Name ______________________________

EE 7700-1

Take-Home Pre-Final Examination

Monday, 3 May 2010 to Early Morning Friday, 7 May 2010

Work on this exam alone. Regular class resources, such as notes, papers, documentation, and code, can be used to find solutions. Do not try to seek out references that specifically answer any question here. Do not discuss this exam with classmates or anyone else. Any questions or concerns about problems should be directed to Dr. Koppelman.

Problem 1	________	(16 pts)
Problem 2	________	(15 pts)
Problem 3	________	(15 pts)
Problem 4	________	(12 pts)
Problem 5	________	(8 pts)
Problem 6	________	(12 pts)
Problem 7	________	(10 pts)
Problem 8	________	(12 pts)
Exam Total	________	(100 pts)

Alias ______________________________

Good Luck!

Problem 1: [16 pts] The cuda kernel below performs a binary search on a table and uses the value found to perform a calculation. At the beginning of the kernel the table is copied to shared memory and the binary search operates on that copy.

```
__constant__ float *stable;   // Some sorted table.
__constant__ float *targets;
__constant__ float *results;
__constant__ int block_size_lg;
const int list_size_lg = 11;

__global__ void bsearch()
{
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
  const int tsize = 1 << list_size_lg;
  const int my_start = threadIdx.x << ( list_size_lg - block_size_lg );
  const int my_share = 1 << ( list_size_lg - block_size_lg );
  const int my_stop = my_start + my_share;

  // Copy stable to shared memory.
  __shared__ float stable_copy[tsize];
  for ( int i=my_start; i<my_stop; i++ ) stable_copy[i] = stable[i];

  __syncthreads();

  int ti = 0;      // Table indexed, used in binary search.
  const float target = targets[idx];

  // Perform binary search to find  closest element to target.
  for ( int tree_level = list_size_lg - 1; tree_level >= 0; tree_level-- )
    {
      const int ti_maybe = ti | ( 1 << tree_level );
      const float elt = stable_copy[ti_maybe];
      if ( target < elt ) continue;
      ti = ti_maybe;
    }

  // A calculation using the value that was found.
  const float diff = stable_copy[ti] - target;
  results[idx] = diff * diff;
}
```

Continued on next page.

(*a*) Suppose the kernel is launched in a grid of just one block, and the block size is 256, the value of `list_size_lg` is 11, and the kernel is run on a GPU of compute capability 1.3 as described by the CUDA Programming Guide 3.0. What is the total amount of requested global memory data per thread?

(*b*) Explain why the global memory access pattern is wasteful and show how it can be improved with a code modification.

(*c*) Would a texture cache improve access to `stable`? Explain, and be specific referring to values of `idx` and `i` in your explanation.

(*d*) The amount of serialization (the need to take extra cycles to issue instructions) due to shared memory access varies with the value of `tree_level`. Let a serialization degree of 0 indicate no serialization, meaning that instructions do not spend any extra time in a core, a 1 means one extra cycle (because memory instructions must be issued in two passes), etc. Plot the best-case and worst-case serialization degrees (y-axis) versus `tree_level` (x-axis). *Hint: In the best case the value of* `target` *is the same for all threads.*

Do this for a device of compute capability 1.3.

(*e*) Repeat the problem above for Fermi, that is a device of compute capability 2.0.

Problem 2: [15 pts] The code below can suffer from branch divergence when run on a device of CC 1.3. Suppose that without branch divergence core utilization would be 100%. That is, every core is busy with an active thread every cycle. Also suppose the code is launched in a configuration with block size $128 \times 3 = 384$, and 128 threads find a value of `0` for `my_plane`, 128 threads find a value of `1` for `my_plane`, and 128 threads find 2.

```
__device__ void diverge(int tsize)
{
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
  const float4 my_pos = targets[idx];
  const int my_plane = planes[idx];
  float distss = 0;

  for ( int i=0; i<tsize; i++ )
    {
      float4 elt = table[i];
      float dist;
      switch ( my_plane ) {
      case 0: dist = my_pos.x - elt.x; break;
      case 1: dist = my_pos.y - elt.y; break;
      case 2: dist = my_pos.z - elt.z; break;
      }
      distss += dist * dist + 1.0f / dist;
    }

  results[idx] = distss;
}
```

(*a*) With the distribution of `my_plane` values given above, it is possible that there will be no branch divergence at all or that every thread suffers the maximum divergence. Describe the conditions necessary for each extreme.

