Name Solution

EE 4702

Final Exam

Friday, 7 December 2012, 12:30-14:30 CST

Problem 1		(15 pts)
Problem 2		(15 pts)
Problem 3		(20 pts)
Problem 4		(15 pts)
Problem 5		(15 pts)
Problem 6		(20 pts)
Exam Total		(100 pts)

Alias 0x501f

Good Luck!

Problem 1: [15 pts]The two code fragments below render the same set of triangles.

```
// Code Fragment A

 glBegin(GL_TRIANGLES);
 for ( int i=0; i<1000; i++ )
   {
     glColor3fv(lsu_spirit_gold); glVertex3fv(v0[i]);
     glColor3fv(lsu_spirit_gold); glVertex3fv(v1[i]);
     glColor3fv(lsu_spirit_gold); glVertex3fv(v2[i]);
   }
 glEnd();

// Code Fragment B
 glColor3fv(lsu_spirit_gold);
 glBegin(GL_TRIANGLE_STRIP);
 for ( int i=0; i< XXX; i++ )
     glVertex3fv(v[i]);
 glEnd();
```

(*a*) Compute the amount of data sent from CPU to GPU for each case. Use an appropriate value of XXX.

For Fragment A:

$$1000 \times 3 \times 2 \times 3 \times 4 = 72000\,\mathrm{B}$$

where the factors are: 1000 triangles, 3 vertices per triangle, two attributes (color and vertex coordinate), three components per attribute, four bytes per component.

For Fragment B, the color is sent once (not per vertex), the size is $3 \times 4 = 12$ bytes. If the same number of triangles are rendered then 1002 vertices need to be sent. The data size is $12 + 1002 \times 3 \times 4 = 12036$ bytes.

Problem 2: [15 pts]Consider the declarations below which come from the vertex shader in the Homework 3 solution.

```
layout ( location = 2 ) uniform float wire_radius;
layout ( location = 3 ) uniform float theta;
layout ( location = 4 ) uniform vec2 texture_scale;
layout ( binding = 1 ) buffer Helix_Coord  { vec4  helix_coord[];  };
layout ( location = 1 ) in ivec2 helix_index;
out vec3 normal_e;
uniform sampler2D tex_unit_0;
```

(*a*) What do the location numbers indicate in the uniform declarations?

They are numbers by which the CPU identifies the respective uniform. The CPU uses these numbers when writing a value to a uniform.

Shown below is CPU code to write a value into the `wire_radius` uniform declared above (in shader code). It is the "2" which tells OpenGL which uniform to deposit the value in. It has nothing to do with the names of the variables, `wire_radius` on the GPU and wireRadius on the CPU.

```
// CPU Code
glUniform1f( 2, wireRadius);
```

(*b*) How is a `uniform` typically used?

It set once before a rendering pass and does not change. One can assume that changing a uniform value is relatively costly.

(*c*) Explain what the `in` and `out` qualifiers used for `helix_index` and `normal_e` indicate.

The `in` indicates that the variable is an input to the vertex shader. Its values are provided by the CPU using the `glVertexAttribX` command. Like colors, normals, and other attributes, a different value can be provided for each vertex. The `out` indicates that the variable is an output from the vertex shader, the vertex shader code must write the variable.

(*d*) Why can't the declarations above be for a geometry shader? (That is, something won't work.)

Because the inputs to a geometry shader must be declared as arrays.

(*e*) Why can't the declarations above be for a fragment shader? (That is, something won't work.)

Because fragment shaders don't have user-declared outputs.

Problem 3: [20 pts]The Homework 5 code simulated a spring, bringing to life the helix we were working with most of the semester. A combination of CPU and GPU code was used to compute the wire surface coordinates. Excerpts of the code appear below.

Grading Note: This problem was based on code used in Homework 5, with which some familiarity was expected.

```
Physics Simulation in CUDA computes helix_position and orientation, read to CPU.
              Wire_Segment* const ws = &wire_segments[i];
              ws->position = helix_position[i];
              ws->orientation = helix_orientation[i];

Computation on CPU:
          helix_coords[i] = ws->position;
          pMatrix_Rotation c_rot(ws->orientation); // Mult: 3 * 9 = 27
          helix_u[i] = c_rot * pVect(0,0,1);
          helix_v[i] = c_rot * pVect(0,1,0);

Computation on Vertex Shader
  vec3 vtx = helix_coord[hidx.x].xyz;
  float pi = 3.14159265;
  float theta = hidx.y * 2 * pi / 20;
  vec3 u = helix_u[hidx.x].xyz;
  vec3 v = helix_v[hidx.x].xyz;
  vec3 normal = normalize( cos(theta) * u + sin(theta) * v );
  // Compute wire surface location by adding normal to helix coordinate.
  //
  vec4 vertex_o;
  vertex_o.xyz = helix_coord[hidx.x].xyz + wire_radius * normal;
  vertex_o.w = 1;
```

Computing the u and v vectors on the CPU requires about 18 multiplications. Consider the method used in the code above and two alternatives: computing these in CUDA or computing them in the vertex shader.

