Name Solution

GPU Programming
EE 4702-1
Final Examination

12 December 2013, 15:00–17:00 CST

Problem 1		(25 pts)
Problem 2		(25 pts)
Problem 3		(15 pts)
Problem 4		(10 pts)
Problem 5		(15 pts)
Problem 6		(10 pts)
Exam Total		(100 pts)

Alias Click Here

Good Luck!

Problem 1: (25 pts) The code below computes forces between all pairs of a set of balls, similar to the code from the pre-final and Homework 3. Unlike those prior versions, the code below is written for the case where the number of balls (`chain_length`) is a multiple of the total number of threads.

Let c denote the number of balls, n denote the total number of threads (`num_threads`), and g denote the grid size (number of blocks). The size of a `float4` is 16 bytes. The code runs on an NVIDIA Kepler GPU, which supports requests of size 32, 64, and 128 bytes.

```
const int tid = threadIdx.x + blockIdx.x * blockDim.x;
const int num_threads = blockDim.x * gridDim.x;
const int a_idx_start = tid;
const int balls_per_thread = dc.chain_length / num_threads;
for ( int i=0; i<balls_per_thread; i++ )
  {
    const int a_idx = a_idx_start + i * num_threads;
    const float4 pos_a = dc.d_pos[a_idx];                 // Global Access A
    float3 force = make_float3(0,0,0);
    for ( int b_idx = 0;  b_idx < dc.chain_length;  b_idx++ )
      {
        const float4 pos_b = dc.d_pos[b_idx];             // Global Access B
        force += force_compute(pos_a,pos_b);
      }
    dc.d_vel[a_idx] += delta_t * dc.d_balls[a_idx].mass_inv * force;
  }
```

(*a*) In terms of n, c, and g compute the amount of data requested and used for the line marked Global Access A.

✓ Data requested by Global Access A. ✓ Data used by A.

Important thing to note: Each value of `a_idx` is used exactly once. That is, no two threads will have the same value of `a_idx_start` and the same is true for `a_idx` because it is `a_idx_start` plus a multiple of the number of threads.

Short answer: Efficient access, so amount of data requested and used is $16c$ bytes.

Long answer: First consider the first `i` iteration, where `i=0`. For that first iteration `a_idx = tid`, which means we obviously have consecutive threads accessing consecutive elements. Each element is 16 bytes, so for four 128-bytes are needed for each warp, and all 128-bytes are used. This of course, is the most efficient type of access.

It should also be obvious that together all n threads are accessing elements 0 to $n-1$, and that each of these elements is read by exactly one thread. Because `a_idx` is `threadIdx.x` plus something, (where something is `blockIdx.x*blockDim.x+i*num_threads`), we know that consecutive threads will be accessing consecutive elements in all `i` iterations. In the second `i` iteration we'll be accessing elements n to $2n-1$, so again no element has been accessed more than once. This holds for all iterations, so in total $\frac{c}{n}n = c$ elements have been accessed, each just once. The size of an element is 16 bytes, so the total data requested is $\boxed{16c \text{ bytes}}$.

(*b*) In terms of n, c, and g compute the amount of data requested and used for the line marked Global Access B.

✓ Data requested by Global Access B. ✓ Data used by B.

Important thing to notice: Every thread in a warp uses the same value of `b_idx`. A multiprocessor will generate a single memory request for `dc.d_pos[b_idx]` on behalf of all of the threads.

Each `b_idx` iteration requests 32 bytes per warp (the minimum request size), but uses only 16. There are a total of $\frac{n}{32}$ warps so the total request is n bytes per `b_idx` iteration. There are c `b_idx` iterations and $\frac{c}{n}$ `i` iterations, for a total request of $\boxed{\frac{c}{n}cn = c^2 \text{ bytes requested}}$ and $\boxed{\frac{1}{2}c^2 \text{ bytes used}}$.

