Name Solution

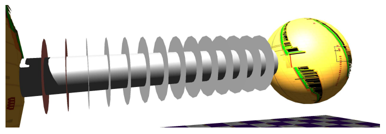
GPU Programming
EE 4702-1
Final Examination

Monday, 5 December 2016 17:30–19:30 CST

Problem 1		(20 pts)
Problem 2		(20 pts)
Problem 3		(15 pts)
Problem 4		(20 pts)
Problem 5		(25 pts)
Exam Total		(100 pts)

Alias Methane?

Good Luck!



Problem 1: [20 pts]

The geometry shader code below is based on the solution to Homework 6 Problem 2 in which a curved link is rendered by having the vertex shader compute points and vectors related to a curve and by having the geometry shader, whose input primitive type is a line strip, emitting a cylinder between the locations described by its two input vertices. Input `In[].t` is the position along the curve, with $0 \le t \le 1$.

Add code to the shader so that it draws rings around the cylinder. The inner radius should be the same as the cylinder, the outer radius should be twice the cylinder radius. Do not emit a ring at the link endpoints (the parts that touch the spheres). See the illustration to the upper right.

✓ Complete the code so that it draws the rings. ✓ Can use suggested and other reasonable abbreviations.

✓ Set `normal_e` properly don't forget to set ✓ `vertex_e` and ✓ `gl_Position`.

✓ The code should be reasonably efficient. ✓ Don't overlap primitives.

Solution appears below. The solution code is in the repo in directory `hw/gpup/2016/fe`.

```
void gs_main_2() {
  const float rad = tex_rad[In[1].iid].z,  sides = sides_rad.x;
  for ( int j=0; j<=sides; j++ ) {
      const float theta = j * ( 2 * M_PI / sides );
      vec3 vect0 = cos(theta) * In[0].norm_e + sin(theta) * In[0].binorm_e;
      vec3 vect1 = cos(theta) * In[1].norm_e + sin(theta) * In[1].binorm_e;

      normal_e = vect1;    vertex_e = In[1].ctr_e + vec4( rad * vect1, 0);
      gl_Position = gl_ProjectionMatrix * vertex_e;  // SUGGESTED ABBREV: glP = glPM * v_e;
      EmitVertex();
      normal_e = vect0;    vertex_e = In[0].ctr_e + vec4(rad * vect0,0);
      gl_Position = gl_ProjectionMatrix * vertex_e;
      EmitVertex();   }
  EndPrimitive();
  float t0 = In[0].t,      t1 = In[1].t;       // Use if needed.
  vec4  c0 = In[0].ctr_e,  c1 = In[1].ctr_e;   // Use if needed.

  if ( In[0].t == 0 ) return;  // SOLUTION: Don't render ring at 1st segment.
  normal_e = cross(In[0].norm_e,In[0].binorm_e);  // SOLUTION: Normal is cross of axes.
  float rad2 = rad * 2;        // SOLUTION: Outer radius of ring. (Inner radius is rad)

  for ( int j=0; j<=sides; j++ ) {
      const float theta = j * ( 2 * M_PI / sides );
      vec3 v0 = cos(theta) * In[0].norm_e + sin(theta) * In[0].binorm_e; // Use if needed.
      vec3 v1 = cos(theta) * In[1].norm_e + sin(theta) * In[1].binorm_e; // Use if needed.

      vertex_e = In[0].ctr_e + vec4(rad * v0,0);     // SOLUTION: Inner edge of ring.
      gl_Position = gl_ProjectionMatrix * vertex_e;  // SOLUTION
      EmitVertex();                                  // SOLUTION

      vertex_e = In[0].ctr_e + vec4(rad2 * v0,0);    // SOLUTION: Outer edge of ring.
      gl_Position = gl_ProjectionMatrix * vertex_e;  // SOLUTION
      EmitVertex();                                  // SOLUTION

    }
}
```

Problem 2: [20 pts] Appearing below is a typical vertex shader.

```
in vec2 gl_MultiTexCoord0;  // Declare size.
void vs_main() {
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  vertex_e = gl_ModelViewMatrix * gl_Vertex;
  normal_e = normalize(gl_NormalMatrix * gl_Normal);
  gl_TexCoord[0] = gl_MultiTexCoord0;
  color = gl_Color;
}
```

(*a*) Suppose that the shader above was used in a rendering pass of v vertices. No buffer objects were used and all shader inputs were sourced from client arrays. Compute the amount of data sent from the CPU to the GPU for the rendering pass.

