Name Solution

GPU Programming
EE 4702-1
Final Examination

Tuesday, 9 December 2014 7:30–9:30 CST

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

Alias Methane?

Good Luck!

Problem 1: [20 pts] Part of the vertex shader from Homework 6 (which is similar to the vertex shader in earlier assignments) appears below. For the questions below let c denote the chain length (the number of balls), let s denote the value of `opt_segments` (the number of iterations of the `t` loop in the CPU code), and let $n = cs$ denote the number of "vertices" sent to the vertex shader.

```
in ivec2 gl_Vertex; // Vertex Shader Inputs

out Data_to_GS {    // Interface block for vertex shader output / geometry shader input.
  vec4 vertex_e[2][2];  // Vertex coordinates in eye space.
  vec4 position[2][2];  // Vertex coordinates in clip space.
  vec2 texCoord[2];
  vec3 normal_e[2];
  vec3 radial_e;  // Normal for edge primitives.
};

void vs_main_lines() {
  const int bidx = gl_Vertex.x;
  const int ti = gl_Vertex.y;
  const int radial_idx = bidx * opt_segments + ti;
  const float delta_t = 1.0 / opt_segments;
  const float t = float(ti) * delta_t;
  const float theta = delta_t * radial_idx * omega;
  vec3 pos1 = balls_pos[bidx-1].xyz;                  // Buffer object access.
  vec3 pos2 = balls_pos[bidx].xyz;                    // Buffer object access.
  vec3 v12 = pos2.xyz - pos1.xyz;
  // ...
```

(a) For an entire rendering pass (execution of `glDrawArrays`) how much data is sent from the CPU to the GPU for this vertex shader? The answer should be in bytes and in terms of c, s, and n.

☑ Amount of data from CPU to GPU:

Short answer: Amount of data is $(8n + 16c)\,\mathrm{B}$.

This particular vertex shader reads data two ways: as vertex-shader inputs (which includes data specified using `glVertex` and `glColor`) and via the access to `balls_pos`.

Though in general a vertex shader can read all kinds of input, such as the values provided by `glNormal` and `glTexCoord`, this vertex shader only reads inputs `gl_Vertex.x` and `gl_Vertex.y`. (Those who took the course in the Fall 2014 semester should remember that, because the vertex shader was used as an example of how one can compute attributes such as vertex coordinates, normals, and texture coordinates given just two integers and access to an array [`balls_pos`]. Future students who are reading this solution might suspect that the code following the ellipsis accesses additional vertex inputs. That would be a reasonable assumption but this solution will be based on the Homework 6 version of the code.)

As we can see from the declaration, the `gl_Vertex` input is a two-integer vector, and so has a size of $8\,\mathrm{B}$. The vertex shader is called once per vertex, so the amount of data sent to the GPU due to the vertex shader inputs is $8n\,\mathrm{B}$.

The vertex shader also reads the array `balls_pos`, which is bound to a buffer object. Because the physics simulation updates ball positions, the buffer object needs to be updated every frame. Each element of `balls_pos` is a four-float vector, which occupies $16\,\mathrm{B}$. (We can see that the code above only uses three components, for example, by looking at `balls_pos[bidx].xyz`. However the presence of the swizzle operator (`.xyz`) implies that each element is a four-element vector.)

There is one element for each ball, so for c balls the total size of `balls_pos` (and the buffer object to which it is bound) is $16c\,\mathrm{B}$.

Counting both the vertex shader inputs and the buffer object, the total amount of data sent is $(8n + 16c)\,\mathrm{B}$.

(b) For an entire rendering pass (execution of `glDrawArrays`) how much data is output by the vertex shader for the geometry shader? The answer should be in bytes and in terms of c, s, and n.

☑ Amount of data output from vertex shader:

Based on the interface block, the amount of data written by each invocation of the vertex shader is as follows. Member `vertex_e` is a 2×2 array of 4-component vectors. Each component is a float which is 4 bytes. The total size is $2 \times 2 \times 4 \times 4\,\mathrm{B} = 64\,\mathrm{B}$. The member `position` is also $64\,\mathrm{B}$. Member `texCoord` is $16\,\mathrm{B}$, `normal_e` is $24\,\mathrm{B}$, and `radial_e` is $12\,\mathrm{B}$. The total size of the output is $180\,\mathrm{B}$. That's for one invocation of the vertex shader. The vertex shader is run n times so the total size of the output data is $\boxed{180n\,\mathrm{B}}$.

