Name ______________________________

EE 7700-1

Take-Home Pre-Final Examination

Wednesday, 29 April 2009 to Early Monday Morning, 4 April 2009

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

Alias ______________________________

Good Luck!

Problem 1: [20 pts]The CUDA code below performs some calculation on each element of an array, `global data`. It operates on a GeForce 8000-like GPU with:

- Eight multiprocessors.
- Core clock frequency of 1.5 GHz.
- Four-hundred cycle global access latency.

The `cuda normal thread` routine consists of 9 instructions including one global load.

```
__host__ void launch(int element_amt_lg)
{
  const int element_amt = 1 << element_amt_lg;
  dim3 block_dim(256,1,1);
  dim3 grid_dim( 1 << element_amt_lg - 8, 1, 1 );
  cuda_normal_thread<<<grid_dim,block_dim>>>(element_amt);
}

__global__ void
cuda_normal_thread(int element_amt)
{
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
  const float coord = global_data[idx];
  output_data[idx] = some_func(transform_matrix,coord);
}
```

(*a*) Determine either the time needed for 2^{20} elements (`work amt lg` is 20), or else determine the computation rate in elements per cycle. (Two ways of expressing the same thing.)

(*b*) The code below may be the true assembly language for `cuda normal thread`. (See `http://www.cs.rug.nl/ wladimier/de`
Based on this code, what would be the minimum number of threads per multiprocessor needed to achieve peak performance? Explain.

```
 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 $r1, $r0, 0x00000002
 4: add.u32 $r0, $r1, c0[0x0040]
 5: mov.u32 $r0, g[$r0]              // <- Global load of data into r0.
 6: add.u32 $r1, $r1, c0[0x0044]
 7: add.rn.f32 $r0, $r0, 0x3f800000
 8: mov.end.u32 g[$r1], $r0
```

Problem 2: [20 pts]The CUDA code below does exactly the same thing as the code from the previous problem, but it does so in an un-cuda-like fashion: looping over elements in a thread rather than spawning one thread per element. The code runs on the system described in the previous problem.

```
__host__ void launch_loop(int element_amt_lg, int thds_per_mp)
{
  const int element_amt = 1 << element_amt_lg;
  const int num_mp = 8; // Number of multiprocessors.
  dim3 block_dim(thds_per_mp,1,1);
  dim3 grid_dim(num_mp,1,1);
  const int iter_per_thread =
    ceil( double(element_amt) / ( thds_per_mp * num_mp ));
  cuda_loop_thread<<<grid_dim,block_dim>>>(iter_per_thread,element_amt);
}

__global__ void
cuda_loop_thread(int iter_per_thread, int element_amt)
{
  const int idx_start =
    ( blockIdx.x * blockDim.x + threadIdx.x ) * iter_per_thread;
  const int idx_stop = idx_start + iter_per_thread;
  for ( int idx = idx_start; idx < idx_stop; idx++ ) {
      const float coord = global_data[idx];    // <-   IGNORE COALESCE PROBLEMS
      output_data[idx] = some_func(transform_matrix,coord);  }
}
```

Here is the assembler for the loop body:

```
label0: mov.u32 $r3, g[$r0]       <- Global load.
add.rn.f32 $r3, $r3, 0x3f800000
add.b32 $r0, $r0, 0x00000004
mov.u32 g[$r1], $r3
set.ne.u32 $p0$o127, $r0, $r2
add.b32 $r1, $r1, 0x00000004
@$p0.ne bra.label label0
```

(*a*) Determine the time needed to process 2^{20} elements (or the computation rate) when `thds per mp` is 1.

(*b*) Determine the time needed to process 2^{20} elements (or the computation rate) when `thds per mp` is 32.

(*c*) Do you think the loop approach in this problem is better or worse than the normal CUDA approach from the previous problem? Explain.

Problem 3: [20 pts]On a system like the GeForce 8000 memory latency can be hidden with lots of threads but that won't work on Larrabee where there are many fewer threads. For comparison purposes treat each Larrabbee thread as 16 GeForce threads, so each Larrabbee core has a total of 64 threads. The Larrabbee equivalent of a warp would be 16 threads.

Since there are too few threads to hide memory latency Larrabbee must rely on prefetching. The code below is the kernel from the previous problem, but with some kind of a prefetch call added. It is to run on a Larrabbee-like implementation and so global accesses can benefit from a cache hit, but for the code below there will only be a cache hit if the prefetch is done correctly.

