Part 1, Theory

Problem 1, GPUs

a) Every thread is contained in thread blocks, which are organized into a grid. Every block is assigned to one SM (streaming multiprocessor). The multiprocessor "*creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps.*" In other words, the SM get several thread blocks it's supposed to execute. It organizes these block's threads into wraps of size 32, and execute one wrap at a time.

b)

- Threads have their personal registers and memory.
- Each thread block have a *shared memory,* which all the threads share. Small (~48 kB), but fast.Used to share data among threads, and avoid too many global memory accesses.
- All the thread blocks share a *global memory (DRAM)*. Bigger (1-6 GB), but several hundred times slower. Used to transfer data from CPU to GPU and launching kernels.

c)

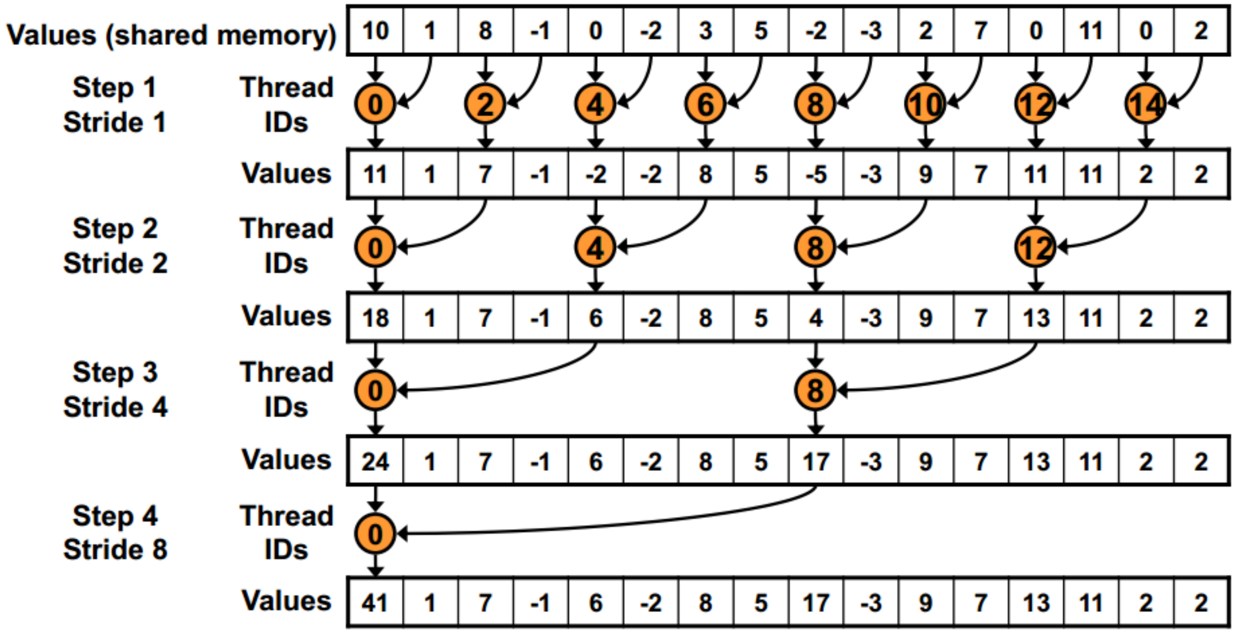
__synchthreads() is a barrier that is only visible inside the respective block thread, i.e. it synchronizes the threads within the block.
 __threadfence() is also a barrier, but it's visible for all the other threads in the GPU. Thus it is used when you wish to halt the execution of a thread, until you are sure that all other threads (in other blocks) are aware of the data that it has written.

Problem 2, Reduction

a)

1. Provide each thread with two values it will use the operand on, and store the result in shared memory.
2. Reduce the number of threads by a factor of 2, and make them compute two of the newley computed values and store the result in shared memory.
3. Repeat step 2 until 1 thread remains.

