Name _____

EE 7700-1

Take-Home Final Examination

Tuesday, 5 May 2009 to Early Monday, 11 May 2009

Problem 1	_____	(20 pts)
Problem 2	_____	(20 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(20 pts)
Problem 5	_____	(10 pts)
Problem 6	_____	(10 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [20 pts]The kernel below runs on a GE 80 style GPU, the native assembly language (according to Decuda) appears below the kernel.

```
__device__ void prob1()
{
  const int idx = blockDim.x * blockIdx.x + threadIdx.x;
  float pf = global_f1[idx] * global_f2[idx];
  float pg = global_g1[idx] * global_g2[idx];
  global_r[idx] = pf + pg;
}
```

```
 # Native assembly code (according to Decuda).
 0:mov.b16 $r0.hi, %ntid.y
 1:cvt.u32.u16 $r1, $r0.lo
 2:mad24.lo.u32.u16.u16.u32 $r0, s[0x000c], $r0.hi, $r1
 3:shl.u32 $r0, $r0, 0x00000002
 4:add.half.b32 $r1, $r0, c0[0x0010]    // <- Constant load.
 5:add.half.b32 $r4, $r0, c0[0x0018]
 6:add.half.b32 $r3, $r0, c0[0x000c]
 7:add.half.b32 $r5, $r0, c0[0x0014]
 8:mov.u32 $r2, g[$r1]                  //  <- Global load
 9:mov.u32 $r1, g[$r4]
10:mov.u32 $r4, g[$r3]
11:mov.u32 $r3, g[$r5]
12:mul.rn.f32 $r1, $r2, $r1
13:mad.rn.f32 $r1, $r4, $r3, $r1
14:add.u32 $r0, $r0, c0[0x0020]
15:mov.end.u32 g[$r0], $r1             //  <- Global store (ignore latency)
```

- The GPU has 14 multiprocessors, the clock frequency is 1.5 GHz.
- All instructions above can issue in 4 cycles (don't confuse that with latency).
- Global memory latency is 600 cycles, non-memory instruction latency is 24 cycles.
- All loads and stores are coherent, ignore the latency for the store.

(*a*) What would be the execution time for 2^{24} threads assuming that there was no limit on block size, threads per MP, or the number of active warps per thread? Show your work and briefly explain your answer.

(*b*) What is the minimum number of threads per MP needed to avoid stalls? Be sure to take the distance between *dependent* instructions into account. Explain your answer.

(*c*) For this code sample, what is the maximum global memory latency that can be hidden by 96 warps per MP?

Problem 2: [20 pts]The code below is the PTX version of the kernel from the previous problem. This is not supposed to match the true machine, but is useful for some purposes. For this problem lets assume the PTX is the true machine language. The goal is to compare the performance of this code to the decuda code. (The PTX code is generated in an earlier stage of the build process and so is less optimized than the decuda-revealed code.)

```
 0:        mov.u16         %rh1, %ctaid.x;
 1:        mov.u16         %rh2, %ntid.x;
 2:        mul.wide.u16    %r1, %rh1, %rh2;
 3:        cvt.u32.u16     %r2, %tid.x;
 4:        add.u32         %r3, %r2, %r1;
 5:        mul.lo.u32      %r4, %r3, 4;
 6:        ld.const.u32    %r5, [global_g1];
 7:        add.u32         %r6, %r5, %r4;
 8:        ld.global.f32   %f1, [%r6+0];         // <-- Global load.
 9:        ld.const.u32    %r7, [global_g2];
10:        add.u32         %r8, %r7, %r4;
11:        ld.global.f32   %f2, [%r8+0];
12:        mul.f32         %f3, %f1, %f2;
13:        ld.const.u32    %r9, [global_f1];
14:        add.u32         %r10, %r9, %r4;
15:        ld.global.f32   %f4, [%r10+0];
16:        ld.const.u32    %r11, [global_f2];
17:        add.u32         %r12, %r11, %r4;
18:        ld.global.f32   %f5, [%r12+0];
19:        mad.f32         %f6, %f4, %f5, %f3;
20:        ld.const.u32    %r13, [global_r];
21:        add.u32         %r14, %r13, %r4;
22:        st.global.f32   [%r14+0], %f6;
```

