}{9} 27 = 30$ 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.
Problem 3: [15 pts] The CUDA kernels below are based on a classroom example. Notice that in both cases the computation of \( \text{tid} \) has been changed. For one of the kernels the change will result in longer execution time though the result will still be correct. For the other kernel there will be little change in execution time but the result will be wrong.

(a) Consider the simple kernel:

```c
__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 ☐ longer or ☑ about the same.

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

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

(b) Consider the efficient kernel:

```c
__global__ void cuda_thread_start_efficient() {
    // 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<array_size; h += num_threads ) {
        float4 p = d_v_in[h];
        d_m_out[h] = dot( p, p );
    }
}
```

☑ Due to change, execution time ☐ longer or ☑ about the same.

☑ 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.

```c
__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, nor did the first checkbox item.

```c
__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 the one correct value for `sum` at the last line.

☑ Show at least four different possible values for `sum` that the code above can write to `out.data`.

☑ Explain.

40, 70, 200, 110, 240
(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?

```c
__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?

```c
if ( threadIdx.x != 20 ) __syncthreads();
```
Problem 5: [20 pts] The geometry shader below, from Homework 4, passes a triangle unchanged.

```glsl
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) is 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 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 $3 \times$ 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.