Name: ____________________________

Directions: MAKE SURE TO COPY YOUR ANSWERS TO A SEPARATE SHEET FOR SENDING ME AN ELECTRONIC COPY LATER.

1. (90) The CUDA code below solves a problem similar to the root finding example in Section 4.11. It finds the root of a user-supplied function **f()**, which is increasing on (0,1) and has a root somewhere inside. The initial search interval is (0,1), but the interval gets smaller with each iteration. At any iteration, the current interval is divided in subintervals, with each thread handling one subinterval. Fill in the blanks.

```
#include <stdlib.h>
#include <stdio.h>
#include <cuda.h>

// f() defined on (0,1), strictly increasing,
// with a root somewhere inside

__device__ float f(float x) {
   return x*x - 0.5;
}

#define BLOCKSIZE 192

__global__ void check1tile(float *devab)
{
   int threadnum =
          blockIdx.x * BLOCKSIZE + threadIdx.x,
       totthreads = gridDim.x * BLOCKSIZE;
   float a = devab[0];
   float b = devab[1];
   float xinc = (b-a) / ______; // blank (a)
   float x = __________;  // blank (b)
   if (f(x) < 0)
      if (f(x+xinc) > 0) {
         __________;  // blank (c)
         __________;  // blank (d)
      }
}

int main(int argc, char **argv)
{   int i;
    float hosab[2] = {0.0,1.0},
          *devab;
    int niters = atoi(argv[1]);
    int nblocks = atoi(argv[2]);
    int float2 = 2 * sizeof(float);
    cudaMalloc((void **)&devab,float2);
    __________________;  // blank (e)
    for (i = 0; i < niters; i++) {
       dim3 dimGrid(nblocks,1);
       dim3 dimBlock(BLOCKSIZE,1,1);
       check1tile<<<dimGrid,dimBlock>>>(devab);
       ______________________;  // blank (f)
    }
    ______________________;  // blank (g)
    for(int i = 0; i < 2; i++)
       printf("%f\n",hosab[i]);
    cudaFree(devab);
}
```

2. (10) As a measure of thread divergence, we might consider *utilization*, meaning the proportion of time threads are actively executing an instruction.

As a toy example, say we have 2 blocks of 4 threads each, executing 10 instruction cycles. That's a possible 80 instruction executions. But suppose 1 of the threads is idle during 6 cycles and another is idle during 15 cycles. Then our utilization would be (80 - 6 - 15)/80 or about 86%.

Consider the code in Problem 1, slightly modified:

```
s = f(x); t = f(x+xinc);
if (s < 0)
   if (t > 0) {
      __________;  // blank (c)
      __________;  // blank (d)
   }
```

Suppose for simplicity that each of the 4 lines starting with the first **if** compiles to one machine instruction. With the specific **f()** used above, find the approximate utilization for those 4 instructions during the first iteration. **Your answer must be an R expression.** Note: Your answer should not depend on the number of threads.

Solutions:

```
#include <stdlib.h>
#include <stdio.h>
#include <cuda.h>

// f() defined on (0,1), strictly increasing,
// with a root somewhere inside

__device__ float f(float x) {
   return x*x - 0.5;
}

#define BLOCKSIZE 192

__global__ void check1tile(float *devab)
{
   int threadnum = blockIdx.x * BLOCKSIZE + threadIdx.x,
       totthreads = gridDim.x * BLOCKSIZE;
   float a = devab[0];
   float b = devab[1];
   float xinc = (b-a) / totthreads;
   float x = a + threadnum * xinc;
   if (f(x) < 0)
      if (f(x+xinc) > 0) {
         devab[0] = x;
         devab[1] = x + xinc;
      }
}

int main(int argc, char **argv)
{   int i;
    float hosab[2] = {0.0,1.0},
          *devab;
    int niters = atoi(argv[1]);
    int nblocks = atoi(argv[2]);
    int float2 = 2 * sizeof(float);
    cudaMalloc((void **)&devab, float2);
    cudaMemcpy(devab, hosab, float2, cudaMemcpyHostToDevice);
    for (i = 0; i < niters; i++) {
       dim3 dimGrid(nblocks,1);
       dim3 dimBlock(BLOCKSIZE,1,1);
       check1tile<<<dimGrid,dimBlock>>>(devab);
       cudaThreadSynchronize();
    }
    cudaMemcpy(hosab, devab, float2, cudaMemcpyDeviceToHost);
    for(int i = 0; i < 2; i++) printf("%f\n",hosab[i]);
    cudaFree(devab);
}
```

2. All threads will execute the first **if**. The threads having a subinterval to the left of $\sqrt{0.5}$ will also execute the second **if**. With n threads, that means that there will be about $n + 2 \times \sqrt{0.5} \times n$ instructions executions during $4n$ opportunities, for a utilization of **(1+2*sqrt(0.5))/4**.