Name Solution

EE 4702

Take-Home Pre-Final Questions

Tuesday, 4 December 2012

These are practice questions for the final exam.

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

Alias 0x5e1f

Good Luck!

Problem 1: [20 pts]The advantage of an OpenGL triangle strip over individual triangles is that vertices shared by multiple triangles only need to be specified once.

(*a*) Why is the code below not a good example?

```
for ( int i=0; i<1000; i++ )
  {
    glBegin(GL_TRIANGLE_STRIP);
    glNormal3fv(normal[i]);  glVertex3fv(v0[i]);
    glNormal3fv(normal[i]);  glVertex3fv(v1[i]);
    glNormal3fv(normal[i]);  glVertex3fv(v2[i]);
    glEnd();
  }
```

The advantage of a triangle strip is that the work performed by the vertex shader on one vertex can be used for up to three triangles (the triangles that share the vertex). Since the code above only renders one triangle the benefit is not realized. That is, the vertex shader runs three times, once for each vertex. That's the same number of times as if individual triangles were used.

(*b*) Explain the impact of the proper use of triangle strips on each of the items below. For one of the items below the answer should consider two cases, when a buffer object is being used and when it is not being used. Remember, for this part assume that triangle strips are being used properly.

- Total communication bandwidth.

Case 1: Without using a buffer object for vertex data (vertices, normals, etc.). A triangle strip reduces CPU/GPU communication by a factor of 3 for each rendering pass. We can assume there are many rendering passes so the factor of 3 should have a bit impact.

Case 2: Using a buffer object for vertex data. Since a buffer object is used the vertex data will be sent from the CPU to the GPU only once. The communication bandwidth will be reduced by a factor of 3 when it is sent but the overall impact should be much smaller since we are sending the vertices just once rather than once for each rendering pass.

- Total vertex shader execution time.

Reduced by a factor of 3.

- Total fragment shader execution time.

There is no difference between triangle strips and individual triangles in stages after the vertex shader, which includes the fragment shader. So the impact on the fragment shader is none.

(*c*) A triangle strip is well suited for situations in which triangles are approximating a smooth surface (such as the surface of a sphere). It is less well suited for surfaces that really are composed of triangles. Explain why. *Hint: Consider colors and normals.*

Consider a situation in which one would like to render triangles that form a strip but are not on the same plane. In this case the geometric normal for each triangle is different, and we would like to use this normal. However, a normal is specified for a vertex, so all triangles sharing the vertex must get the same normal.

Problem 2: [20 pts]Assume that the lighted color of a triangle depends on conditions at its centroid (a point equidistant from its three vertices), including light and viewer location. Though computed for the center, this same lighted color would be used for all fragments.

(*a*) Which would be the most appropriate shader to compute this color: vertex, geometry, or fragment? Explain.

Geometry. The shader is called once per triangle and has access to all vertices. It will be able to compute the centroid, compute the lighted color, and use that color for each vertex.

(*b*) Regardless of your answer above, explain how a vertex shader could be used to compute the centroid (a first step in computing the color). This will require some extra help from the CPU, but the burden of computation should be on the shader. (*Note: the challenge here is getting the vertex locations.*)

For the vertex shader to compute the centroid it needs to know the other two vertices. The CPU would have to send along that information, by defining additional attributes for the vertex. (Or by borrowing unused attributes such as texture coordinates for texture units that are not in use.) The CPU could also compute the centroid and send that information, but that would not be shifting the burden of computation to the shader.

(*c*) Shown below are three possible declarations for the output of the shader that computes the triangle's lighted color. Each of these will produce the desired result. But one is clearly best, and one is clearly worst. Identify the best and worst to use and explain why for each.

```
flat out vec3 tri_color;
smooth out vec3 tri_color;
noperspective out vec3 tri_color;
```

Since the color is the same over all fragments the lighted color computed by the vertex shader should be delivered unchanged to each fragment shader invocation. That is what the `flat` declaration does. The `noperspective` performs a linear interpolation, and `smooth` performs a perspective-correct interpolation. The `noperspective` is by far the most computationally expensive so it is the worst to use.

Problem 3: [20 pts]Suppose an NVIDIA GPU of compute capability 2.0 has 8 multiprocessors. Each multiprocessor has 32 cores, but a single warp can use only 16 cores (it takes two different warps to keep all 32 cores in a multiprocessor busy).

The GPU is to be used to add two arrays element-wise. Suppose that the number of array elements is 2^{24}.

Let t_1 denote the amount of time it takes one thread (yes, just one) to perform the entire calculation on the GPU. The kernel code is shown below:

```
__device__ void prob(int array_size) {
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  for ( int i=tid; i<array_size; i += num_threads )
    result[i] = a[i] + b[i];
}
```

(*a*) Compute the amount of time it would take to perform the computation for each of the following launch configurations, and explain why they are the same or different (whichever the case applies):

☑ One block of 16 threads.

Execution time is $t_1/16$.

☑ Two blocks of 8 threads each.

Execution time is $t_1/16$.

☑ Explain why same or different.

The total number of threads is 16 in both cases. The one-block configuration would run on one multiprocessor and would be able to use 16 cores, the two-block configuration would run on two multiprocessors and would be able to make use of 8 cores each, for a total of 16.

Since there are enough cores for all threads none of the threads will have to wait for a core (that is occupied by another thread), and so the rate of computation for each thread will be the same as the rate of computation in the one-thread case. Since work is divided evenly between threads, the execution time in both cases will be $t_1/16$.