☑ CPU to GPU data for a rendering pass of v vertices counting shader inputs is:

Short Answer: counting `gl_Vertex`, `gl_Normal`, `gl_MultiTexCoord0`, and `gl_Color` the amount of data is

$$4(4+3+2+4)v\,\mathrm{B} = 52v\,\mathrm{B}.$$

☑ CPU to GPU data for a rendering pass of v vertices counting uniform variables is:

Short Answer: counting `gl_ModelViewProjectionMatrix`, `gl_ModelViewMatrix`, and `gl_NormalMatrix` the amount of data is $4(4^2+4^2+3^2)\,\mathrm{B} = 164\,\mathrm{B}$.

Explanation: Uniforms are sent once.

Problem 2, continued: The vertex shader below is used for an instanced draw of spheres. The vertex shader inputs are sourced from buffer objects and other buffer objects are accessed directly using the instance ID.

```
layout ( binding = 1 ) buffer sr { mat4 sphere_transform[]; };
layout ( binding = 3 ) buffer sc { vec4 sphere_color[]; };
void vs_main_instances_sphere() {
  mat4 transform = sphere_transform[gl_InstanceID INP];
  vec3 ctr = transform[3].xyz;         // Extract coordinate of sphere center.
  float radius = transform[3][3];      // Extract sphere radius.
  mat3 rot = mat3(transform);          // Extract rotation matrix.

  vec3 normal_o = rot * gl_Vertex.xyz INP;
  vec4 vertex_o = vec4( ctr + radius * normal_o, 1 );

  color = sphere_color[gl_InstanceID INP];
  gl_Position = gl_ModelViewProjectionMatrix UFM * vertex_o;
  vertex_e = gl_ModelViewMatrix UFM * vertex_o;
  normal_e = normalize(gl_NormalMatrix UFM * normal_o );
  gl_TexCoord[0] = gl_MultiTexCoord0 INP;  }
```

(*b*) In the code above label the vertex shader inputs and uniform variables.

☑ Label vertex shader inputs with a I and uniform variables with a U.

Solution appears above in blue. To make the labels easier to find vertex shader inputs are labeled INP and uniforms are labeled UFM.

(*c*) Compute the amount of data sent from the CPU to the GPU for an instanced draw rendering pass in which n instances (spheres) are rendered, each consisting of v vertices. Assume that initially all buffer objects are on the CPU. Note that `mat4` is a 4×4 matrix of floats.

☑ Amount of CPU to GPU data for n instances of v vertices, accounting for ☑ uniforms, ☑ buffer objects, ☑ and anything else that really needs to be moved.

Short Answer: $164\,\mathrm{B}$ for uniforms (see part a), $n(4^3+4^2)\,\mathrm{B} = 80n\,\mathrm{B}$ for bound buffer objects, and $v(4\times4+4\times2)\,\mathrm{B} = 24v\,\mathrm{B}$ for shader inputs.

Explanation: The vertex shader inputs are **gl_InstanceID**, **gl_Vertex**, **gl_MultiTexCoord0**. The size of a **gl_Vertex** is $4 \times 4\,\mathrm{B}$ and assuming a 2-component texture coordinate the size of **gl_MultiTexCoord0** is $4 \times 2\,\mathrm{B}$. It is reasonable to assume that the data for **gl_Vertex** and **gl_MultiTexCoord0** were put in buffer objects and sent to the GPU. The size of the two buffer objects would be $v(4\times4+4\times2)\,\mathrm{B} = 24v\,\mathrm{B}$. Input **gl_InstanceID** is just an ID number created by the GPU software, it's not something sent from the CPU to the GPU, and so zero data is sent for **gl_InstanceID**. (That is, it would be silly to send an array of values $0, 1, \ldots, n-1$ from the CPU to the GPU, the GPU could generate such values with a simple loop. All that's needed is the number of instances, n.)

The shader accesses two bound buffer objects, **sphere_transform** and **sphere_color**. Based on the layout declaration each element of **sphere_transform** is $4 \times 4 \times 4\,\mathrm{B}$ and each element of **sphere_color** is $4 \times 4\,\mathrm{B}$. Since these arrays are indexed using **gl_InstanceID** it is reasonable to assume that they each have n elements. So the total amount of data for the bound buffer objects is $n(4^3 + 4^2)\,\mathrm{B} = 80n\,\mathrm{B}$.

The shader accesses three uniforms, the same uniforms as are accessed in part a. So their total data is $164\,\mathrm{B}$.

Problem 3: [15 pts] Consider our well written CUDA example (slightly simplified):

```
__global__ void kmain_efficient() {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int n_threads = blockDim.x * gridDim.x;

  // Note: There's no need to understand this code. Just remember
  // that each thread does about the same amount of work.

  for ( int h=tid; h<d_app.array_size; h += n_threads ) {
      float4 p = d_app.d_in[h];
      d_app.d_out[h] = dot(p,p);
    }
}
```

Let n denote the value of array_size, and let S denote the number of streaming multiprocessors (abbreviated SMs and sometimes MPs). Assume that n is large so that there's plenty of work to do for each thread.