☐ Conditions for no divergence.

☐ Conditions for maximum divergence.

(*b*) Provide an estimate of core utilization under the worst-case divergence. Several assumptions need to be stated about the code to answer this question, such as the number of instructions in various parts of the code. In terms of these provide a formula or number for core utilization.

Problem 3: [15 pts] The kernel below can execute two ways, if `opt_plan_a` is true then it executes the same as the code in the previous problem. If `opt_plan_a` is false, call that Plan B, then the alternative code runs which does the same computation but avoids the switch statement overhead inside the loop.

```
__device__ void
plan_a_or_b(int tsize)
{
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
  const float4 my_pos = targets[idx];
  const int my_plane = planes[idx];
  float distss = 0;

  if ( opt_plan_a )
    { // Plan A
      for ( int i=0; i<tsize; i++ )
        {
          float4 elt = table[i];
          float dist;
          switch ( my_plane ) {
          case 0: dist = my_pos.x - elt.x; break;
          case 1: dist = my_pos.y - elt.y; break;
          case 2: dist = my_pos.z - elt.z; break; }
          distss += dist * dist + 1.0f / dist;
        }
    } else
    { // Plan B
#     define LOOP(comp) \
      for ( int i=0; i<tsize; i++ ) \
        {  const float dist = my_pos.comp - table[i].comp; \
          distss += dist * dist + 1.0f / dist;  }

      // Each case statement below executes entire loop, not just an iteration.
      switch ( my_plane ) {  case 0: LOOP(x); break;
                             case 1: LOOP(y); break;
                             case 2: LOOP(z); break;  }
    }

  results[idx] = distss;
}
```

(*a*) If there is no divergence Plan B is better. Which is better if there is divergence, Plan A or Plan B? Explain.

(*b*) Modify the code so that `opt_plan_a` is set to false if there is no divergence. The kernel must determine whether there is divergence, it can't just read a pre-computed yes/no value.

Problem 4: [12 pts] Answer the following GF3 instructions.

The instructions in the GF3 can swizzle (rearrange vector components) and negate any of their vector operands. Consider an alternative design in which instructions could not swizzle and negate their operands, instead a separate swizzle instruction would have to be executed.

```
! GF3 Code
 add r1, r2.yzwx, -r3.zyxw

! Alternative Design, need to call new swizzle instructions.
 swizzle r12, r2.yzwz
 swizzle r13, -r3.zyxw
 add r1, r12, r13
```

(*a*) There is an obvious reason why this would reduce hardware cost (ignoring the need for more instruction memory). What is it?

(*b*) Why might the alternative design need fewer input and output buffers? That is, why might fewer threads be needed to attain full efficiency? *Hint: Think about how the number of threads is chosen.*

(*c*) The following gf3 question is unrelated to the two above. The design of the gf3 limited the number of vertex attributes to 16. Describe why the following statement is not fully true: "Increasing the number of attributes to 32 would require more data transfer time (or instead bandwidth) slowing the processing of all vertices while only providing a functional benefit to vertices using more than 16 attributes."

Problem 5: [8 pts] Answer each texture-related question below.

(*a*) Explain why long lines appropriate for a CPU cache is not the best organization for a texture cache.

(*b*) The texture mipmap level can be chosen in a vertex processor or a fragment processor. Explain why higher quality results are obtainable if a mipmap level is chosen in a fragment processor.

Problem 6: [12 pts] Answer each question below.

(*a*) Based on the g80 organization and the precedent of gf3, we expect that frequently changing the transformation matrix may slow down execution, whereas frequently changing the color of a vertex would not slow down execution. Why do we expect a matrix change to slow down execution whereas changing color does not. In your answer consider a case in which each vertex has a different color, but the transformation matrix is changed once every eight vertices.

(*b*) Consider a GPU microarchitecture like the g80 except that a warp has only 8 threads. Other things are unchanged, in particular, an SM still has 8 cores and instruction latencies are unchanged. Each cycle the scheduler chooses a different warp to execute. (On the g80 and gt200 a new warp is chosen every four cycles.)

☐ How might the 8-thread warp affect hardware cost or clock frequency?

