Name Solution

EE 4702

Take-Home Pre-Final Questions

Due: 9 December 2013

Please work on this exam alone. You may use references for OpenGL, CUDA, and similar topics.

Problem 1		(20 pts)
Problem 2		(30 pts)
Problem 3		(15 pts)
Problem 4		(20 pts)
Problem 5		(15 pts)
Exam Total		(100 pts)

Alias 0x5e1f

Good Luck!

Problem 1: [20 pts]The code below is the `data_unpack` routine from the Homework 3 solution. Recall that the code in Homework 3 requires that the number of threads must be no smaller than the number of balls (the value of `chain_length`).

Re-write the routine so that it works correctly even if the number of balls is greater than the number of threads.

☑ Re-write so number of threads can be fewer than `chain_length`.

☑ Compute the number of threads using CUDA-provided variables such as `blockDim`.

☑ Array `dc.d_pos` must be accessed efficiently.

The solution appears below. The code writing `ball->position` is now in a loop. The loop starts with the same ball index (`bi`) in the pre-solution code, which was set equal to the global thread id (shown as `tid` in the solution). The solution code computes the total number of threads and assigns it to `num_threads`. The increment for the loop is `num_threads`, that will guarantee that we get a value of `bi` that no other thread has.

Access is efficient because consecutive threads will have consecutive values of `bi`. We know that because `bi` was initialized to `threadIdx.x + SOMETHING` (where something is the same for all threads in a warp).

```
// SOLUTION
__global__ void data_unpack_mb() {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int num_threads = blockDim.x * gridDim.x;
  for ( int bi = tid;  bi < dc.chain_length; bi += num_threads )
    {
      Ball* const ball = &dc.d_balls[bi];
      ball->position = dc.d_pos[bi];
    }
}
```

Problem 2: [30 pts]Recall that the code for Homework 3 simulated a chain of balls (or string of beads). In this problem we'll extend that code by giving the balls electric charge, either positive or negative. The balls are sealed in a special coating that retains the charge.

The CUDA kernel below applies the force due to these charges. Like the penetration routine it must consider nearly all possible pairs. The code is written to optionally use shared memory when accessing the "b" object.

```
__global__ void time_step_gpu_sol_part_m(float delta_t) {
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;
  const int threads_per_object = num_threads / dc.chain_length;

  const int a_idx = tid % dc.chain_length;
  const int b_idx_start = tid / dc.chain_length;

  __shared__ float4 c_pos[1024];

  float4 pos_a = dc.d_pos[a_idx];                                      // Global Access A
  float3 force = make_float3(0,0,0);

  for ( int j=b_idx_start; j < dc.chain_length; j += threads_per_object )
    {
      if ( use_cache )
        {
          __syncthreads();
          if ( a_idx == 0 ) c_pos[b_idx_start] = dc.d_pos[j];         // Global Access B
          __syncthreads();
        }

      if ( a_idx != j )
        {
          float4 pos_b = use_cache ? c_pos[b_idx_start] : dc.d_pos[j];// Global Access C
          pNorm a_to_b = mn(pos_a,pos_b);
          const bool repel = ( a_idx & 1 ) == ( j & 1 );
          force += ( repel ? -0.15f : 0.15f ) / a_to_b.mag_sq * a_to_b;
        }
    }

  const float3 delta_v = delta_t * dc.d_balls[a_idx].mass_inv * force;
  atomicAdd(&dc.d_vel[a_idx].x,delta_v.x);
  atomicAdd(&dc.d_vel[a_idx].y,delta_v.y);
  atomicAdd(&dc.d_vel[a_idx].z,delta_v.z); }
```

Questions are on next page.

(a) In this part assume that `use_cache` is false, and so shared memory won't be used. Compute the amount of data requested due to accesses to `d_pos` and compute how much of that data is actually used. Do this for the following configuration: `chain_length=256`, a block size of 1024 threads, and a grid size of 16 blocks. The code runs on an NVIDIA Kepler GPU which has memory request sizes of 32, 64, and 128 bytes. For your convenience, comments show where global memory can be accessed. (Array `c_pos` is not in global memory, and is anyway won't be needed until the next subproblem.)

☑ Total size of requests (in bytes) to `d_pos` considering all threads.

☑ Amount of data in those requests that was needed.

First consider the assignment of `pos_a`. Let c be the value of `chain_length` and n be the number of threads.