Problem 2: (25 pts) The code below is similar to the code from the previous problem except that shared memory is used to hold values needed for `pos_b`. The value of `csize` is not well chosen given the way the code is written. As before let c denote the number of balls, n denote the total number of threads, and g denote the grid size (number of blocks). The size of a `float4` is 16 bytes. The code runs on an NVIDIA Kepler GPU, which supports requests of size 32, 64, and 128 bytes.

```
int num_threads = blockDim.x * gridDim.x,  tid = threadIdx.x + blockIdx.x * blockDim.x,
int a_idx_start = tid,                     balls_per_thread = dc.chain_length / num_threads;
const int csize = 8;
__shared__ float4 c_pos[csize];
for ( int i=0; i<balls_per_thread; i++ ) {
    const int a_idx = a_idx_start + i * num_threads;
    const float4 pos_a = dc.d_pos[a_idx];                  // Global Access A
    float3 force = make_float3(0,0,0);
    for ( int b_idx = 0;  b_idx < dc.chain_length;  b_idx++ ) {
        const int c_idx = b_idx % csize;
        if ( c_idx == 0 ) {
            const int loc = threadIdx.x % csize;

            __syncthreads();

            c_pos[loc] = dc.d_pos[b_idx + loc];            // Global Access B

           //  In table:  SH = loc;   GB = b_idx + loc;

            __syncthreads();

          }
        const float4 pos_b = c_pos[c_idx];                // SA = c_idx;
        force += force_compute(pos_a,pos_b);
      }
    dc.d_vel[a_idx] += delta_t * dc.d_balls[a_idx].mass_inv * force;
  }
```

(*a*) In terms of n, c, and g compute the amount of data requested and used for the line marked Global Access B.

[✓] Data requested by Global Access B. [✓] Data used by B.

Relatively short answer: The line executes every 8 `b_idx` iterations, when it does 8 consecutive locations are read (these same 8 locations repeat). When they do execute each warp reads 8 distinct locations or $8 \times 16\,\mathrm{B} = 128\,\mathrm{B}$. The total number of executions per thread is $\frac{c}{n} c \frac{1}{8} = \frac{c^2}{8n}$. The total number of warps is $\frac{n}{32}$. The total number of executions considering all warps is $\frac{n}{32}\frac{c^2}{8n} = \frac{c^2}{256}$. As stated above, each execution of Global Access B by a warp accesses $128\,\mathrm{B}$ and so the total data accessed is $\frac{c^2}{256} 128\,\mathrm{B} = \frac{c^2}{2}\,\mathrm{B}$. Note that the method used to compute data access changes when `csize` is 32 or larger.

Here's what the code is doing. Instead of performing Global Access B *every* iteration the code does the access once every 8 (or `csize`) iterations and the loaded values are put in shared memory for use in the current iteration and the next seven iterations. The table below shows what is being accessed by threads of `tid` from 0 to 9. Each row shows information on one thread. The numbers at the top of the columns show the values of `b_idx`. The numbers in the GB column show which global locations are being accessed (look for GB in the code above), and the SH column shows to which shared memory location they are being written. The SA column shows which shared memory location is being read. Notice that under the GB column access is consecutive with `tid`, which is good, but that the same array index repeats (starting in the row for `tid` 8).

b_idx ->		0	0	1	2	3	4	5	6	7	8	8	9
i	tid	SH<-GB	SA	SA	SA	SA	SA	SA	SA	SA	SH<-GB	SA	SA
0	0	0 <- 0	0	1	2	3	4	5	6	7	0 <- 8	0	1
0	1	1 <- 1	0	1	2	3	4	5	6	7	1 <- 9	0	1
0	2	2 <- 2	0	1	2	3	4	5	6	7	2 <-10	0	1
0	3	3 <- 3	0	1	2	3	4	5	6	7	3 <-11	0	1
0	4	4 <- 4	0	1	2	3	4	5	6	7	4 <-12	0	1
0	5	5 <- 5	0	1	2	3	4	5	6	7	5 <-13	0	1
0	6	6 <- 6	0	1	2	3	4	5	6	7	6 <-14	0	1
0	7	7 <- 7	0	1	2	3	4	5	6	7	7 <-15	0	1
0	8	0 <- 0	0	1	2	3	4	5	6	7	0 <- 8	0	1
0	9	1 <- 1	0	1	2	3	4	5	6	7	1 <- 9	0	1

(*b*) Choose `csize` to minimize memory access by Global Access B.