(See illustration)



b)

Code doing the what's illustrated in the figure above:

```
__global__ void reduce(int *g_idata, int *g_odata) {

extern __shared__ int sdata[];
int thread_id = threadIDx.x;
int index = blockIDx.x*blockDim.x + thread_id;
//Each thread loads one element of the array into shared memory.
sdata[thread_id] = g_idata[index]
__synchthreads();

//For loop, preforming each step in the tree.
for (int i = 1; s < blockDim.x; i *=2) {
        if (thread_id%(2*s) == 0) { //Check if thread should be used on this step.
                sdata[thread_id] += sdata[thread_id + i]; //Compute data
        }
}

if (thread_id == 0) g_odata[blockIDx.x] = sdata[0]; //Store result in global mem.
```

Sources:
http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf

Problem 3

a) Divergent branches are when the program make the threads branch to different parts of the program, i.e. based on thread id. This causes a reduction in performance, since the threads have to execute the same part of the program at the same time. Lets say that a thread enters a if-block, while another one branches to the else-block. Then the second thread will be disabled while the first thread executes the if-block, and visa versa.

An important point is that a divergence can only happen within a wrap. Thus if you're able to make all the threads in the same wrap branch to the same block, you will have no divergence and no performance loss.

b)

1) blockDim/16 = 128/16 = 8. Thus 8 threads will branch to the if-block. This might cause a big loss of performance, if all these threads are spread into all the warps. If the 8 threads are set into the same wrap, the performance loss will not be to terrible.

2) No problem. Since all the threads in the same thread block, and thus the same wrap, will enter the same branch-block. Thus there will be no divergence.

3) Every other thread will branch to a different branch-block, which is very likely to case a performance loss. Unless you're really lucky with the wraps.

Problem 4, Deadlocks

To avoid deadlocks in cuda, it's important that __synchthreads() is called unconditionally within a block. Thus the first one is not safe, since some threads in each block will go into the if-block, and some will go into the else-block. Which will make the program deadlock the first synchthread.

The second is safe since every thread in the same block will enter the same branch-block (since the condition is basically which block the thread is in).

The third might be safe, depending on the return value of my_function1. If it returns the same value for all threads in the same block, it’s safe. Otherwise it might deadlock.

Problem 5

In my solution (which you see by checking my code) each block is to preform a blue filter on a row. Thus I load each row and it’s neighbour rows into the shared memory. Each thread loads the pixel it’s supposed to calculate, and the pixels in the above row and the row underneath. E.g. if we have this matrix:

1 2 3
4 5 6
7 8 9
10 11 12
13 14 15

Then block 1 will load:

1 2 3
4 5 6 <- “block 1”s domain.
7 8 9

Block 2 will load:

4 5 6
7 8 9 <- “block 2”’s domain.
10 11 12

Part 2, Code

The algorithm is divided into four parts:

1. Reset the centroids position.
2. Sum all their points positions into their respective centroids position.
3. Calculate the new position to the centroids.
4. Find the best centroid for each point.

Part 1 and 3 read and write only to centroids. Part 2 and 4 read and writes each point, and reads and writes every centroid. Thus I decided that I wanted to preform part 1 and 3 in parallel, and 2 and 4 in parallel. Since I can then make each thread work on a single centroid or point.

In order to do this I had to create an additional centroid array, in order to avoid data dependencies. So, if we have two centroid arrays C1 and C2, my algorithm would look like this:

```
Initialize C1 with random centroids
Initialize P1 with random points
Preform part 1 and part 2 on C1
while (updated)
    In parallell:
        - C2: reset the centroids positions
        - C1: calculate the new positions to the centroids
    In parallell:
        - Find the best centroid for each point using C2
        - Sum all their points positions into their respective centroids position.
    Switch C1 and C2
C1 = C2
Done.
```

In the second parallell part I put all the centroids into the shared memory for fast access. Since every block can maximum contain 4096 centroids, I'll have to switch the out the shared memory when 4096 centroids have been processed, and repeat this until every centroid have been processed.