We'll start with something simple: computing the total amount of data written to `pos_a` (not requested) by all the threads. The type of `pos_a` is a `float4`, which has a size of 16 bytes. The total number of threads is the product of the block size and grid size which is $n = 1024 \times 16 = 2^{10}2^4 = 2^{14} = 16384$ threads. Since each thread accesses 16 bytes the total data written to `pos_a` is $16n = 2^{14}2^4 = 2^{18} = 262144$ bytes.

This would be the total size of all requests if each byte in every request is used exactly once. If some bytes are used zero times the total request size will have to be larger than the amount of data written. If some bytes are used multiple times (because two threads in the same warp want the same data) the total request size can be smaller.

Remember that instructions in NVIDIA GPUs (so far up to CC 3.5) execute in groups called warps. When they need to load data all the threads in a warp (actually a half warp, but we won't consider that here) present an array index (actually a memory address and size) to the multiprocessor hardware. Essentially the 32 indices will be sorted, duplicates will be removed, and the remaining indices will be partitioned into *requests*. A request is a command to read memory and consists of a starting address and a size. The size can be 32, 64, or 128 bytes (for CC 2.X and 3.X devices). The goal of the multiprocessor hardware is to partition the indices so as to minimize the number of requests.

To see how this works, consider the following examples. If each thread accesses 1 byte (which does not apply to our code) and the indices are consecutive then a single 32-byte request will be sufficient for a warp. If the threads each accesses an 8-byte quantity and the indices are consecutive, the total the $32 \times 8 = 256$ bytes can be satisfied by two requests. If all threads access *the same* 8-byte quantity there will be a single request of size 32 bytes and three quarters of the request will go unused.

For Kepler (CC 3.5) devices there is no global L1 memory cache, so all requested memory must be used by the requesting warp or it will be lost. That is, if two warps access the same data, then that data must be requested twice.

Getting back to Global Access A. It should be easy to verify that consecutive threads will have consecutive values of `a_idx` and that with a 16-byte element size there is enough data to fill exactly four 128-byte requests. So each requested data item is used exactly once. Therefore for `pos_a` the total amount requested and needed data is $16n$ or 262144 bytes.

Since `use_cache` is false Global Access B does not occur. The load at Global Access C is to an element at index `j`, which starts at `b_idx_start`. It should be easy to verify that for the first 256 threads `b_idx_start` is zero, for threads 256 to 511 it is one, and so on. Therefore, within a warp all threads will have the same value of `j` and so they will all be requesting the same 16-byte value. A single 32-byte request will be dispatched.

For those following what the code is doing should see that the total number of iterations, counting all threads, will be c^2, since we are doing an all-to-all computation. Even without that leap of insight, by examining the `j` loop bounds we can see that the number of iterations is $c/(n/c) = c^2/n$. Since n threads execute the loop, the total number of executions of the body is c^2.

Each warp (or thread) executes c^2/n iterations of the loop requesting $\frac{c^2}{n}32$ bytes of data but using only half of it. The total number of warps is $n/32$ so the total data requested is $\frac{n}{32}\frac{c^2}{n}32 = c^2$ bytes, the amount of data needed is $\frac{1}{2}c^2$ bytes.

Combining the contributions from the two global accesses (ignoring Global Access B since `use_cache` is off): Total data requested is $16n + c^2 = 2^{18} + 2^{16} = 262144 + 65536 = 327680$ bytes. The amount of data needed is $16n + \frac{1}{2}c^2 = 2^{18} + 2^{15} = 262144 + 32768 = 294912$ bytes.

(*b*) Repeat the problem above for the case when `use_cache` is true. Remember that memory requests are not generated when reading or writing `c_pos` itself.

☑ Total size of requests (in bytes) to `d_pos` considering all threads.

☑ Amount of data in those requests that was needed.

The amount of data accessed by Global Access A will be the same as the `use_cache=false` case, $16n$ bytes. The amount of global data accessed for Global Access C will be zero (since it is now accessing shared memory rather than global memory).

We need to think about Global Access B, in which data from global memory is written to shared memory. As with Global Access C, this one uses index `j`. However because of the `if` statement it is not being executed by every thread. In fact `a_idx==0` will only be true for threads in which `tid` is a multiple of c, which will be for $n/c = 2^{14}/2^8 = 2^6 = 64$ threads. It should be easy to verify that `a_idx==0` for at most one thread in a warp. Therefore the request size will be 32 bytes (of which 16 are needed). The total of all requests will be $32 \times \frac{n}{c} = 32 \times 64 = 2048$ bytes per iteration or $32 \times \frac{n}{c}\frac{c^2}{n} = 32c = 8192$ bytes. The total data needed will be half that, 4096 bytes.