(a) Determine a launch configuration in terms of n and S that uses the minimum number of blocks needed to maximize warp (or thread) occupancy on the SMs. Do this for a device with a block size limit of 1024 threads, an SM limit of 2048 threads, and an SM limit of 16 blocks. *Hint: There's no need to use both n and S.* Use B for the block size and G for the number of blocks (grid size).

✓ Block size $B = 1024$

✓ Explain.

Short Answer: Use the maximum block size to minimize the number of blocks.

To maximize warp (thread) occupancy we need to have 2048 threads per SM. Since we need to *minimize* the number of blocks, we should make the blocks as large as possible. The problem states that the maximum block size is 1024 threads, so that's the size used.

✓ Grid size (number of blocks) $G = 2S$

✓ Explain.

It takes two 1024-thread blocks per SM to achieve full occupancy, so for all S SM's the number of blocks needed is $2S$.

(b) Let B and G be a solution to the problem above (meaning a good choice for B and G for S SMs) and let t_o be the execution time in a kernel launch of that configuration. Estimate the execution time for the configurations below. *Hint: Remember that G was chosen to maximize the number of warps per SM.*

✓ Estimate the execution time for a launch of G blocks of size $B - 1$. ✓ Explain.

Short Answer: assuming execution time is limited by computation, scale up the time: $\frac{B}{B-1}t_o$. Since the code above is bandwidth-limited the time would likely still be t_0 since the same amount of data is read.

Explanation: threads are assigned to hardware in groups of 32 (in NVIDIA devices so far) called warps. If a block has 1023 threads there will be 31 warps with 32 threads, and one warp with 31 threads. There is no way to give the hardware that would be used by the 1024'th thread to the 1023 remaining threads so the remaining threads work as fast as they would have before, but now they have slightly more work, $\frac{B}{B-1}$ times as much as before. If execution time were determined by the amount of work per thread execution would take $\frac{B}{B-1}$ times longer.

✓ Estimate the execution time for a launch of $G - 1$ blocks of size B. ✓ Explain.

Note: This answer went beyond what was covered in 2017. Short Answer: assuming execution time is limited by computation and 2048 threads were needed to hide latency, scale up the time: $\frac{2S}{2S-1}t_o$. Since the code above is bandwidth-limited the time would likely still be t_0 since the same amount of data is read. Either way the execution time doesn't change much.

✓ Estimate the execution time for a launch of $G+1$ blocks of size B. ✓ Explain.

Short Answer: $2t_0\frac{2S}{2S+1}$ because the one extra block must wait for the first $2S$ blocks to finish.

Explanation: Since $G = 2S$ blocks fully occupy a GPU, in a launch of $2S+1$ blocks the first $2S$ blocks first run, when they are finished the one remaining block starts running. Even though that one last block has the whole GPU to itself, it will run no faster (under certain assumptions) than if it were running with others. The amount of work done by each block in a $2S+1$-block launch in comparison to a $2S$-block launch is $\frac{2S}{2S+1}$, slightly less work. The first group of $2S$ blocks finishes in $\frac{2S}{2S+1}t_0$ time, and then the last block starts and it also takes $\frac{2S}{2S+1}t_0$. Note that this one extra block made things alot worse.

Problem 4: [20 pts] Consider the CUDA kernel below. Let n denote the value of array_size, B the block size, and G the number of blocks.

```
__global__ void prob_x(float2 *d_in, float *d_out) {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int n_threads = blockDim.x * gridDim.x;

  for ( int h=tid; h<d_app.array_size-1; h += n_threads ) {
      float2 p = d_in[h];
      float2 q = d_in[h+1];
      d_out[h] = dot(p,q);
    }
}
```

(a) Compute how much data is read from global memory during the execution of the kernel. Note that each element d_in[i] is accessed by two threads.

☑ Amount of data read:

☑ Explain.

About $2n2 \times 4\,\mathrm{B} = 16n\,\mathrm{B}$, twice the size of the array.

In a system with a cache the data read by thread τ in the d_in[h] access would be placed in a cache where it would be found by thread $\tau - 1$ when it performs the d_in[h+1] accesses. (Or, $\tau - 1$ might bring in data for τ.) The total amount of data would be $8n\,\mathrm{B}$ (each element is $2 \times 4\,\mathrm{B}$).