(c) For an entire rendering pass (execution of `glDrawArrays`) how much data is read by the geometry shader (not shown) per frame? It's important to remember that the input to the vertex shader is line strips and the input to the geometry shader is lines. The answer should be in bytes and in terms of c, s, and n.

☑ Amount of data input to geometry shader:

The vertex shader is run once for each vertex sent (or specified) by the CPU. The number of times the geometry shader is run depends on the input primitives. The CPU sends vertices as line strips. The geometry shader reads them as lines. To read a line the geometry shader needs to read two primitives, for a size of $360\,\mathrm{B}$. The geometry shader will be run once per line. Because line strips are being used the geometry shader is run $n - 1$ times. (If the CPU had send the data as individual lines the geometry shader would have been run $n/2$ times.) The total amount of data read by the geometry shader is therefore $\boxed{360(n-1)\,\mathrm{B}}$.

Problem 1, continued: Recall that `balls_pos` is an array of ball positions, something that changes each frame. Note that the vertex shader is reading the positions from a buffer object.

It would be possible, though wasteful, to use a vertex shader input for the ball positions.

(*d*) Show the vertex shader input declarations that can be used for ball positions. *Hint: This part is easy, look at the code on the previous page.*

☑ Declarations:

The solution appears below. Two `vec4`'s are declared, corresponding to the two ball positions read by the vertex shader. This looks harmless, until you compare the amount of data that would have to be sent for these inputs in comparison to the amount of data when using buffer objects. See the next subproblem.

```
// SOLUTION
in ivec2 gl_Vertex; // Vertex Shader Inputs
in vec4 ball_pos_i;
in vec4 ball_pos_im1;
```

(*e*) Compute the amount of data sent from the CPU to the GPU needed when the ball positions are read as vertex shader inputs. The answer should be in bytes and in terms of c, s, and n.

☑ Amount of data for ball positions as vertex shader inputs:

The two ball positions together consume 32 bytes per vertex shader invocation, and so the total amount of data is $\boxed{32n\,\mathrm{B}}$. Recalling that $n = cs$, we can see that this is much larger than the amount of data sent when buffer objects are used. That's because when vertex inputs are used the position of each ball must be sent from the CPU to the GPU $2s$ times, with buffer objects it is sent just once.

Problem 2: [15 pts] The fragment shader used in many class demonstrations and homework assignments performs lighting calculations. This yields the best results, but is computationally expensive. The historic location for computing lighting is the vertex shader.

(*a*) With the help of a diagram describe how fragment-shader lighting computation yields a more realistic image than vertex-shader lighting computation.

☑ Fragment shader lighting more realistic because:

Because the lighted color of a fragment is based upon the position of the light in relation to that fragment. In contrast, when the vertex shader is used to compute lighting the lighting is computed for the primitive's vertices and these values are interpolated to get the lighted color of a fragment. Suppose the light were close to the center of a triangle. We would expect there to be a brightly-lit pool near the light and the vertices to be darker. We get just that with fragment-shader computed lighting. With vertex shader lighting the triangle could have uniform coloring (if the light were equidistant from the vertices).

(*b*) How many times more work does it take to compute lighting in the fragment shader? **State any assumptions** used for your answer.

☑ Using the fragment shader for lighting requires $x\times$ more computation assuming:

Assuming that there are $3x$ fragments. Regardless of the projected size we would compute lighting 3 times if using the vertex shader on a triangle, so if the fragment shader requires $x\times$ more computation it must have $3x$ fragments.

(*c*) Doing lighting calculations in the fragment shader does not always make a big difference in appearance. Consider a hybrid scheme in which a fragment-shader input determines whether the fragment shader should do the lighting calculations or just use the `color` input. Which stage is best suited to determine if the fragment shader should do lighting calculations?

☑ Shader stage best suited to determine where to compute lighting?

☑ Explain.