In summary, total size of all requests: $16n + 32c = 262144 + 8192$ bytes, amount needed is $16n + 16c = 262144 + 4096$ bytes.

(*c*) The `d_pos` accesses that are used to fill shared memory, `c_pos[b_idx_start] = dc.d_pos[j];`, are wasteful (though the waste is small in proportion to the total amount of data accessed). Explain why the accesses are wasteful and fix the problem.

☑ Fix problem.

☑ Why are these accesses wasteful?

The accesses are wasteful because at most one thread in a warp is active, and so half the request goes unused. (See the solution to the previous problem.) The code below fixes the problem by having the first few threads load shared memory.

```
// SOLUTION
const int b_idx_start_thd_0 = blockIdx.x * blockDim.x / dc.chain_length;
const int threads_per_object_per_block = blockDim.x / dc.chain_length;
int b_to_shared_idx =
  b_idx_start_thd_0 + threadIdx.x % threads_per_object_per_block;

for ( int j=b_idx_start; j < dc.chain_length; j += threads_per_object )
  {
    if ( use_cache )
      {
        __syncthreads();
        // SOLUTION
        if ( threadIdx.x < threads_per_object_per_block )
          {
            c_pos[b_idx_start_thd_0+threadIdx.x] = dc.d_pos[b_to_shared_idx];
            b_to_shared_idx += threads_per_object;
          }
        __syncthreads();
      } }
```

(*d*) The amount of memory read when assigning `pos_a` can be reduced by using shared memory. Modify the code to do so.

☑ Use shared memory to reduce redundant global accesses to initialize `pos_a`.

Solution appears below. The values of `a_idx` are already consecutive, but they repeat four times (for the sample values given for the problem) within a block. Because of this repetition each position is loaded from global memory four times. In the solution each position is loaded only once, a position is written to shared memory by one thread and then read by all the threads that need it.

```
// SOLUTION
__shared__ float4 c_pos[1024];
if ( threadIdx.x < dc.chain_length ) c_pos[threadIdx.x] = dc.d_pos[a_idx];
__syncthreads();
float4 pos_a = c_pos[a_idx];
```

Problem 3: [15 pts]Consider the following excerpt from the Cone code from the Homework 2 solution. (File `hw2-sol.cc` or visit `http://www.ece.lsu.edu/koppel/gpup/gpup/2013/hw2-sol.cc.html`).

```
void render_probb(pCoor base, float radius, pVect to_apex) {
    if ( opt_lod != bo_lod ) { // If needed lod != stored lod.
        bo_lod = opt_lod;

        /// Code for computing coordinates and normals omitted.
        if ( !buffer_obj_coords ) {
            glGenBuffers(1,&buffer_obj_coords);
            glGenBuffers(1,&buffer_obj_norms);
            glGenBuffers(1,&buffer_obj_colors); }                 /// SOLUTION

        glBindBuffer(GL_ARRAY_BUFFER, buffer_obj_coords);
        glBufferData(GL_ARRAY_BUFFER, coords.occ() * sizeof(float),
                     coords.get_storage(), GL_STATIC_DRAW);

        glBindBuffer(GL_ARRAY_BUFFER, buffer_obj_norms);
        glBufferData(GL_ARRAY_BUFFER, norms.occ() * sizeof(float),
                     norms.get_storage(), GL_STATIC_DRAW);

        /// SOLUTION BELOW
        PStack<float> repcolors;
        for ( int i=0; i<num_coords; i++ )
          { repcolors += color.r; repcolors += color.g; repcolors += color.b; }
        glBindBuffer(GL_ARRAY_BUFFER, buffer_obj_colors);
        glBufferData(GL_ARRAY_BUFFER, repcolors.occ() * sizeof(float),
                     repcolors.get_storage(), GL_STATIC_DRAW);
        /// SOLUTION ABOVE
      }
    // Transform computation and modelview update omitted.
    glBindBuffer(GL_ARRAY_BUFFER, buffer_obj_colors);             /// SOLUTION
    glColorPointer( 3, GL_FLOAT, 0, NULL );                       /// SOLUTION
    glEnableClientState(GL_COLOR_ARRAY);                          /// SOLUTION

    // Tell OpenGL to get coordinates and normals from buffer objects.
    glBindBuffer(GL_ARRAY_BUFFER, buffer_obj_coords);
    glVertexPointer( 3, GL_FLOAT, 0, NULL);
    glEnableClientState(GL_VERTEX_ARRAY);

    glBindBuffer(GL_ARRAY_BUFFER, buffer_obj_norms);
    glNormalPointer(GL_FLOAT,0,NULL);
    glEnableClientState(GL_NORMAL_ARRAY);

    // Draw the cones. (There will be minor flaws since 1 strip used.)
    glDrawArrays(GL_QUAD_STRIP,0,num_coords);

    // Cleanup code omitted.
```

