Name ______________________

EE 7700-2

Take-Home Pre-Final Examination

Monday, 22 April 2013 to Monday, 29 April 2013

Work on this exam alone. Regular class resources, such as notes, papers, documentation, and code, can be used to find solutions. Do not discuss this exam with classmates or anyone else, except questions or concerns about problems should be directed to Dr. Koppelman.

Problem 1	______	(52 pts)
Problem 2	______	(12 pts)
Problem 3	______	(12 pts)
Problem 4	______	(12 pts)
Problem 5	______	(12 pts)
Exam Total	______	(100 pts)

Alias ______________________

Good Luck!

Problem 1: [52 pts] This problem is based on the spring code shown in class, which itself is based on a solution to GPU Programing (EE 4702 Fall 2012) Homework 5. This problem asks about a simplified version of a routine used in that solution, the code appears below. One does not need to understand how the entire spring program works to solve this problem.

The spring demo code models a spring by dividing it into N segments (a segment is a short piece of the spring). For a spring of a given size choosing a larger value of N will yield a more accurate and more time-consuming simulation.

The code below, based on the spring demo code, tests for interpenetration (intersection) between all possible N^2 pairs of segments (including self pairs). The code first calls `find_interpenetration` to perform computationally inexpensive interpenetration test, if this passes (meaning there is possibly an intersection) `resolve_interpenetration` is called to determine the forces needed to separate the objects. These forces are added to shared array `force` and are used after the loop (not shown) to update the velocity. What happens after the loop is not part of this problem.

Assume that the only segment data needed by this code is the position of a helix, in the array `helix_position`. Each element of `helix_position` is a `float4`. Routines `find_interpenetration` and `resolve_interpenetration` themselves do not read anything from global memory.

- A loop iteration consists of 20 instructions if the test fails and 52 instructions if the test passes.
- The kernel will be launched in a configuration in which the number of blocks is a multiple of N.
- Assume that on average each segment interpenetrates four other segments.
- Let N denote the number of segments (`hi.phys_helix_segments`) in the code, T the number of threads per block, e the number of "a" elements processed per block (`a_per_block` in the code), and let C denote the expected number of instructions per iteration.
- Typical Run: $N = 1280$ and $T = 640$.
- Typical System: NVidia GPU of compute capability 2.0, with 10 multiprocessors, a CUDA core clock frequency of 1.5 GHz, and an off-chip data transfer rate of 200 GB/s.

Continued on Next Page

```
__global__ void time_step_intersect_simple() {
  // Number of a segments handled per block.   Symbol: e
  int a_per_block = hi.phys_helix_segments / gridDim.x;

  // "a" segment operated on by thread 0 in block.
  int a_idx_block = blockIdx.x * a_per_block;

  int a_local_idx = threadIdx.x % a_per_block;

  // "a" segment operated on by this thread.
  int a_idx = a_idx_block + a_local_idx;

  // First "b" segment read by this thread.
  int b_idx_start = threadIdx.x / a_per_block;
  int stride = blockDim.x / a_per_block;

  // Get position of "a" segment.
  float4 a_position = helix_position[a_idx];

  __shared__ pVect __restrict__ force[1024]; // Only need a_per_block elts.

  if ( threadIdx.x < a_per_block ) force[a_local_idx] = mv(0,0,0);

  __syncthreads();

  for ( int j=b_idx_start; j<hi.phys_helix_segments; j += stride )
    {
      float4 b_position = helix_position[j];

      bool intersect = find_interpenetration(a_position,b_position);

      if ( !intersect ) continue;

      float3 f = resolve_interpenetration(a_position,b_position);

      atomicAdd(&force[a_local_idx].x,f.x);
      atomicAdd(&force[a_local_idx].y,f.y);
      atomicAdd(&force[a_local_idx].z,f.z);
    }

  // Wait for all threads to finish.
  __syncthreads();

  // Additional code here, ignore for this problem.

}
```

(*a*) The minimum value for e is 1, this corresponds to the solution to Problem 2 of the original homework assignment. What is the maximum possible (not necessarily the best) value for e based on the given code above? Explain why the code won't run correctly with larger values of e.