The geometry shader, because it has access to all the vertices in a primitive and so has the information it needs to determine if the light is much closer to the interior of a primitive than to any vertex. (The geometry shader, and all programmable stages have access to light source coordinates and other information.) The vertex shader can't do this because it can only see one vertex. The fragment shader does not normally have access to all the vertices, but even if it did it would not be suitable because just determining whether or not to perform lighting calculations would use a substantial amount of computing power (because the fragment shader is often executed many more times than the vertex shader, as those who read the solution to the previous subproblem know).

Problem 3: [20 pts] The CUDA kernel below computes an operation on all pairs of elements of array `d_pos`, the number of array elements is `chain_length`. Let B denote the block size of the kernel launch. The grid size (number of blocks) is (and must be) `chain_length`.

```
__global__ void force_pairs()
{
  const int a_idx = blockIdx.x;
  const float4 pos_a = d_pos[a_idx];                    // Global Access A

  float3 force = make_float3(0,0,0);

  for ( int b_idx = threadIdx.x;  b_idx < chain_length;  b_idx += blockDim.x)
    {
      const float4 pos_b = d_pos[b_idx];
      force += force_compute(pos_a,pos_b);
    }

  d_vel[a_idx] += delta_t * d_balls[a_idx].mass_inv * force; // Error here.
}
```

(*a*) Compute the amount of data read for the entire kernel due to the line Global Access A. The size of a `float4` is 16 bytes and the minimum request size is 32 bytes. Compute this number in terms of B (block size) and c (`chain_length`). *Note: In the original exam n was used for chain length.*

☑ Amount of data read due to Access A.

Short answer: $\boxed{\frac{32cB}{32}\,\mathrm{B} = cB\,\mathrm{B}}$. (Don't get confused between B which is the block size and B which is the unit *byte*.)

The total number of threads is the product of the block size and the number of blocks, Bc in this case. Global Access A accesses element at index `a_idx`, which is set to `blockIdx.x`. Recall that all threads in a block share the same value of `blockIdx.x` and so only c different elements of `d_pos` are being read. However, the code above can access the same element multiple times. In class we assumed that requests are formed based on the combined accesses of all threads in a warp. For the code above all threads in a warp are accessing the same element, so a single request is formed for a warp. The size of that request is 32 bytes, the minimum size request (which is wasteful for us since we need only 16 bytes of data). To determine the total amount of data accessed we need to compute the number of warps. That's the number of threads divided by 32, or $cB/32$ in our case. (We will conveniently assume that B is a multiple of 32.) That gives us the total data size of $\frac{32cB}{32}\,\mathrm{B} = cB\,\mathrm{B}$.

(*b*) Modify the code so that it uses shared memory to reduce the amount of data read due to Global Access A.

☑ Modify to use shared memory to reduce data read for A.

The modified code appears below. Variable `pos_a` is now declared shared. Only `threadIdx.x` 0 initializes it, so there is just one global access per block. The `__syncthreads` prevents the other threads from accessing `pos_a` until thread 0 writes it.

```
__global__ void force_pairs() {
  const int a_idx = blockIdx.x;
  // SOLUTION below
  __shared__ float4 pos_a;  // Declare pos_a shared.
  if ( threadIdx.x == 0 ) pos_a = d_pos[a_idx]; // Just 1 thread writes it.
  __syncthreads();   // All threads must wait until thread 0 writes it.
  // SOLUTION above.

  float3 force = make_float3(0,0,0);
```

Problem 3, continued: The CUDA code below is the same as the previous page. The line marked "Error here" has two problems.

```
__global__ void force_pairs()
{
  const int a_idx = blockIdx.x;
  const float4 pos_a = d_pos[a_idx];                    // Global Access A

  float3 force = make_float3(0,0,0);

  for ( int b_idx = threadIdx.x;  b_idx < chain_length;  b_idx += blockDim.x)
    {
      const float4 pos_b = d_pos[b_idx];
      force += force_compute(pos_a,pos_b);
    }

  d_vel[a_idx] += delta_t * d_balls[a_idx].mass_inv * force; // Error here.
}
```

(*c*) Due to one problem the wrong value will be written into `d_vel`. Explain why.

