ECE459 Final W2011

Question 1

a) Not covered.
b) Throughput
c) Memory accesses
d) CUDA equivalent: work items
e) Problem size
f) Array aliasing
g) Minimal performance impact when not being actively used
h) Safety and reliability
i) Stores
j) Not covered.
k) Not covered.
l) Compile sources with prof-gen option, run instrumented executable, compile with prof-use option
m) Not covered.
n) Mutual exclusion
o) Not covered. Register renaming
p) Profile
q) Parallelization
r) Acquire the same lock multiple times without deadlock
s) Client-server
t) Kernel

Question 2

Not covered.

Question 3

Not covered.

Question 4

Since the architecture is not sequentially consistent, operations may be reordered.

Final values	Execution order
r1=0, r2=1, x=1, y=1	x=1; r1=y; y=x; r2=x (T1; T2) (can swap first 2 ops for same result)
r1=0, r2=0, x=1, y=0	y=x; r2=x; x=1; r1=y (T2; T1) (can swap last 2 or first 2 ops for same result)
r1=1, r2=1, x=1, y=1	x=1; y=x; r1=y; r2=x (can swap last 2 ops for same result)
r1=0, r2=1, x=1, y=0	y=x; x=1; r1=y; r2=x (can swap last 2 ops for same result
r1=0, r2=0, x=1, y=1	r1=y; r2=x; x=1; y=x (can swap first 2 ops for same result)

Question 5

One compiler optimization could be loop unrolling: we could process 4 elements at a time in the inner loop by incrementing i by 4 each time instead and updating the summation accordingly. This requires the condition that there are no loop-carried dependencies, which there aren't.

Another compiler optimization could be function inlining: we could inline the max function at line 17.

Question 6

Written in CUDA instead of OpenCL. The atomic max function is used to ensure atomic writes to the result.

```
__global__ void kernel1 (double *u, double *w, int M, int N) {

        int i = blockIdx.x*blockDim.x + threadIdx.x + 1;

        int j = blockIdx.y*blockDim.y + threadIdx.y + 1;

        w[ i *N+j ] = ( u [ (i – 1)*N +j ] + u [( i + 1)*N+ j ] + u [ i *N + j −1] + u [ i *N + j +1] ) / 4 . 0;

}

__global__ void kernel2(double *u, double *w, int M, int N, double *diff) {

        int i = blockIdx.x*blockDim.x + threadIdx.x + 1;

        int j = blockIdx.y*blockDim.y + threadIdx.y + 1;

        double diffNew = fabs(w[i*N+j] – u[i*N+j]);

        atomicMax(diff, diffNew);

}
```