(*b*) Determine the amount of data read **per block** into the multiprocessor due to the code shown. Give your answer in terms of the symbols above. State any assumptions made about cache behavior.

(*c*) Using your answer from the previous part derive the computation to communication ratio assuming that the intersection test always fails.

(d) Using your answers from previous parts, characterize the performance of the code over a range of values of e from 1 to its maximum value. Show a graph with the x axis ranging from 1 to the maximum value of e. Plot performance-related quantities on the y axis. Label the point or range of values on the x axis for which best performance is expected. Over other ranges indicate what is limiting performance. Also show areas in which execution is computation bound and where it is communication bound.

☐ Show point or range yielding best performance.

☐ Identify (and label) cause of sub-optimal performance in other regions.

☐ Show compute-bound region (not necessarily sub-optimal performance).

☐ Show execution-bound region (not necessarily sub-optimal performance).

(*e*) Suppose $N = 16000$ (larger than the typical system). Find the smallest value of e for which the L1 cache does something useful.

(*f*) How might the $L2$ cache when $N = 16000$?

(*g*) The number of instructions executed in the loop body may be 20 or 52, depending on whether there is interpenetration. In the previous parts a 20-instruction loop body was assumed. In this part we will consider both possibilities.

Assume that on average each segment penetrates four other segments. What should be the number of instructions per iteration used to analyze the code? *Hint: It's not* $20\frac{N-4}{N} + 52\frac{4}{N}$.

Problem 2: [12 pts] With the design of a CC 2.0 NVidia processor as a starting point, consider and comment on the changes below.

(*a*) Suppose the warp size were increased from 32 to 64.

☐ Describe the characteristic of code that would be helped or hurt by this change.

☐ Explain how this would impact the cost. Would their be less or more of something? Could something be made slower?

Problem 3: [12 pts] Suppose a CC 2.0 multiprocessor, instead of having one pipeline for odd warps and one pipeline for even warps, had a just single pipeline for use by any warp. There would still be 32 CUDA cores.

(*a*) Describe the characteristics of kernel code and a launch configuration for which the proposed single pipeline system would be faster. *Hint: A 100% correct answer must say something about both the launch configuration and kernel code characteristics.*

(*b*) What is the difference in the number of warps needed to hide latency?

Problem 4: [12 pts] Consider the code fragment below, for execution on a Phi. Describe the big difference in the resulting code between the following two cases: (1) stride is equal to 32 and this is known at compile time or (2) stride is not known at compile time. Note that the compiler is going to use vector instructions such that each vector instruction handles 16 iterations worth of computations. Consider single-core execution.

```
for ( int idx = 0; idx < idx_stop; idx++ )
  {
    a = array[idx/stride];
    x = another_array[a];
    aout[idx] = x*x;
  }
```

Problem 5: [12 pts] The code below performs a vertex transformation, similar to the code used in some homework assignments, but the size of the vertex can vary from 2 to 4. The input and output array elements are four elements, a second array, `vertex_size` gives the size of each element.

```
for ( int h=tid;  h<array_size;  h += num_threads )
  {
    Vertex p = d_app.d_v_in_f4[h];
    Vertex q = zero;
    int N = vertex_size[h];

    // Perform the transformation.
    for ( int i=0; i<N; i++ )
      for ( int j=0; j<N; j++ ) q.a[i] += d_app.matrix[i][j] * p.a[j];

    d_app.d_v_out[h] = q;
  }
```

(*a*) Why might the code run much less efficiently than it can? Consider reasons other than wasted space (using four-elements to store 2-element vectors).

(*b*) Fix the code so that it runs efficiently (but still of course handles vertices of varying size). Do **not** change the data layout.