ECE408/CS483 Practice Exam #2, Fall 2012

- You are allowed to use any notes, books, papers, and other reference material as you desire. No electronic assistance other than a calculator is permitted.
- No interactions with humans other than course staff are allowed.
- This exam is designed to take 80 minutes to complete.
- This exam is based on lectures as well as lab MPs/projects.
- The questions are randomly selected from the topics we covered this semester.
- You can write down the reasoning behind your answers for possible partial credit.
- **Good luck!**

Question 1 (50 points): Short Answer

Answer each of the following questions in as few words as you can. Your answer will be graded based on completeness, correctness, and conciseness.

1. (7 points) Give two reasons for adding extra "padding" elements to arrays allocated in GPU global memory?

Answers:

1) To achieve coalesced global memory operations by ensuring all accesses in a Warp always form good coalescing groups (aligned to 64B boundaries)..
2) To eliminate code complexity and branch divergence handling cases when the thread blocks don't evenly divide into the size of the dataset.

2. (7 points) Give two potential disadvantages associated with increasing the amount of work done in each CUDA thread, such as loop unrolling techniques, using fewer threads in total?

Answers:

1) With more resource usage in each enlarged thread block, the number of thread blocks that can be simultaneously executed by each SM can decrease, potentially reducing the total number of resident threads in each SM.
2) Fewer thread blocks per kernel invocation, may reduce the ability of the algorithm to scale to future GPUs
3) The decrease in total thread blocks results in a higher penalty for cases where the number of SMs doesn't evenly divide into the number of thread blocks as the last group of thread blocks completes their work (load imbalance penalty).

3. (8 points) The following CUDA code is intended to perform a sum of all elements of the **`partialSum`** array. First, provide the missing operation, and second estimate the fraction of the iterations of the **`for`** loop that will have branch divergence.

```
__shared__ float partialSum[];
unsigned int tid = threadIdx.x;

for (unsigned int stride = blockDim.x;
     stride > 0;
     stride = stride >> 1)
{
     _________________________ ;; missing operation
     if (tid < stride)
         partialSum[tid] += partialSum[tid+stride];
}
```

Answer: Yes, when stride is less than 32. The loop will iterate 9 times. The last 5 times will have branch divergence. 5/9 of the iterations will have branch divergence.

(There can only be up to 512 threads in a block, thus stride cannot be more than 512 in this code.)

4. (7 points) For the following code, estimate the average run time in cycles of a thread. Assume for simplicity that the **__sin()** function operates on radians, and that all operations including the **__sin()** take 1 cycle.

```
if (__sin( (float) threadIdx.x) > 0.5)
{
      :
      A operations
      :
}
else
{
      :
      B operations
      :
}
```

Answer: (#operations executing the if [anywhere from 2-5 depending on ISA details]) + A + B

5. (7 points) Provide two advantages of a programmer-managed memory, such as the CUDA shared memory, over a hardware managed cache?

Answer: simpler hardware structures, and better utilization due to tighter control.

6. (7 points) Assuming capacity were not an issue for registers or shared memory, give one case where it would be valuable to use shared memory instead of registers to hold values fetched from global memory?

Answer: threads can communicate read values with each other, saving bandwidth through read-sharing or better coalescing.

7. (7 points) Assume the following code will execute all in the same warp of a CUDA thread block. Write a piece of CUDA code in which the threads within the warp reverse an array **list** of length 32 in shared memory. Make your code as efficient in time and space as possible.

```
__shared__ float list[32]
```

list[threadIdx % 32] = list[32 – (threadIdx % 32) - 1];

Question 2 (25 points): CUDA Basics

The following code computes 1024 dot products, each of which is calculated from a pair of 256-element sub-vectors. Assume that the code is executed on G80. Use the code to answer the following questions.

```
1    #define VECTOR_N  1024
2    #define ELEMENT_N 256
3    const int DATA_N       = VECTOR_N * ELEMENT_N;
4    const int DATA_SZ      = DATA_N * sizeof(float);
5    const int RESULT_SZ    = VECTOR_N  * sizeof(float);
...
6    float *d_A, *d_B, *d_C;
...
7    cudaMalloc((void **)&d_A, DATA_SZ);
8    cudaMalloc((void **)&d_B, DATA_SZ);
9    cudaMalloc((void **)&d_C, RESULT_SZ);
...
10   scalarProd<<<VECTOR_N, ELEMENT_N>>>(d_C, d_A, d_B, ELEMENT_N);
11
12   __global__ void
13   scalarProd(float *d_C, float *d_A, float *d_B, int ElementN)
14   {
15       __shared__ float accumResult[ELEMENT_N];
16       //Current vectors bases
17       float *A = d_A + ElementN * blockIdx.x;
18       float *B = d_B + ElementN * blockIdx.x;
19       int tx = threadIdx.x;
20
21       accumResult[tx] = A[tx] * B[tx];
22
23       for(int stride = ElementN / 2; stride > 0; stride >>= 1)
24       {
25           __syncthreads();
26           if(tx < stride)
27               accumResult[tx] += accumResult[stride + tx];
28       }
30       d_C[blockIdx.x] = accumResult[0];
31   }
```

