CS6235: Parallel Programming for GPUs
Midterm Exam
March 28, 2012

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.

CS6235: Parallel Programming for GPUs
Midterm Exam
March 28, 2012

I. Definitions (10 points)

Provide a very brief definition of the following terms:

a. Spatial locality

b. Atomic operation

c. Scoreboarding

d. Memory bandwidth

e. Memory fence

II. Short answer (20 points)

Provide a very brief answer to the following four questions.

a. Describe one mechanism we discussed for eliminating shared memory bank conflicts in code that exhibits these bank conflicts. Since the occurrence of bank conflicts depends on the data access patterns, please explain any assumptions you are making about the original code with bank conflicts.

b. Give one example of a synchronization mechanism that is *control-based*, meaning it controls thread execution, and one that is *memory-based*, meaning that it protects race conditions on memory locations.

c. Briefly compare double precision floating point support in Fermi vs. previous generation GPUs.

d. What happens if two blocks assigned to the same streaming multiprocessor each use more than half of either registers or shared memory? How does this affect scheduling of warps? By comparison, what if the total register and shared memory usage fits within the capacity of these resources?

III. Problem Solving (60 points)

In this set of three questions, you will be asked to provide code solutions or explanations related to 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. **Parallelization and data placement in the memory hierarchy**

Given a sparse matrix-vector multiplication, consider two different GPU implementations: (a) one in which the sparse matrix is stored in compressed sparse row format (see below code); and, (b) one which uses an ELL format for the sparse matrix because the upper limit on number of nonzeros per row is fixed. For (a), describe the mapping of this code to GPUs. How is it decomposed into threads, and where are t, data, indices and x placed in the memory hierarchy? For (b), how does the ELL representation affect your solution? Would you consider a different thread decomposition and data placement?

```
// Sequential sparse matrix vector multiplication using compressed sparse row
// CSR) representation of sparse matrix. "data" holds the nonzeros in the sparse
// matrix

for (j=0; j<nr; j++) {
   for (k = ptr[j]; k<ptr[j+1]; k++)
     t[j] = t[j] + data[k] * x[indices[k]];
```

b. **Tiling for the memory hierarchy**

The following sequential computation has significant data reuse that can be exploited on the GPU. For each data structure, analyze how to place data in the memory hierarchy. (a) How would you parallelize this computation? (b) What modifications do you need to make to the code to improve its memory hierarchy and parallelization? (c) Where would you place a, b and c in the memory hierarchy and why? If there are multiple options, describe each one you considered.

```
for (i=1; i<n; i++)
   for (j=1; j<n; j++)
      a[j][i] = a[j][i-1] + b[j][i] + c[i];
```

c. **Divergent branches**

Consider the code below, which potentially includes divergent branches. Describe if there is a way to rewrite the code to eliminate these. Describe in general terms what divergent branches can be eliminated by code rewriting versus cannot be eliminated.

```
Main() {
    float h_a[1024], h_b[1024], h_data[1024];
    …
    dim3 dimblock(128);
    dim3 dimgrid(8);
    compute<<<dimgrid, dimblock,0>>>(d_a,d_b,d_data);
    /* assume d_b is copied back from the device using call to cudaMemcpy */
}

__global__ compute (float *a, float *b, float *data) {

if (threadIdx.x % 2 == 0)
   (void) even_kernel (a, b);
else /* (threadIdx.x % 2 == 1) */
   (void) odd_kernel (a, b);
if (data[threadIdx.x] > threshold)
   (void) large_kernel (data, a);
else
   (void) small_kernel (data, b);
}
```

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

a. Describe how a __synchthreads() construct affects warp scheduling, as discussed in the last class. What is the difference between scheduling a kernel function with and without a synchronization between two statements?
b. We examined a sorting algorithm on GPUs, and walked through the algorithm. What were the key features of that algorithm, and how does it differ from sequential and parallel sorting algorithms for conventional architectures? In other words, how does the GPU architecture affect the solution?
c. Describe three optimizations that were performed for the MRI application case study.