Name _____

EE 7700-1

Take-Home Final Examination

Monday, 10 May 2010 to Friday, 14 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	_____	(15 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(14 pts)
Problem 4	_____	(14 pts)
Problem 5	_____	(14 pts)
Problem 6	_____	(14 pts)
Problem 7	_____	(14 pts)
Exam Total	_____	(100 pts)

Alias _____

Good Luck!

Problem 1: [15 pts] The code below finds the location of all occurrences of zero in array `strings`, and writes the locations to array `z_locations`. It accesses global memory inefficiently.

```
__global__ void
find_zeros()
{
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
  const int my_start = iter_per_thread * idx;
  const int my_stop = my_start + iter_per_thread;
  __shared__ int z_count;
  if ( threadIdx == 0 ) z_count = z_count_starts[blockIdx.x];
  __syncthreads();
  for ( int i=my_start; i<my_stop; i++ ) {
      char c = strings[i];
      if ( c != 0 ) continue;
      const int z_idx_idx = atomicAdd(&z_count,1);
      z_idx[z_idx_idx] = i;
    }
}
```

(*a*) The code above accesses memory inefficiently. Indicate whether the following statement is true or false and explain why: "We could avoid inefficient memory access by accessing array elements in a different order, but if we have enough threads we don't have to because there will be enough threads to hide the memory access latency."

(*b*) Regardless of your answer to the previous part, rewrite the code so that memory access of the array is done with 100% efficiency on devices of compute capability 1.3. Pay attention to the memory request sizes that are available.

Problem 2: [15 pts] The code below computes a prefix sum of array `bins`.

```
__device__ int
prefix_sum(int val)
{
  const int tid = threadIdx.x;
  bins[tid] = val;
  __syncthreads();
  for ( int bit = 0; bit < b_lg; bit++ ) {
      const uint32_t writer_vec = 1 << bit;
      const uint32_t read_pos = writer_vec - 1;
      __syncthreads();
      if ( tid & writer_vec ) {
          const int tid_other = ( tid ^ writer_vec ) | read_pos;
          bins[tid] += bins[tid_other];
        }
    }
  return bins[tid] - val;
}
```

(*a*) The second `__syncthreads()` call, which slows down execution, is not needed every iteration. Modify the code so that it is only called when needed. Also, explain why it isn't sometimes needed.

(*b*) Show the amount of serialization in accesses to `bins` in terms of the loop index, `bit`. A serialization of 1 means a warp has to be issued one extra time, etc.

Problem 3: [14 pts] The code below, based on the code from the previous problem, avoids the serialization problem (without changing the result), at the cost of requiring `__syncthreads()` at every iteration.

```
__device__ int
prefix_sum(int val)
{
  const int tid = threadIdx.x;
  bins[tid] = val;
  __syncthreads();
  for ( int bit = 0; bit < b_lg; bit++ )
    {
      __syncthreads();
      const int tid_other = tid - ( 1 << bit );
      if ( tid_other < 0 ) continue;
      bins[tid] += bins[tid_other];
    }
  return bins[tid] - val;
}
```

(*a*) Why must the second `__syncthreads()` be executed on every iteration (unlike in the previous problem)?

(*b*) Estimate which is faster based on the CUDA programming guide. Explain.

Problem 4: [14 pts] On g80 and similar systems many instructions (such as floating point add, multiply, integer operations) have a latency of 24 cycles.

```
I0: mul.rn.f32 $r3, $r6, $r26          // r3 = r6 * r25
I1: mad.rn.f32 $r1, $r27, $r7, $r1     // r1 = r27 * r7 + r1
I2: mad.rn.f32 $r3, $r5, $r27, $r3
I3: add.half.rn.f32 $r10, $r11, $r2
```

(*a*) The code above is launched in a kernel with many threads per block. Let W0, W1, ... denote the warps. and let W0I0 indicate the issuing (start of execution if you prefer) of instruction I0 in warp 0. Suppose `W0I0` is issued at cycle 0. Show the times that the next three instructions in warp 0 will be issued. **Please check the code for dependencies!**

(*b*) Based on your answer above, determine how many warps are needed to cover latency.

Problem 5: [14 pts] In Fermi, as described by the V1.1 whitepaper, the number of threads per multiprocessor has "increased" from 1024 (32 warps) to 1536 (48 warps).

(*a*) Why could that be considered a decrease from the point of view of hiding latency?

(*b*) How many threads would Fermi need per multiprocessor so that it could hide the same amount of latency as the gt200 can hide?

Problem 6: [14 pts] The code below executes on a Larrabee-like processor. Each iteration reads 64 bytes of memory, does a computation, and stores the result. A single Larrabee thread will simultaneously execute 16 instances of the code below, one per lane. (The organization in last year's problem 3 was different.)

Without cache misses, an iteration takes 10 cycles. Memory latency is 400 cycles.

```
void llane_one_of16(int my_start, int my_stop) {

  const int DISTANCE =     ;// Fill in.

  for ( int i=my_start; i<my_stop; i++ )
    {
      prefetch( &data1[i + DISTANCE ] );
      float16 vec1 = data1[i];  // Load 64 bytes of data.
      results[i] = compute_something(vec1);
    }
}
```

(*a*) Compute the prefetch distance so that data arrives just in time. Setting DISTANCE to 1 will prefetch just one iteration ahead. (In last year's final one would have to set DISTANCE to 16 to prefetch one iteration ahead.)

(*b*) If DISTANCE is too small data will not arrive in time. But if DISTANCE is too high data will arrive, but then be evicted. Given a cache size of 32 kiB what is the maximum DISTANCE for which data will still be there, assuming the entire cache is devoted to prefetched data?

Problem 7: [14 pts] The addition of a programmable vertex shader to the OpenGL API made the fixed-function lighting model unnecessary, it was retained in OpenGL 3.0 (grudgingly it seems) only for compatibility and convenience. The vertex shader can compute a lighted color however it likes, there is no need for the OpenGL standard to specify how.

(*a*) For a similar reason, why haven't fragment shaders made the different texture filtering modes (such as bi-linear) unnecessary. Why not have the fragment shader code fetch and combine texels to produce a filtered texel? Instead, OpenGL specifies a set of filtering modes and the fragment shader is delivered a texel filtered using one of these modes. Shouldn't texture filtering modes be as obsolete as lighting models?

(*b*) Primitives are still part of the OpenGL 3 spec, the only important one is a triangle. Consider a new programmable stage called a *primitive shader*. It could be used to code user-defined primitives, for example, spheres, disks, etc. Like the geometry shader, a primitive shader would read a set of vertices. Its execution would result in the primitive perfectly rendered.

A geometry shader could approximate a primitive shader by emitting triangles that tessellated the desired shape. However, the result would be approximate unless the triangles covered individual pixels, and in that case execution might be too slow.

☐ What parts of the rendering pipeline would a primitive shader replace?

☐ What would be a reasonable output for the primitive shader?