[✓] Show new value of `csize`. [✓] Explain.

Short answer: We want to avoid multiple threads accessing the same value of `d_pos`, we can do that by setting `csize` to the block size. If we do that values for GB won't repeat in the table above on a particular block. The amount of data accessed per `i` iteration will be $16gc\,\mathrm{B}$ and in total $\frac{c}{n} 16gc\,\mathrm{B} = 16\frac{gc^2}{n}\,\mathrm{B}$.

Let s denote the value of `csize`. The amount of data accessed for $s \geq 32$ can be derived as follows. Considering all threads, each time Global Access B executes the amount of data read is $n16\,\mathrm{B}$ (even though only $16s\,\mathrm{B}$ worth of distinct data is read). The number of times it executes is $\frac{c}{n}\frac{c}{s} = \frac{c^2}{ns}$. The amount of data read is $\frac{c^2}{ns} n16\,\mathrm{B} = 16\frac{c^2}{s}\,\mathrm{B}$ for valid values of s. The largest we can make s is the block size, n/g, for that value the amount of data read is $16\frac{c^2}{n/g}\,\mathrm{B} = 16\frac{gc^2}{n}\,\mathrm{B}$. Re-writing this in terms of the block size $b = n/g$ we get $16\frac{n/bc^2}{n}\,\mathrm{B} = 16\frac{c^2}{b}\,\mathrm{B}$.

(*c*) Modify the code so that the amount of data accessed due to Global Access B matches the previous part when `csize` is set to the original value, 8. The change should be within the `if ( c_idx==0 )` block and it is only one line of code.

[✓] Modify code to reduce waste when `csize=8`.

Looking the first GB column in the visible part of the table from the solution above we see that two threads access element 0, and both write it into the same location in shared memory. That's wasteful, only one thread needs to do that. The solution is to only allow the first `csize` threads to load shared memory, but it's important that all threads call `__syncthreads`:

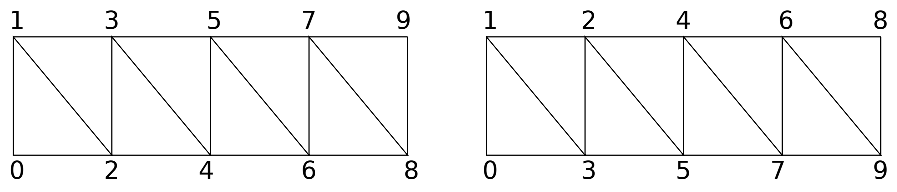
```
for ( int b_idx = 0;  b_idx < dc.chain_length;  b_idx++ ) {
    const int c_idx = b_idx % csize;
    if ( c_idx == 0 ) {
        const int loc = threadIdx.x;
        __syncthreads();
        if ( threadIdx.x < csize )    // SOLUTION: This line inserted.
           c_pos[loc] = dc.d_pos[b_idx + loc];        // Global Access B
```

Problem 3: (15 pts) Answer the following questions about triangles.

(*a*) The diagrams below show two possible numberings for vertices for rendering a rectangle as a triangle strip.

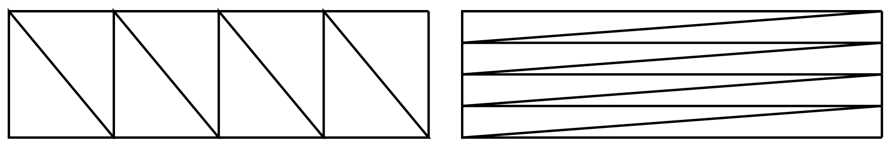
☑ Which one is correct?

☑ Show how the triangles would be rendered for the incorrect case.



The figure on the left is correct. If the numbering on the right were used the diagonals would run from lower-left to upper-right and the line between 0 and 3 would be missing.

(*b*) Several possible strategies are being considered to render a rectangle.



Suppose the left figure above were rendered as a triangle strip (with the vertices correctly numbered).

☑ How many times would the vertex shader be executed?