(*b*) Estimate (an exact answer isn't possible) the amount of time it would take to perform the computation for each of the following launch configurations and explain why they are the same or different (whichever the case applies):

☑ One block of 1024 threads.

Lets assume that 256 threads are enough to keep all of the cores in a multiprocessor busy. Then there would be no difference in performance in configurations from 256 to 1024 threads per block. Then the time is $t_1/256$.

☑ Two blocks of 512 threads each.

Again lets assume that 256 threads are sufficient to keep all cores in a multiprocessor busy. However, since there are two blocks two multiprocessors will be used and so the time will be based on 512 threads, so the execution time is $t_1/512$.

☑ Explain why same or different.

The times are different because with just one block the threads all must run on one multiprocessor, with two blocks they can run on two and so have access to the same number of CUDA cores.

(*c*) Explain how higher memory latency would increase the number of threads needed for good performance.

Memory latency is the amount of time needed to retrieve data from global memory (assuming a cache miss). A thread that need data loaded from global memory must wait (and do nothing) until it arrives. While one thread waits another can execute, so having more threads means that more threads can fill this role (executing while another thread is waiting). The longer the latency the more threads that are needed.

Problem 4: [20 pts]Consider the CUDA code below in which two alternatives are shown: putting a balls array in shared memory or in global memory. Assume the code runs on GPUs of compute capability 2.0 (though the answers could apply to 1.0 through 2.5).

(*a*) Explain why the Option 1 (shared memory) code below will run slowly. Fix the problem making as few changes as possible.

```
struct Ball {
  float3 position;   float3 velocity;
  float radius;      float mass;      };

__shared__ Ball balls[256]; // Option 1 - Shared memory.

__device__ void
update(float deltat)
{
  int start = threadIdx.x * balls_per_thread;
  int stop = start + balls_per_thread;
  for ( int i=start; i<stop; i++ )
    balls[i].position += deltat * balls[i].velocity;
}
```

The code suffers from shared memory bank conflicts.

(*b*) Explain why the Option 2 code below (using global memory) will run slowly. Fix the problems making as few changes as possible. (There are two major things to be fixed.)

```
struct Ball {
  float3 position;   float3 velocity;
  float radius;      float mass;      };

Ball* balls; // Option 2 - Global memory.

__device__ void
update(float deltat)
{
  int start = ( threadIdx.x + blockIdx.x * blockDim.x ) * balls_per_thread;
  int stop = start + balls_per_thread;
  for ( int i=start; i<stop; i++ )
    balls[i].position += deltat * balls[i].velocity;
}
```

Memory access is most efficient when consecutive threads access consecutive elements and when the size of the values accessed are a power of 2. In the code above consecutive threads will access memory addresses that differ by a factor of `balls_per_thread * sizeof(Ball)`, wasting memory requests. Two things are done to fix that. First, the array index will be computed so that it is equal to `threadIdx.x + something`, where `something` is the same for all threads in a block.

Second, the data will be reorganized from an array of structures, to just two arrays, `b_pos` and `b_vel`. (The code for initializing these arrays is not shown). This way an access to `b_pos[0]`, `b_pos[1]`, . . ., `b_pos[31]` can be satisfied by fetching a contiguous block of memory.

Another change was using `float4` rather than `float3` for position and velocity. This ensures that access to the vectors is done by one instruction rather than three.

```
// SOLUTION
float4* b_pos, b_vel;

__device__ void update(float deltat)
{
  const int num_threads = blockDim.x * gridDim.x;
  const int start = threadIdx.x + blockIdx.x * blockDim.x;
  for ( int i=0; i<balls_per_thread; i++ )  {
      const int idx = start + i * num_threads;
      b_pos[idx] += deltat * b_vel[idx];
    }
}
```

Problem 5: [20 pts]Answer each question below.

(*a*) Why is the if/return needed in the CUDA code below?

```
__device__ void prob(int array_size) {
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if ( tid >= array_size ) return;
  result[tid] = a[tid] + b[tid];
}
```

The number of threads must be a multiple of the block size. We can assume that `array_size` can be any positive integer.

(*b*) The line of code below checks for a special case to avoid a square root. That might make sense for a CPU, but not for necessarily for CUDA code on a GPU. Describe a situation in which it makes sense for CUDA and a different situation when it makes no sense (meaning it would be faster to do the square root all the time). Assume that 50% of the time d is equal to 1.

```
  if ( d == 1 ) s = 1; else s = sqrt(d);
```

It makes sense if there is no branch divergence, meaning that for all warps the branch is either always taken or always not taken.

If there is branch divergence then the code will run more slowly than code always performing the square root since within a warp the execution time will be the sum of both paths through the if.

(*c*) The loop below is innocent on a CPU, but on a GPU it can execute inefficiently. Identify the problem and fix it.

```
  for ( int i=0; i<32; i++ )
    {
      if ( a[i] == t ) { do_stuff(i); break; }
    }
```

Unless the compiler is really clever, threads within a warp computing `do_stuff` will not only compute it in parallel with other threads when the value of `i` matches. The solution is to move `do_stuff` outside of the loop. See the code below.

```
  int dsi = -1;
  for ( int i=0; i<32; i++ )
    {
      if ( a[i] == t ) { dsi = i; break; }
    }
  if ( dsi >= 0 ) do_stuff(dsi);
```