But, recent GPU generations (Kepler to Maxwell) don't automatically cache data, and even when they do the chances of a hit are smaller. For that reason both the d_in[h] and the d_in[h+1] accesses would bring in data from global memory (or maybe the L2 cache), doubling the amount of data read: $16n\,\mathrm{B}$.

What can be done about it? Shared memory could be used so that thread τ could read data loaded by thread $\tau + 1$. Or, the code could be written so that the accesses to d_in use the read-only cache (available starting at Kepler 3.5).

(*b*) The kernel below includes a declaration for shared memory intended to fix the inefficiency in the kernel above. Complete the kernel.

☑ Use shared memory to avoid waste.

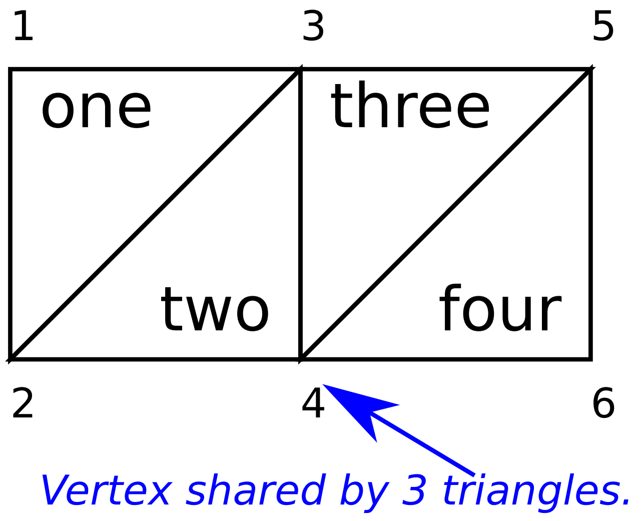
```
__global__ void prob_x2(float2 *d_in, float *d_out) {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int n_threads = blockDim.x * gridDim.x;

  __shared__ float2 sm[1025];

  for ( int h=tid; h<d_app.array_size; h += n_threads ) {
      float2 p = d_in[h];

      // SOLUTION
      __syncthreads();
      sm[threadIdx.x] = p;
      if ( threadIdx.x == blockDim.x - 1 ) sm[threadIdx.x+1] = d_in[h+1];
      __syncthreads();

      float2 q = sm[threadIdx.x+1];  // SOLUTION
      d_out[h] = dot(p,q);
    }
}
```

Problem 5: [25 pts] Answer each question below.

(*a*) Show a diagram illustrating the relationship between vertices and primitives in a triangle strip. Number both the vertices and the triangles in the correct order. Show at least three triangles.

☑ Diagram with number vertices and triangles illustrating a triangle strip.

Solution appears below.



(*b*) Describe what is tested in the stencil test and what the stencil test might be used for.

☑ A stencil test checks:

The stencil test, applied to a fragment, compares a reference value provided in the OpenGL `glStencilFunc` call with the value in the stencil buffer location corresponding to the fragment. The stencil buffer is one of several frame buffer layers recognized by OpenGL. Others include the color buffers (which connect to a display) and the depth buffer.

☑ Typical use for a stencil test:

A stencil test might be used for a mirror effect. The stencil buffer is written in a pass where the primitive corresponds to mirror locations. Then in a subsequent pass other objects are rendered with a transform that maps their coordinates to their reflected location. The stencil test will only allow fragments from these objects to be written in the location at which there are mirrors.

(*c*) Interpolation qualifiers, `flat`, `noperspective`, and `smooth`, are used for the inputs of which rendering pipeline stage? What do they do?

☑ They appear at the input to: the fragment shader.

☑ They affect the qualified variable by:

Determining how to combine the values of the variable at the primitives vertices. For `flat` the value provided to the fragment shader is the value emitted by the provoking (last) vertex of the primitive. For `noperspective`, the value is linearly interpolated between the values at the vertices based on clip space coordinates. For `smooth`, the value is linearly interpolated base on object space coordinates (effectively). The `noperspective` interpolation is not geometrically accurate but is less costly to compute.

(*d*) Between which coordinate spaces does the OpenGL projection matrix transform.

☑ The projection matrix transforms coordinates ☑ from ☑ to

From eye space to clip space.

(*e*) What is the difference between a material property (including attributes emitted with `glColor3f`) and a lighted color?

☑ Difference between material property and lighted color?

A material property is a color given to the OpenGL rendering pipeline, it can apply to all vertices (as a uniform) or just one (as a vertex shader input). The rendering pipeline typically computes a lighted color by using the material property as well as light locations, the normal associated with the vertex, and other factors. If material properties were used instead of lighted colors then there would be no cues about shape and so, a sphere would look like a circle.