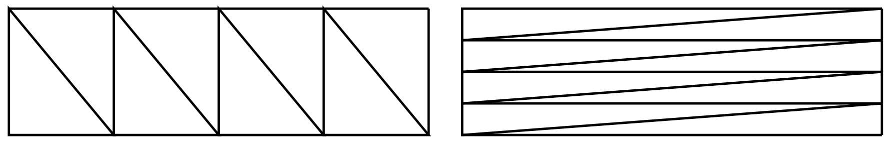
Ten times, once per vertex.

Suppose the left figure above was rendered as individual triangles.

☑ How many times would the vertex shader be executed?

There are eight triangles and so 24 vertices. So the shader would be run 24 times.

(*c*) Consider the following two ways of tessellating a rectangle.



Suppose that lighting is computed in the vertex shader.

[✓] Which tessellation above is better? [✓] Explain.

The tessellation on the right is worse because vertices of each triangle are far away from each other. Lighting is computed at the vertices and interpolated. Suppose a light source were at the center of the rectangle. For the tessellation on the left the center triangles would be more brightly lit than those at the ends. But for the tessellation on the right they would all have about the same brightness.

(*d*) Suppose that lighting were being computed in the fragment shader.

[✓] Why would either tessellation above be wasteful?

Because there would be no need for eight triangles. The entire area could be covered by just two triangles. The waste is in running the vertex shader 10 times (for eight triangles) instead of just 4 times (for two triangles).

Problem 4: (10 pts) In class we saw how we could obtain the mirror image of a vertex easily if the mirror were on the xz plane at $y = 0$. For such a mirror, the mirror image of a vertex at (x, y, z) would be at $(x, -y, z)$.

Appearing below are a vertex shader and geometry shader that do nothing special: The vertex shader computes the clip-space coordinates and lighting, and the geometry shader emits one triangle.

Modify the shader(s) so that for each original triangle two are emitted: one in the original position, and the other in the reflected position. (Don't worry about stenciling or anything like that.)

Hint: Don't forget that the mirror is in object-space (world-space) coordinates.

☑ Modify to emit reflected triangle, in addition to regular.

☑ Think about coordinate spaces.

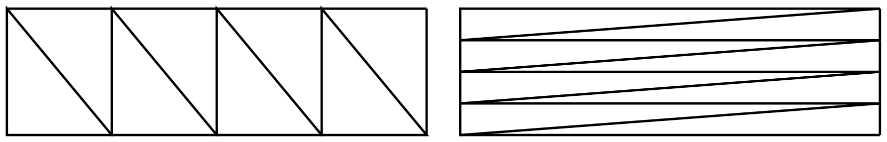
Solution shown below. The modified lines have been identified with `SOLUTION` to the right, or above and below. In the vertex shader first the reflected coordinate is computed in object space (just by flipping the sign of the y component). That is converted to clip space for use by the geometry shader. Note that flipping the sign of the y component of the clip-space coordinate would not work.

In the interface block (`in Data` and `out Data`) a new declaration was added for the reflected coordinate. In the geometry shader a second triangle is emitted.

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

  // Compute eye-space vertex coordinate and normal.
  // These are outputs of the vertex shader and inputs to the frag shader.
  vec4 var_vertex_e = gl_ModelViewMatrix * gl_Vertex;
  vec3 var_normal_e = normalize(gl_NormalMatrix * gl_Normal);
  vec4 lcolor = generic_lighting( var_vertex_e, gl_Color, var_normal_e);

  // SOLUTION below                                                    SOLUTION
  vec4 mir_vertex_o = vec4(gl_Vertex.x, -gl_Vertex.y, gl_Vertex.z, gl_Vertex.w);
  mir_vertex_c = gl_ModelViewProjectionMatrix * mir_vertex_o;
  // SOLUTION above                                                    SOLUTION

  gl_BackColor = gl_FrontColor = lcolor;
}

out Data  {
  int hidx;  // Not used.
  vec4 mir_vertex_c;                                               /// SOLUTION
};
#endif