Questions are on next page.

Problem 3, continued:

(*a*) The code uses buffer objects for vertex coordinates and normals. Why doesn't it also use a buffer object for color?

☑ Why doesn't color need a buffer object?

Because all vertices are assigned the same color. The way OpenGL works, that one color is sent to the GPU just once (rather than once for each vertex), and so there would be little benefit to using a buffer object.

(*b*) Modify the code so that it uses a buffer object for color, even if that's not a good idea.

☑ Modify code so color *does* use a buffer object for color.

Solution appears on the previous page. First, a buffer object was prepared which contained multiple copies of `color`.

In the rendering code function `glColorPointer` was used to indicate that colors would be specified in an array. The function `glColorPointer` was never used in class, but could be found in the OpenGL 4.3 specification section 10.3.2. That section also gives the constant to use with `glEnableClientState`, GL_COLOR_ARRAY.

(*c*) Suppose that each time the routine above was called `opt_lod` had a different value. Also suppose that `if ( !buffer_obj_coords )` were somehow changed to `if ( true )`.

☑ What would eventually happen?

By changing the `if` condition to always be true, a new buffer object name is created each time the code is called. The old buffer objects are not destroyed, they continue to hold their data. So the system would eventually run out of memory. (With the `if` condition as it was, the old buffer object name would be re-used on a second call so the data in the old buffer object would be replaced by the new data.)

Problem 4: [20 pts]Consider another excerpt from the Cone code from the Homework 2 solution. (File `hw2-sol.cc` starting at line 266. or visit
`http://www.ece.lsu.edu/koppel/gpup/gpup/2013/hw2-sol.cc.html`).

```
// Compute transform moving and positioning code from local to global
// space.
//
const float to_height = to_apex.mag();
pVect from_apex(0,0,1);
pNorm rn(from_apex,to_apex);
const float rot_angle = pangle(from_apex,to_apex);
pMatrix_Translate trans_transl(base);
pMatrix_Rotation trans_rot(rn,rot_angle);
pMatrix_Scale trans_scale(radius);
trans_scale.rc(2,2) = to_height;
pMatrix xform = trans_transl * trans_rot * trans_scale;

// Specify our transformation to OpenGL.
//
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glMultTransposeMatrixf(xform.a);

if ( !dont_set_color ) glColor3fv(color);

glBindBuffer(GL_ARRAY_BUFFER, buffer_obj_coords);
glVertexPointer( 3, GL_FLOAT, 0, NULL);
glEnableClientState(GL_VERTEX_ARRAY);

glBindBuffer(GL_ARRAY_BUFFER, buffer_obj_norms);
glNormalPointer(GL_FLOAT,0,NULL);
glEnableClientState(GL_NORMAL_ARRAY);

glDrawArrays(GL_QUAD_STRIP,0,num_coords);
```

The code at the top of the excerpt computes a transform matrix that will position and scale the cone to the desired location. The transform is computed in terms of coordinate `base`, scalar `radius`, and vector `to_apex`. The command `glMultTransposeMatrixf` takes the transform matrix we computed and multiplies it by the existing modelview matrix. The resulting matrix is written as an OpenGL Shading Language uniform with name `gl_ModelViewMatrix`.

In this problem we'll consider computing the transform on the GPU instead. The GPU code will compute the transform in terms of `base`, `radius`, and `to_apex`.

Questions are on next page.