☑ Wrong value written to `d_vel` because ....

The `+=` operator reads memory, adds something to the value, then writes the sum. The problem is that multiple threads can be executing that statement at the same time, for example, all the threads in a warp. Suppose we wanted each thread in a warp to add 1 and that the original value were 10, we would want the final value to be equal to 42 (the original value plus 32). But, if each thread reads a 10, adds one to it, then writes the result, we'd have 32 threads writing 11. So we'd get 11 instead of 42.

(*d*) The line marked "Error here" is also inefficient. Explain why and fix the problem. *Hint: Use shared memory.*

☑ Reason for inefficiency.

☑ Fix with shared memory.

Global memory access is slow, especially when many threads are accessing the same location.

Notice that all threads in a block are accessing the same elements of `d_vel` and `d_balls`. Also notice that the same result would be obtained if we added up the values of `force` for each thread in a block, and used that value to update `d_vel`. The code below does just that. A shared array `forces` is used to hold the force computed by each thread. A block-wide sum of forces is computed in two steps. In the first step, performed by the first warp in a block, the sum of forces in each lane is computed. (The lane is `threadIdx.x` modulo the warp size, 32.) In the second step, performed by just one thread, the force sums for each lane are added together yielding a block-wide sum. That is used to update `d_vel`.

```
// SOLUTION below
__shared__ float3 forces[1024];  // Array for sharing forces.
forces[threadIdx.x] = force;     // Write thread's computed force.
__syncthreads();
if ( threadIdx.x >= 32 ) return;

// Compute force sum for each lane.
for ( int i=threadIdx.x+32; i<blockDim.x; i+=32 )
  force[threadIdx.x] += forces[i];

if ( threadIdx.x > 0 ) return;
```

```
// Compute force sum for entire block.
for ( int i=0; i<32; i++ ) forces[0] += forces[i];

d_vel[a_idx] += delta_t * d_balls[a_idx].mass_inv * forces[0];
```

Problem 4: Answer each question below.

(*a*) [5 pts] A CUDA application launches a kernel with a grid size of 20 blocks. Describe a situation in which that is inefficient and explain why.

☑ Twenty blocks is inefficient when:

The number of multiprocessors does not evenly divide 20. For example, if the number of multiprocessors were 3, 6, 7, 9, 11 to 19, and 21 or higher. Twenty blocks can be efficient when the number of multiprocessors is 1, 2, 4, 5, 10, or 20.

☑ Because:

Because the work cannot be divided evenly and so some multiprocessors would finish early. If our goal were minimum computation time we'd want all the multiprocessors to be working until the computation finished.

(*b*) [5 pts] A CUDA application launches a kernel with a block size of 19 threads. Why is that inefficient?

☑ Nineteen threads is inefficient because:

Lots of reasons. First, there are not enough threads to use the common functional units on Kepler GPUs. (For example, the 192 single-precision FP units.) Even if there were just 19 functional units of the type needed by the code, additional threads are needed to hide latency. The general rule given in class was 256 threads per multiprocessor.

Problem 4, continued:

(*c*) [5 pts] The loop body in the CUDA code below does one of five kinds of operations. Which is the biggest concern for the code below, grid size, block size, or warp size?

```
const int tid = threadIdx.x + blockIdx.x * blockDim.x;
for ( int i = tid; i<size; i += num_threads ) {
  int k = kind[i];
  float4 d = data[i];
  switch ( k ) {
    case 0: operation_x(i,d); break;
    case 1: operation_y(i,d); break;
    case 2: operation_z(i,d); break;
    case 3: operation_a(i,d); break;
    case 4: operation_b(i,d); break;
  }
}
```

☑ Biggest concern for code is grid, block, or [warp]. (Circle one.)

☑ Explain:

The warp size is the biggest concern because one instruction is fetched for all threads in a warp, though only the threads that need the instruction will execute it. If `k` had the same value for all threads in a warp, there would be no problem, every thread would need whatever instruction was fetched. But if some threads in a warp retrieved a value of 0 for `k` and others a value of 1, then execution might first follow the path down `case 0`, with the threads getting `k=1` not executing those instructions. Later the path down `case 1` would be followed with the `k=0` threads ignoring instructions. With five cases, execution time could be five times longer. The smaller the warp size the smaller this problem. For example, if the warp size were just two then at worst execution time would be twice as long. And perhaps it would be easier to cluster `kind` values so that they would be the same for the two members of these hypothetical 2-thread warps.