#ifdef _GEOMETRY_SHADER_
in Data {
  int hidx;  // Not used.
  vec4 mir_vertex_c;                                               /// SOLUTION
} In[3];

void gs_main_helix() {
  /// Geometry Shader
  //
  // Pre-defined input: gl_PositionIn[] (an array, size determined by prim)
  // Pre-defined output: gl_Position (a scalar, read when EmitVertex called).
  for ( int j=0; j<2; j++ )                                        /// SOLUTION
  for ( int i=0; i<3; i++ )
    {
      gl_FrontColor = gl_FrontColorIn[i];
      gl_BackColor = gl_BackColorIn[i];
      gl_Position = j == 0 ? gl_PositionIn[i] : mir_vertex_c[i];  /// SOLUTION

      EmitVertex();
    }
  EndPrimitive();
  }                                                                /// SOLUTION
}
```

Problem 5: (15 pts) Answer each question below.

(*a*) Suppose we have a buffer object for vertices corresponding to the object on the lower left.



☑ How can we use that buffer object to render an object that will look like the one on the right?

Notice that the object on the right can be obtained from the object on the left by rotating and scaling it. So before rendering set the modelview transformation matrix to the following: $M\,T_{\vec{c}}\,R_{\hat{n},90^\circ}\,S_{0.25,4,1}\,T_{-\vec{c}}$, where $T_{-\vec{c}}$ is a translation that moves the center of the object to the origin, $S_{0.25,4,1}$ is a scale transformation that compresses the x axis by 4 and stretches the y axis by 4, and $R_{\hat{n},90^\circ}$ is a rotation matrix that rotates by 90°, and M is the existing modelview transformation.

(*b*) Describe a situation in which using `glBegin` with `glVertex` rather than vertex arrays with (`glDraw`) is a good idea.

☑ Situation in which `glVertex` better:

This would make sense when, for some reason, the number of primitives to render is small, for example, just one triangle. In that case, there is no benefit to `glDraw` and in fact the overhead of setting up the arrays would make it slower.

(*c*) Describe a situation in which using vertex arrays but without buffer objects is a good idea.

☑ Situation in which vertex arrays without buffer objects better:

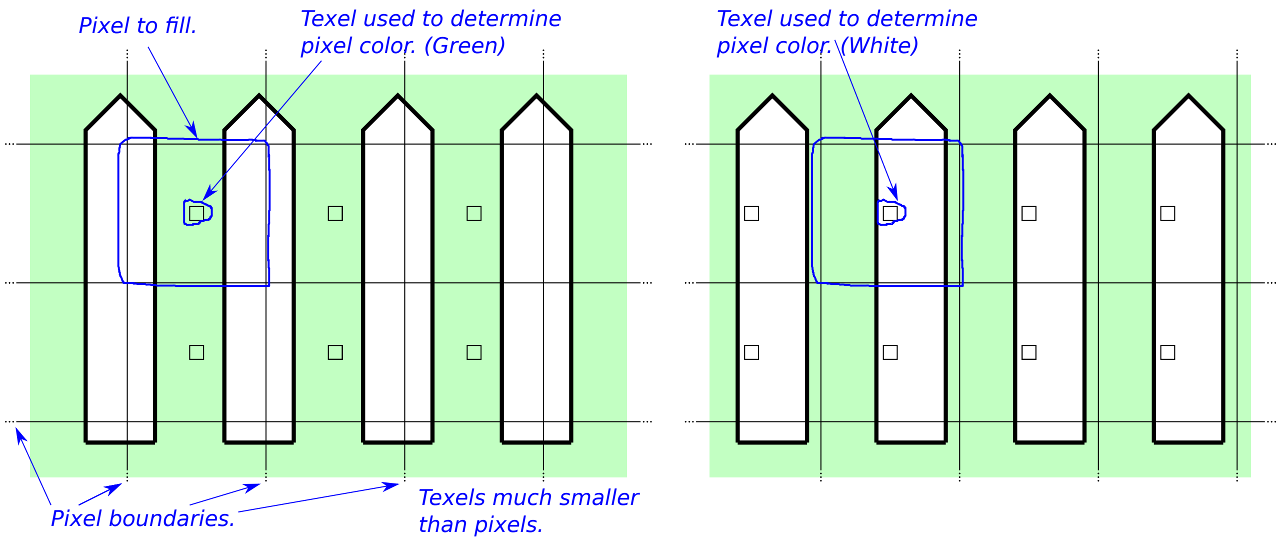
Buffer objects have an advantage over vertex arrays when the same buffer object is used multiple times. In that case the data should be sent to the GPU just once, whereas with vertex arrays the data is sent each time. So vertex arrays without buffer objects are good idea if the vertices would be used just once. This might happen because the set of triangles we are trying to render changes position every frame.

Problem 6: (10 pts) In class the topic of texture filtering was introduced with a description of what would happen if a texture image of a picket fence (white stripes in front of a green background) where rendered without an appropriate MIPMAP level and without filtering. Either the fence or the grass would disappear.

