CS6963: Parallel Programming for GPUs
Midterm Exam
April 5, 2010

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. Parallel programming model

b. Data reuse

c. Atomic operation

d. Memory access coalescing

e. Race condition

f. SPMD

g. Barrier synchronization

h. Hardware execution model

II. SIMT Execution of a warp (4 points)

Select ONE of the following aspects of the NVIDIA architecture warp execution and describe briefly (in a couple sentences) how it works:

(a) selecting a warp to be scheduled for execution;
(b) executing a branch operation in a warp;
(c) scheduling global memory accesses for a warp;
(d) allocating registers to a thread.

III. Problem Solving (80 points)

In this set of four 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. **Managing memory bandwidth**
Given the following CUDA code, how would you rewrite to improve bandwidth to global memory and, if applicable, shared memory? Explain your answer for partial credit. Assume c is stored in row-major order, so c[i][j] is adjacent to c[i][j+1].

```
N = 512;
NUMBLOCKS = 512/64;

float a[512], b[512], c[512][512];

__global compute(float a, float *b, float *c) {
int tx = threadIdx.x;
int bx = blockIdx.x;

for (j = bx*64; j< bx+64; j++)
   a[tx] = a[tx] - c[tx][j] * b[j];
}
```

b. **Divergent Branch**

Given the following CUDA code, describe how you would modify this to derive an optimized version that will have fewer divergent branches.

The functions *starting_kernel* and *default_kernel* compute b from a in different ways. (Note: '%' here is the standard C mod operator, so the conditional is testing whether the threadIdx.x is divisible by 16).

```
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) {
float a[4][256], b[4][256];
int tx = threadIdx.x;
if (tx % 16 == 0)
   (void) starting_kernel (a[bx][tx], b[bx][tx]);
else /* (tx % 16 > 0) */
   (void) default_kernel (a[bx][tx], b[bx][tx]);
}
```

c. **Tiling**
The following sequential image correlation computation compares a region of an image to a template. Show how you would tile the image and threshold data to fit in 128MB global memory and the template data to fit in a 16KB shared memory? Explain your answer for partial credit.

TEMPLATE_NROWS = TEMPLATE_NCOLS = 64;
IMAGE_NROWS = IMAGE_NCOLS = 5192;

int image[IMAGE_NROWS][IMAGE_NCOLS], th[IMAGE_NROWS][IMAGE_NCOLS];
int template[TEMPLATE_NROWS][TEMPLATE_NCOLS];

```
for(m = 0; m < IMAGE_NROWS - TEMPLATE_NROWS + 1; m++){
    for(n = 0; n < IMAGE_NCOLS - TEMPLATE_NCOLS + 1; n++){
      for(i=0; i < TEMPLATE_NROWS; i++){
        for(j=0; j < TEMPLATE_NCOLS; j++){
          if(abs(image[i+m][j+n] – template[i][j]) < threshold)
            th[m][n]+= image[i+m][j+n]
        }
      }
   }
  }
```

d. **Parallel partitioning and synchronization (LU Decomposition)**
Without writing out the CUDA code, consider a CUDA mapping of the LU Decomposition sequential code below. Answer should be in three parts, providing opportunities for partial credit: (i) where are the data dependences in this computation? (ii) how would you partition the computation across threads and blocks? (iii) how would you add synchronization to avoid race conditions?

```
float a[1024][1024];

for (k=0; k<1023; k++) {
   for (i=k+1; i<1024; i++)
      a[i][k] = a[i][k] / a[k][k];
   for (i=k+1; i<1024; i++)
      for (j=k+1; j<1024; j++)
         a[i][j] = a[i][j] – a[i][k]*a[k][j];
}
```

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.)