1. (2 pts) How many threads are there in total?
   Answer: 1024*256

2. (2 pts) How many threads are there in a Warp?
   Answer: 32

3. (2 pts) How many threads are there in a Block?
   Answer: 256

4. (2 pts) How many global memory loads and stores are done for each thread?
   Answer: 2 loads, 1 store.

5. (2 pts) How many accesses to shared memory are done for each block? Count one access

from all threads as *blockdim.x* accesses.

Answer: 256 for the initial store, (128+64+32+16+8+4+2+1)*3 = 255*3 combined loads and stores for the tree reduction, and 256 loads for the final result.

6. (3 pts) List the source code lines, if any, that cause shared memory bank conflicts.

Answer: There are no bank conflicts.

7. (3 pts) How many iterations of the *for* loop (line 23) will have branch divergence? Show your derivation.

Answer: 5

8. (3 pts) What is the largest and smallest ratios of floating point arithmetic to global memory access in each thread?

Answer: (1+9)/3, (1+1)/3

9. (3 pts) Identify an opportunity to significantly reduce the bandwidth requirement on the global memory. How would you achieve this? How many accesses can you eliminate?

Answer: Line 30 stores an identical value to global memory 256 times to the same address, which causes massive performance losses in the memory system performance. The simple solution is to add a conditional so that only thread 0 will do so. We can eliminate 1024*255 stores to the global memory.

10. (3 pts) How many registers will be needed to accommodate variable A in each block?

Answer: 256.

Question 3 (25 points). The following kernel is executed on a large matrix, which is tiled into submatrices. To manipulate tiles, a novice CUDA programmer has written the following device kernel to transpose each tile in the matrix. The tiles are of size BLOCK_SIZE by BLOCK_SIZE, and each of the dimensions of matrix A is known to be a multiple of BLOCK_SIZE. The kernel invocation and code are shown below. BLOCK_SIZE is known at compile-time, but could be set anywhere from 1 to 20.

```
dim3 blockDim(BLOCK_SIZE,BLOCK_SIZE);
dim3 gridDim(A_width/blockDim.x,A_height/blockDim.y);
BlockTranspose<<<gridDim, blockDim>>>(A, A_width, A_height);

__global__ void
BlockTranspose(float* A_elements, int A_width, int A_height)
{
  __shared__ float blockA[BLOCK_SIZE][BLOCK_SIZE];

  int baseIdx = blockIdx.x * BLOCK_SIZE + threadIdx.x;
  baseIdx += (blockIdx.y * BLOCK_SIZE + threadIdx.y) * A_width;

  blockA[threadIdx.y][threadIdx.x] = A_elements[baseIdx];

  A_elements[baseIdx] = blockA[threadIdx.x][threadIdx.y];
}
```

1. (7 points) Out of the possible range of values for BLOCK_SIZE, for what values of BLOCK_SIZE will this kernel function correctly when executing on a current device?

Answer: BLOCK_SIZE = 1 through 5. Since there is no synchronization, the whole block must be within one warp in order to function correctly.

2. (6 points) If the code does not execute correctly for all BLOCK_SIZE values, suggest a fix to the code to make it work for all BLOCK_SIZE values.

Answer: Add a __synchthreads() call between the write and the read to blockA[].

3. (6 points) Out of the possible range of values for BLOCK_SIZE, for what values of BLOCK_SIZE will the kernel completely avoid uncoalesced accesses to global memory?

Answer: BLOCK_SIZE = 16 only

4. (6 points) For what value of BLOCK_SIZE (from 1 to 20) will the kernel experience the most severe shared-memory conflicts?

Answer: BLOCK_SIZE = 16, where you have 16-way bank conflicts in the load from shared memory.