(*a*) Explain why more threads would be needed to hide memory latency with this code than with the code revealed by decuda. *Hint: It has to do with the placement of the load instructions. Hint: It has* **nothing** *to do with the fact that there are more instructions.*

(*b*) Fix the problem by modifying the PTX (without changing what it does). *Hint: It's very easy to do.*

Problem 3: [20 pts]A scene contains many copies of the same object, but at different locations.

Suppose each object has v vertices and each vertex has 40 bytes of data associated with it, including its coordinates. There are a total of n objects.

(a) In the implementation below a prototype object is sent n times, each time with a different transformation matrix. The data is sent from client memory. Estimate the amount of data sent for this code. Please show work and state any assumptions.

```
for ( int i=0; i<n; i++ ) { // Render Object i
    glPushMatrix();
    glMultMatrixf(mat[i]);  // Multiply current transform by mat[i]
    glNormalPointer(...) glVertexPointer(...);
    glDrawArrays(...,v);    // Render the v vertices in the object.
    glPopMatrix();          // Restore previous transform.
  }
```

(b) Suppose the code above were modified so that just before the `for` loop the object were loaded into a buffer object and that buffer object was used to reduce data transfer costs. Estimate the amount of data sent by this code. Show work and state assumptions.

(c) Why is the overhead of changing transformation matrices high compared to the amount of data in a matrix and the time needed to send it?

Problem 4: [20 pts]Continue to consider the scene with n copies of the same v-vertex object. Suppose that a translation transformation was all that was necessary to move the prototype object into the desired position. (Each of the n objects would still have its own translation.)

In this problem a vertex shader will be used to avoid the need for changing the transformation matrix for each object.

(a) The code below shows the start of a vertex shader for the program. Add code to the vertex shader so that objects are properly moved. The solution should contain declarations.

```
// Solution declarations here.

void vs_main()
{
 // Solution code here.



 // Assume existing shader starts here.
 ...
}
```

(b) Show what the matrix load commands would be replaced with in the code below. **Do not** show setup code, such as code loading the shader. Only show the code needed inside the loop.

```
  setup_shader(..);
  for ( int i=0; i<n; i++ ) { // Render Object i
      //  Show replacement for glPushMatrix, etc.



      glNormalPointer(...) glVertexPointer(...);
      glDrawArrays(...,v);    // Render the v vertices in the object.
      // Don't need anymore: glPopMatrix();
    }
```

Problem 5: [10 pts]In the Geforce 3 (as described in Lindholm 2001) there is no performance penalty for using an attribute for a value that doesn't change. For example, in the code below sending `attrib\_value` as an attribute takes no more time than sending a uniform.

```
 glUniform4fv(uniform_name,uniform_value);
 glVertexAttrib4fv(attrib_name,attrib_value);  // Attribute 1
 glBegin(GL_TRIANGLES);
 ...  glColor4f(r,g,b,a); // different for every vertex.
 glEnd();
}
```

(*a*) How does the GE 3 keep the cost (impact on performance) of sending the attribute closer to the cost of the uniform than to the cost of the color?

(*b*) On something like the GE 80 (8000 series) why might the cost of the unchanging attribute be closer to that of the color, which changes with every vertex? Base your answer on the GE 80 structure described for CUDA.

Problem 6: [10 pts]OpenGL specifies an elaborate lighting model.

(*a*) How could one argue that the lighting model limits either the design of early GPUs, or what a user can get out of them.

(*b*) Why was the lighting model necessary for earlier GPUs, even if it was limiting?

(*c*) Why is the lighting model (which hasn't changed) no longer limiting?