(*c*) In the GTX 8800 (and similar GPUs) frame buffer update is performed by separate raster op processors (rops). The rops are not programmable and so we are stuck with the frame buffer update options provided. For example, we might want to update the frame buffer red channel with $a_c r_c + (1 - a_c) a_f r_f$, where a_c and r_c are alpha and red-channel values for the fragment with the smaller z value and a_f and r_f are alpha and red-channel values for the fragment with the larger z value. There is no such update option in OpenGL and won't can't implement it in a fragment shader because one can't read the frame buffer. (The fragment shader knows it's values of a, r, and z but it does not know if they are the close or far set.)

Why not let the fragment shaders read the frame buffer in GPUs such as the GTX 8800 series?

Problem 7: [10 pts] Two code fragments appear below, one using a triangle strip, the other using individual triangles. Both render the same triangles. Comment on the amount of benefit realized by using triangle strips and explain why there is benefit or why there isn't benefit.

```
        // Using individual triangles
        glBindBuffer(GL_ARRAY_BUFFER,gpu_coor_buffer_individual_triangles);
        glVertexPointer(3,GL_FLOAT,sizeof(pCoor),NULL);
        glBindBuffer(GL_ARRAY_BUFFER,0);
        glEnableClientState(GL_VERTEX_ARRAY);
        glDrawArrays(GL_TRIANGLES,0,num_v_individual_triangles);
        glDisableClientState(GL_VERTEX_ARRAY);

// NOTE: Either this code runs or the one above, not both.

        // Using triangle strips
        glBindBuffer(GL_ARRAY_BUFFER,gpu_coor_buffer_triangle_strip);
        glVertexPointer(3,GL_FLOAT,sizeof(pCoor),NULL);
        glBindBuffer(GL_ARRAY_BUFFER,0);
        glEnableClientState(GL_VERTEX_ARRAY);
        glDrawArrays(GL_TRIANGLE_STRIP,0,num_v_triangle_strips);
        glDisableClientState(GL_VERTEX_ARRAY);
```

☐ Benefit for GPU / CPU communication, reason.

☐ Benefit for vertex shader workload, reason.

☐ Benefit for fragment shader workload, reason.

Problem 8: [12 pts] Below are CUDA code fragments followed by generated PTX code. Describe the problem revealed by the PTX and how the code might be improved using this info.

(*a*) The CUDA code below does something reckless (or foolish or overly trusting), but the PTX code shows that the compiler was smart, avoiding the problem.

```
 // CUDA

      const float3 pos1 = xyz(balls_x.position[bidx1]);
      const float radius1 = balls_x.position[bidx1].w;
      const float3 vel1 = xyz(balls_x.velocity[bidx1]);

 // PTX Code
ld.const.u64  %rd11, [balls_x+0];
add.u64  %rd12, %rd10, %rd11;
ld.global.v4.f32  {%f1,%f2,%f3,%f4}, [%rd12+0];
.loc 3 1029 0
ld.const.u64  %rd13, [balls_x+16];
add.u64  %rd14, %rd10, %rd13;
ld.global.v4.f32  {%f5,%f6,%f7,_}, [%rd14+0];
```

☐ Explain, what the compiler did.

☐ Modify the PTX code, showing how a less capable compiler would not save the user from reckless coding.

Problem 8, continued:

(*b*) Shown below are portions of cuda and the corresponding ptx code used to collect offsets. (This is an alternate implementation of the code in the repo.) The offsets are held in a union. For writing individual offsets the c member is used, which is an array of eight characters. Once all offsets have been written into the array the entire array is written to memory using the integer member.

```
 // CUDA Code  (Excerpts, surrounding code not shown.)

  // Initialize
  union { int64_t i; char c[8];} offsets_u;
  offsets_u.i = 0;

  // Assign offset  (this code within a loop)
  offsets_u.c[proximity_cnt] = offset;

  // Write offset to memory.
  cuda_prox[idx9] = offsets_u.i;


 // PTX Code

        // Initialize
mov.s64  %rd15, 0;
st.local.s64  [__cuda___cuda_offsets_u_0104+0], %rd15;

        // Assign offset
cvt.u64.s32  %rd25, %r11;
add.u64  %rd26, %rd25, %rd16;
st.local.s8  [%rd26+0], %r15;


        // Write offset to memory.
ld.local.s64  %rd27, [__cuda___cuda_offsets_u_0104+0];
st.global.s64  [%rd7+0], %rd27;
```

The PTX code indicates that execution will be slow.

☐ Why will execution be slow?

☐ What about the CUDA code led the compiler generate this slow code?