The values of `hidx.x` (position [index of wire segment] along the helix) vary from 0 to $S-1$ and the values of `hidx.y` (position around the cylinder of a wire segment) vary from 0 to $M-1$.

(a) Compute the amount of data moving between CPU and GPU (in both directions) for the code shown.

For the solution consider the arrays `helix_position`, `helix_orientation`, `helix_u`, `helix_v`, and `hidx`. Note that `helix_coord` has the same data as `helix_position`.

Let S denote the number of helix segments, this is the size of arrays `helix_position`, `helix_orientation`, `helix_u`, `helix_v`, and `helix_position`. Let W denote the number of points on the wire surface (which surrounds the helix), W will be some multiple of S. Array `hidx` has W elements. The CUDA code is launched with a total of S threads (each computing one helix segment), the vertex shader is run on a rendering pass of W vertices.

Array `helix_orientation` holds quaternions, which are four-element vectors of floats. The size of an element of `helix_orientation` is $4 \times 4 = 16$ bytes. Each element of `hidx` is a two-element integer vector, the total size is 8 bytes. The other arrays hold 3-element FP vectors, of size 12 bytes.

The data moving from GPU to CPU are the arrays `helix_position` and `helix_orientation`, the size is $S(12+16) = 28S$ bytes.

The CPU uses `helix_orientation` to compute the u and v vectors, which it sends to the GPU. The CPU also prepares and sends the `hidx` array. The position array (renamed `helix_coord`) is also sent. Notice that the orientation is not re-sent to the GPU. The total size is $(12 + 12 + 12)S + 8W = 36S + 8W$.

(b) Compute the amount of data that would be moved if u and v were computed in CUDA.

Assume that if u and v were computed in CUDA there would be no need to send the orientation to the CPU. In that case the data sent from the CPU to the GPU would be `helix_position` and the vectors: $(12 + 12 + 12)S$ bytes, and increase of $8S$ bytes. Note that the GPU would be able to compute these quantities faster than the CPU, possibly offsetting the increased communication time.

(c) Compute the amount of data that would be moved if u and v were computed in the vertex shader.

If u and v were computed in the vertex shader we would no longer need to send them from the CPU, but we would have to send the orientation. The data from CPU to GPU would be $(12 + 16)S + 8W = 28S + 8W$, a reduction of $8S$ bytes.

(d) Which option computes the same thing multiple times? Is there any benefit to this redundancy?

Because the vertex shader is run W times each value of u and v will be computed W/S times, rather than once in the other variations. There will be a benefit if execution time were limited by the time needed to send data.

Problem 4: [15 pts]A goal of Homework 2 was to use `glDrawElements` in place of `glDrawArray`. For our task of rendering a helix, `glDrawElements` reduces the number of vertices that need to be sent from CPU to GPU by a factor of 2.

(*a*) Explain why half the number of vertices are needed.

The helix constructed in Homework 2 consists of multiple triangle strips. There are vertices in one triangle strips that are in the same position as vertices in an adjacent triangle strip. When using `glDrawArrays` such vertices need to be sent twice, one for each strip. The command `glDrawElements` avoids that by using vertex indices (position in an array). (See answer to next part.) This way a vertex can be specified any number of times. For the helix this reduces by half (over a triangle strip) the number of vertices sent.

(*b*) When using `glDrawElements` what needs to be sent from CPU to GPU that was not needed with `glDrawArray`? How large is it in comparison to the vertex arrays?

With `glDrawElements` one must send an array of vertices and an array of indices. It is the array of indices which is not needed using the simpler `glDrawArray` command.

A vertex consists of a coordinate and its attributes (such as the normal, a color, etc.). A coordinate typically consists of three floats, for a size of 12 bytes, a normal is another 12 bytes. Stopping there we have 24 bytes per vertex. An index is an integer, which might be 4 bytes, one sixth the size of a vertex.

Based on the way we used `glDrawElements` in Homework 2, two indices are used per unique vertex, so the index sizes would be one third the vertex size.

Problem 5: [15 pts]Answer the following questions abut coordinate spaces. For the transformations use $T_{\vec{v}}$ to indicate a translation transformation and $R_{\vec{u}\to\vec{v}}$ indicate a rotation transform that rotates $\vec{u}$ to $\vec{v}$.

(*a*) Let P_o be a coordinate in object space coordinates, let E be the location of the eye and $\vec{v}$ be a normal to the projection surface. Show how to transform P_o into an eye-space coordinate.

In eye space the eye is at the origin and the projection surface normal points in the $-z$ direction. (The projection surface can be thought of as the computer monitor that the eye is looking at [or the plane in which the monitor's screen resides].)