Note: This paragraph did not appear in the original exam. Assume that the compiler unrolls the loop 16 times assigning each of the 16 loop body copies to a different lane. After unrolling the number of iterations will be `iter\_per\_thread / 16`. The `threadIdx.x` variable below refers to a Larrabbee thread.

(*a*) Finish the code so that `pre idx` is set to an appropriate value for the prefetch. Assume a 400-cycle memory latency. As before ignore coalesce issues. The prefetched data should arrive when and where it is needed, and should not arrive too early (wastes cache space) or too late (results in stall).

```
__global__ void
l_loop_thread(int iter_per_thread, int element_amt)
{
  const int idx_start =
    ( blockIdx.x * blockDim.x + threadIdx.x ) * iter_per_thread;
  const int idx_stop = idx_start + iter_per_thread;
  for ( int idx = idx_start; idx < idx_stop; idx++ )
    {
      const float coord = global_data[idx];
      const int pre_idx =                          ; // <-  PUT ANSWER HERE.

      prefetch(&global_data[pre_idx]);
      output_data[idx] = some_func(transform_matrix,coord);
    }
}
```

(*b*) Once the problem above is understood it should not have been too hard to determine a prefetch address (or array index). The general case of inserting prefetch instructions in ordinary CPU code is much harder, often prefetches wastefully bring in data that is never used.

A Larrabbee critic might say that since prefetching is hard Larrabbee will be slowed down by cache misses (because the prefetch was wrong) and so the GeForce 8000 approach of many threads to hide latency is better. Explain the fallacy in this argument. *Hint: Consider how addresses are computed in GeForce (cuda) threads.*

Problem 4: [20 pts]The familiar CUDA code below will probably be slowed down because of inefficient global memory access. Re-write the code to fix the problem. The code runs on a GeForce 8000-like system. (Refer to the CUDA Programming guide for help.)

```
__global__ void
cuda_loop_thread(int iter_per_thread, int element_amt)
{
  const int idx_start =
    ( blockIdx.x * blockDim.x + threadIdx.x ) * iter_per_thread;
  const int idx_stop = idx_start + iter_per_thread;
  for ( int idx = idx_start; idx < idx_stop; idx++ )
    {
      const float coord = global_data[idx];
      output_data[idx] = some_func(transform_matrix,coord);
    }
}
```

Problem 5: [20 pts]The code below sums values provided by all threads of a block. The `block lg` argument is the log base 2 of the block size, `shared array` is a pointer to shared memory, `my value` is the thread's value to sum, and `all` determines whether all callers get the sum, or just thread 0. (This is a simplified version of the code in https://svn.ece.lsu.edu/svn/gp/gpgpu/balloon-kernel.cu.)

```
__device__ float
reduce(int block_lg, float *shared_array, float my_value, bool all)
{
  const int tid = threadIdx.x;
  const int block_lg_h = block_lg >> 1;
  const int block_lg_l = block_lg - block_lg_h;
  const int upper_size = 1 << block_lg_h;
  const int lower_size = 1 << block_lg_l;

  float vol_sum = shared_array[tid] = my_value;
  __syncthreads();

  // Round 1
  //
  if ( tid < lower_size )
    {
      for ( int i=1; i<upper_size; i++ )
        vol_sum += shared_array[ (i << block_lg_l ) + tid ];
      shared_array[tid] = vol_sum;
    }

  // Round 2
  //
  if ( lower_size > warpSize ) __syncthreads();

  if ( tid == 0 )
    for ( int i=1; i<lower_size; i++ ) vol_sum += shared_array[i];

  if ( !all ) return vol_sum;
  if ( tid == 0 ) shared_array[0] = vol_sum;
  __syncthreads();
  return shared_array[0];
}
```

Questions on next page.

Problem 5, continued:

(*a*) How much divergence is caused by the `if` statement right after the "Round 2" comment. (Easy)

☐ How much? Explain.

(*b*) Explain why the `syncthreads` call just after the "Round 2" comment is definitely needed when `lower size` is greater than the warp size.

☐ Why needed?

☐ Show an example in which execution would be incorrect without it.

(*c*) Explain why the `syncthreads` call might not be needed when `lower size` is not larger than the warp size.

☐ Why not needed?

☐ Explain why execution is correct with example from previous part.