Problem 4, continued:

(*a*) Declare `base`, `radius`, and `to_apex` as uniforms as they would be in an OpenGL SL source file. *Hint: Use the demo-10 (Vertex and Geometry Shaders) code as an example, and look for* `wire_radius`*. The demo-10 code can be accessed from the repo or*
`http://www.ece.lsu.edu/koppel/gpup/code/gpup/demo-10-shader.cc.html` *(CPU code),*
`http://www.ece.lsu.edu/koppel/gpup/code/gpup/demo-10-shdr-simple.cc.html` *(simple shaders), and*
`http://www.ece.lsu.edu/koppel/gpup/code/gpup/demo-10-shdr-geo.cc.html` *(more elaborate shaders).*

Solution appears below. Notice that the names of the vector data types in OpenGL shading language and CUDA differ, so base is `vec4`, not `float4`.

```
// SOLUTION
layout ( location = 1 ) uniform vec4 base;
layout ( location = 2 ) uniform float radius;
layout ( location = 3 ) uniform vec3 to_apex;
```

(*b*) Write code to send the uniforms from the CPU to the GPU. *Hint: See the previous hint.*

The solution appears below. The command **`glUniformX`** writes a value into a uniform declared in an OpenGL shading language program (as was done for the previous part). The first argument specifies the location, see above. The remaining arguments are the values. Note that the digit near the end of the command specifies the number of arguments and that the last letter specifies the data type, float in this case.

```
    glUniform4f(1, base.x, base.y, base.z, base.w );
    glUniform1f(2, radius);
    glUniform3f(3, to_apex.x, to_apex.y, to_apex.z);
```

(*c*) Suppose that `xform` is computed in a vertex shader making use of the enormous floating-point capability of the mighty GPU. Do you expect execution to be faster or slower than using the CPU to compute xform? Explain.

Slower than the CPU. An individual GPU thread is slower than a CPU thread. The CPU computes the transform **once** for use by all vertices, so the GPU's parallelism is no advantage. Each vertex shader would need to compute the transform and would take more time to do so than the CPU.

(*d*) Suppose that `xform` is computed in a geometry shader. Will execution be faster or slower than using the vertex shader when we are rendering quad strips? (That is, the systems we are comparing both use quad strips, one uses the vertex shader to compute the xform, the other uses the geometry shader.) Explain. *Note: I should have asked about triangle strips.*

Assuming that lighting and other calculations performed by the vertex shader to not take much time, the geometry shader will be faster. With quad strips, most vertices are shared by two quads and so the number of vertices is twice the number of quads. That means the transform would be computed twice as many times if the vertex shader were used. (The answer would be the same number of times if triangle strips were used.)

(*e*) Suppose that `xform` is computed in a geometry shader. Will execution be faster or slower than using the vertex shader when we are rendering individual quads? (That is, the systems we are comparing both use individual quads, one uses the vertex shader to compute the xform, the other uses the geometry shader.) Explain.

With individual quads the geometry shader has a big advantage, since the vertex shader is called four times for each quad and so the transform would be computed four times as often if the vertex shader were used.

Problem 5: [15 pts]Answer each question below.

(*a*) Texture access is performed in the fragment shader. Suppose texture access was performed by the vertex shader and the texel values were interpolated, the same way other attributes such as color were.

☑ What impact would that have on appearance?

Yuck! Suppose the texture image was of a house, and the whole house was supposed to appear on the primitive. Since texels are grabbed at the vertices, only three texels would be retrieved and their colors would be interpolated. Unless the triangle were small the image of the house would be unrecognizable.

☑ What impact would that have on performance?

Performance would be alot higher since texture access and filtering is computationally costly and the fragment shader is called many more times than the vertex shader for typical primitives.

(*b*) Inputs to a fragment shader have interpolation qualifiers, they are `flat`, `noperspective`, and `smooth`. Why are they not needed for the inputs to the geometry shader?

☑ Interpolation qualifiers not needed for geometry shader inputs because:

Interpolation only has meaning for fragments. (The points that fill the area of a primitive.) The geometry shader operates on primitives, say the three vertices of a triangle. So there is nothing to interpolate.

(*c*) The OpenGL command `glMatrixMode(GL_MODELVIEW)`, used in setting the modelview matrix, is deprecated. (Meaning that its use is discouraged and that it may be removed from the language.) Why was the command once essential but now considered unnecessary?

☑ `glMatrixMode` now unnecessary because:

Because parts of the rendering pipeline are now programmable, including the vertex shader. The vertex shader is supposed to transform object-space coordinates to clip space. The fixed-functionality used to do that using, in part, the modelview matrix. Since the fixed functionality no longer computes clip space coordinates there is no need to have a way of specifying a modelview matrix. The use can pass the model view matrix (or the user's equivalent) as a user-defined variable.