(*d*) [5 pts] For many classroom CUDA examples a structure is used to hold application related variables, such as the one below. For the GPU, this had been declared in the `__constant__` address space, but it could have been declared in the global (device) address space (shown commented out below). What's the disadvantage of the global space for this?

```
struct App {
  int num_threads, array_size;
  Vector *v_in;
  Vector *d_v_in;
};

App app;  // In host address space.
__constant__ App d_app;  // In device constant address space.
// __device__ App d_app;  // In device global address space.
```

☑ Disadvantage of global address space for the `App` structure?

The global address space is much slower than the constant address space. (The constant address space is much smaller than the global address space and it can't be written by the GPU, but that's not an issue for the `App` structure.)

Problem 5: Answer each question below.

(*a*) [5 pts] The vertex shader below uses the deprecated output `gl_FrontColor` to hold a lighted color. Those setting the OpenGL standard would prefer that you define your own vertex shader output to hold the lighted color. On the other hand, `gl_Position` is not deprecated and it must be written (if not by the vertex shader, then by the geometry shader). Why?

```
void vs_main() {
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

  vec4 vertex_e = gl_ModelViewMatrix * gl_Vertex;
  vec3 normal_e = normalize(gl_NormalMatrix * gl_Normal);
  gl_FrontColor = generic_lighting(vertex_e,gl_Color,normal_e);
}
```

☑ Cannot substitute `gl_Position` with a user-defined output because:

The value of `gl_Position` is needed by the fixed functionality for clipping and rasterizing. In contrast the fixed functionality never uses the lighted color (except for the deprecated compatibility profile) so there is no need for OpenGL to define a name for it.

(*b*) [5 pts] It's possible to define user-defined outputs for the vertex and geometry shaders, but not for the fragment or compute shaders.

☑ Why can't there be user-defined fragment shader outputs?

A user-defined output of one stage can only be recognized if it's a user-defined input of a subsequent stage. (For example, if you define a vertex shader output named `yadayada`, you need to define a geometry shader input named `yadayada` to use the result.) The fragment shader is the last programmable stage, so there's nothing left to recognize a user-defined output.

☑ Why can't there be user-defined compute shader outputs?

The vertex, geometry, and fragment shaders are part of the rendering pipeline. The term *pipeline* indicates that data flows through these shaders in order, with the output of one stage being sent to the input of another (with possible re-grouping or rasterizing). The compute shader is not part of the rendering pipeline, so it does not have or need inputs and outputs.

(*c*) [5 pts] In the OpenGL shader code below `b` is assigned the value of `a`, but with the elements reversed. It does so the hard way. Do it the easy way (using what we called swizzle operations).

```
vec4 a, b;
b.x = a.w;  b.y = a.z;  b.z = a.y;  b.w = a.x;
```

☑ Assign `b` the easy way.

The easy way is `b=a.wzyx`.

(*d*) [5 pts] The fragment shader input's `smooth` interpolation qualifier (which is the default for floating-point values) indicates that the qualified variable (`color` in the example below) should be linearly interpolated based on object-space coordinates. Linear interpolation is easy to do, so why is `smooth` interpolation expensive?

```
in Data_GF {
  smooth vec4 color;
}
```

☑ `smooth` interpolation is expensive because ...

Because we are working with window-space coordinates, and the interpolation is not linear in window space.

(*e*) [5 pts] The statement below is correct but does not provide the full justification for GPUs. Complete the statement.

> Executing 3D graphics is computationally intensive and so many computers have GPUs to take this computation load off the CPU.

☑ To justify GPUs the statement should go on to say ....

... a significant amount of CPU hardware is not needed for typical 3D graphics computations, so a GPU can be built with more of the hardware that it does need, such as floating-point computation units. (The unneeded hardware includes structures that allow instructions to execute out of order and large caches.)