First translate (move) E to the origin, that is done by $T_{\vec{E0}}$, where $\vec{E0}$ is a vector from E to the origin. Next rotate the space so the projection surface normal, use $R_{\vec{v}\to(0,0,-1)}$. The transformation matrix is the product $R_{\vec{v}\to(0,0,-1)}T_{\vec{E0}}$, note that translation is on the right (and so is performed first). The transformed coordinate is $R_{\vec{v}\to(0,0,-1)}T_{\vec{E0}}P_0$.

(*b*) Let P be a point in clip space. How can we determine if P is in the view volume?

In clip space the view volume is a cube centered on the origin, aligned with the axes, and with edges of length 2. The cube vertices are therefore at $(\pm1, \pm1, \pm1)$. Let $P = (x, y, z)$. It is in the view volume if $|x| \leq 1$, $|y| \leq 1$, and $|z| \leq 1$. If P is homogeneous there is no need to homogenize. Let $P = (x, y, z, w)$; it is in the clip space if $|x| \leq w$, $|y| \leq w$, and $|z| \leq w$.

(*c*) Suppose we are rendering to a window of size 1000×1000 pixels. Let $P_c = (0.5, 0.5, 0.5)$ be a coordinate in clip space.

☑ Provide an example of clip space coordinate $P_d \neq P_c$ that is likely to be in the same pixel as P_c.

In clip space x ranges from -1 to +1, and that spans 1000 pixels. So in clip space the distance between adjacent pixels in the x direction is $2/1000 = .002$. A coordinate likely in the same pixel can be obtained by moving less than 0.002: $P_d = (0.501, 0.5, 0.5)$.

☑ Provide an example of clip space coordinate that is in a pixel close to P_c.

Move a distance .002 in clip space: $P_d = (0.502, 0.5, 0.5)$.

Problem 6: [20 pts]Consider a multiprocessor in an NVIDIA CC 2.0 GPU, the type frequently discussed in class. Such a multiprocessor had 32 CUDA cores, two schedulers, and contexts for 48 warps. For all questions below assume a kernel is launched with one block per multiprocessor.

(*a*) Explain why a block size of one warp is guaranteed to leave half the cores unused.

Because one scheduler is dedicated to even-numbered warps and can only issue to the first 16 CUDA cores, the other scheduler is dedicated to odd-numbered warps and can only issue to the last 16 CUDA cores. If a block only has one warp it is even and so can only use half the cores.

(*b*) Explain why a block size of two warps will likely result in cores being idle most of the time.

With two warps it will be possible to issue to all of the cores. However instruction latency is high in NVIDIA GPUs, meaning that many cycles must pass from the time an instruction executes until its result is ready for another instruction. If there are many warps that's not a problem because while one warp is waiting for an instruction result other warps can issue and so keep the cores busy.

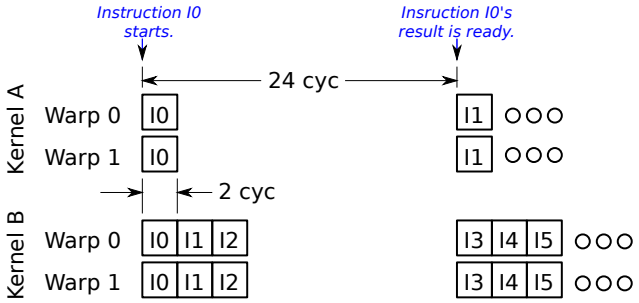
For the next two parts consider Kernel A and Kernel B, below. Notice that in Kernel A I1 must wait for I0 and I2 must wait for I1. In Kernel B I3 must wait for I0, but I1 does not have to wait for I0. Inspect the code for additional dependencies. Suppose that this pattern of dependencies continues in each kernel. That is, in Kernel A each instruction depends on its immediate predecessor, while in Kernel B each depends on the third previous instruction.

```
# Kernel A
I0:  add  r1, r2, r3
I1:  mul  r4, r1, r5
I2:  sub  r6, r4, r7
I3:  or   r8, r6, r9
...

# Kernel B
I0:  add  r1, r2, r3
I1:  add  r10, r12, r13
I2:  add  r20, r22, r23
I3:  mul  r4, r1, r5
I4:  mul  r14, r11, r15
I5:  mul  r24, r21, r25
```

(*c*) Suppose that each kernel is launched with two warps per block (and one block per MP). Assume that the instruction latency is 24 cycles. Draw a diagram showing the time that each instruction executes, until the pattern is obvious.

Diagram appears below.



(*d*) Compute the CUDA core utilization (the fraction of time a core will be used) for each kernel.

For Kernel A during the 24-cycle latency each thread can execute one instruction, for a total of 64 instruction counting all threads in a block. During 24 cycles the hardware can issue $24 \times 32 = 768$ instructions, so the utilization for A is $\frac{64}{768} = \frac{1}{12} = 8.3\%$. In Kernel B during the 24-cycle latency waiting for results from `I0` a thread can execute `I1` and `I2`. Counting all 64 threads, that's 192 instructions. The utilization for B is $\frac{192}{768} = \frac{1}{4} = 25\%$.