CS6963: Parallel Programming for GPUs
Midterm Exam
March 25, 2009

Instructions:
This is an in-class, open-note exam. Please use the paper provided to submit your responses. You can include additional paper if needed. The goal of the exam is to reinforce your understanding of issues we have studied in class.

CS6963: Parallel Programming for GPUs
Midterm Exam
March 25, 2009

I. Definitions (16 points)

Provide a very brief definition of the following terms:

a. Amdahl's Law

b. Data locality

c. Warp

d. Memory access coalescing

e. Bank conflict

f. SIMD

g. Block

h. Divergent branch

II. Constraints (4 points)

List a specific constraint on either parallelism (threads, blocks, dimensionality of each) or memory capacity (for one specific part of the GPU memory hierarchy), and in one sentence, describe how this impacts your GPU program as compared to a sequential CPU program.

III. Problem Solving (80 points)

In this set of five questions, you will be asked to provide code solutions to solve particular problems. This portion of the exam may take too much time if you write out the CUDA solution in detail. I will accept responses that sketch the solution, without necessarily writing out the code or worrying about correct syntax. Just be sure you have conveyed the intent and issues you are addressing in your solution.

a. Derive dependence distance vectors (including input dependences) for the following sequential code. Given these distance vectors, how might you perform transformations and generate CUDA code that makes use of constant memory.

```
float a[1024][1024], b[1024];

for (j=0; j<1024; j++)
   for (i=0; i<1024; i++)
       b[j] += 0.25 * (a[i][j-2] + a[i][j-1] + a[i][j+1] + a[i][j+2]);
```

b. Given the following sequential code, sketch out a CUDA implementation. Derive a partitioning into threads and blocks that does not exceed various hardware limits. Assume all data is stored in global memory.

```
…
float a[1024][1024], b[1024];

for (i=0; i<1024; i++)
   for (j=0; j<1024-i; j++)
      b[i+j] += arbitrary_function(a[i][j]);
```

c. For your response in b above, indicate whether global memory accesses will be coalesced and whether there will be bank conflicts in shared memory. Explain why or why not. (Note that if your response for b is incorrect, this will not count against your answer for this one. If you have skipped b, then use another code example to respond to c).

d. Given the following CUDA code, describe how you would modify this to derive an optimized version that will have fewer divergent branches. The functions *even_kernel* and *odd_kernel* compute b from a in different ways. (Note: '%' here is the standard C mod operator, so the conditional is testing whether the threadIdx is divisible by 2).

```
Main() {
    float h_a[1024], h_b[1024];
    …
    /* assume appropriate cudaMalloc called to create d_a and d_b, and d_a is */
    /* initialized from h_a using appropriate call to cudaMemcpy */
    dim3 dimblock(256);
    dim3 dimgrid(4);
    compute<<<dimgrid, dimblock,0>>>(d_a,d_b);
    /* assume d_b is copied back from the device using call to cudaMemcpy */
}

__global__ compute (float *a, float *b) {
if (threadIdx.x % 2 == 0)
   (void) even_kernel (a, b);
else /* (threadIdx.x % 2 == 1) */
   (void) odd_kernel (a, b);
}
```

e. Given the following CUDA code, add synchronization to derive a correct implementation that has no race conditions. (Hint: You should be able to simply insert __synchthreads() calls without modifying the code.)

```
__global__ compute (float *a, float *b, int BLOCKSIZE) {
   __shared__ s_a[128], s_b[128];
   /* copy portion of input data into shared memory */
   s_a[threadIdx.x] = a[blockIdx.x*BLOCKSIZE + threadIdx.x];

   /* Time step loop */
    for (int t = 0; t<MAX_TIME; t++) {
       /* alternate inputs and outputs on even/odd time steps */
       if (t % 2 == 0) {
          int boundary = min((blockIdx.x+1)*BLOCKSIZE-1,
                             blockDim.x*BLOCKSIZE-1,threadIdx+2);
          s_b[threadIdx.x] = s_a[threadIdx.x] + s_a[boundary];
       }
       else /* (t%2 == 1) */ {
          int boundary = min((blockIdx.x+1)*BLOCKSIZE-1,
                             blockDim.x*BLOCKSIZE-1,threadIdx+2);
          s_a[threadIdx.x] = s_b[threadIdx.x] + s_b[boundary];
      }
  }

/* Result is in s_b, and must be copied to b */
b[blockIdx.x*BLOCKSIZE + threadIdx.x] = s_b[threadIdx.x];

}
```

Extra Credit: (Brief) Essay Question (10 points)
Pick one of the following four topics and write a very brief essay about it, no more than 3 sentences.

a. Describe the features of computations that are likely to obtain high speedup on a GPU as compared to a sequential CPU.
b. Explain how CUDA threads and blocks are mapped to the GPU and scheduled for execution.
c. Consider the architecture of the current GPUs and impact on programmability. If you could change one aspect of the architecture to simplify programming, what would it be and why? (You don't have to propose an alternative architecture.)
d. Suppose you could sponsor development of a tool for GPUs --- either for constructing programs, debugging or performance tuning --- that would make it easier to develop GPU programs. What would it do for you that is really hard to do now? (You don't have to imagine how to build such a tool.)