(*a*) Illustrate the problem. On your diagram show the fence, some pixels and some texels. It is important to choose the size of the pixels and texels correctly.

☑ Illustration of problem. ☑ Show pixels. ☑ Show texels.

The illustration below shows the picket fence and pixel boundaries and some texel boundaries, the texels are much smaller than the pixels. The color of a pixel is determined by the color of the texel in the center of the pixel (those are the only texels illustrated). For the left-hand figure the pixels would be green (and so the fence would not be visible). For the right hand figure the pixels would be white (and so the grass would not be visible). The problem is that while each pixel covers both fence and grass, the sampled texel covers *either* fence or grass. Note: This diagram illustrates the OpenGL `NEAREST` texture filtering method when used with just one MIPMAP level.

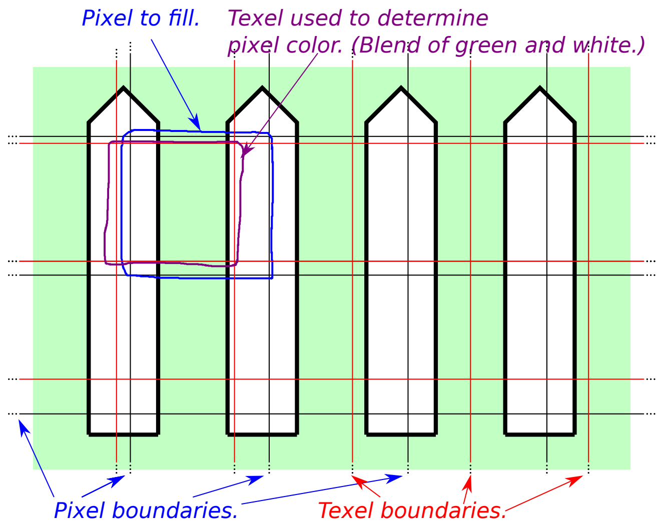

Pixel to fill.
Texel used to determine pixel color. (Green)
Pixel boundaries.
Texels much smaller than pixels.
Texel used to determine pixel color. (White)

(*b*) Re-draw the diagram showing how the choice of an appropriate MIPMAP level can avoid the problem of the fence disappearing.

☑ Illustration of how MIPMAP level avoids problem.

Re-drawn illustration shown below. In this case a higher MIPMAP level was used, meaning that the texels were larger. (A larger texel is obtained by blending the colors in the smaller texels of a lower MIPMAP level.) In this case the texels are only a little smaller than the pixels. Still, one texel is selected for each pixel. In the illustration the selected texels will be a blend of green and white. Because of the pixel size it will not be possible to discern the pickets, but the color won't suddenly change from white to green as the image moves.

Note: This diagram illustrates the OpenGL `NEAREST` texture filtering method when used with a complete set of MIPMAP levels.


Pixel to fill.
Texel used to determine pixel color. (Blend of green and white.)
Pixel boundaries.
Texel boundaries.

(*c*) Draw a diagram showing an example of how linear filtering can improve the appearance of the fence.

☑ Example of how linear filtering improves appearance.

Refer to the illustration from the previous part. The texel under the word `Blend` will not be used for any pixel because the center of the texels to the left and right are closer to the center of the respective pixels. As a result the image might appear greener than it should. With linear filtering an average of several pixels is used, this avoids the problem of skipping certain texels.