Problem Set 5

Jørgen Faret - MTDT

October 31, 2013

Part 1, Theory

1

a)

Thread

A thread in CUDA is different to a traditional CPU thread in the sense that is a lot more lightweight. A thread in CUDA is an execution of the kernel with a given thread index. For example when accessing elements in an array, each thread uses it's thread index to determine whch element in the array to access. This means that a thread in CUDA can be looked at as a basic data element to be processed.

Warp

The term warp in CUDA means a collection of threads with the collection size corresponding to the minimum amount of data processed in SIMD fashion by a CUDA multiprocessor. In theory, this is system-dependant but in practice currently the warp size is 32 always threads.

Block

A block is, like a warp, a collection of threads. The block size is specified by the programmer, and should be a multiple of the warp size.

Grid

A grid is a collection of blocks. The size of grids is system-specific. A kernel is executed as a grid of blocks of threads.

b)

Registers

Registers are thread-local and are the fastest memory types available.

Thread-local memory

Local memory is memory that is local and only visible to one thread. This memory actually resides in the global memory unless placed in registers, and memory accesses are therefore quite slow.

Thread-shared memory

This is fast memory that is local to a specific block of threads. The amount of shared memory is limited to 16 KB per block.

Constant Memory

Fast memory that is local to a grid of threads. The memory includes a cache, but is limited to 64 KB.

Texture Memory

Texture memory is a read-only memory that is located in the same location as global memory. Because it is read-only it does not suffer the same performance hits as global memory, but texture memory only supports the data types int, char, and float.

Global Memory

Global memory is the most abundant memory space on the GPU. All threads can access elements in global memory, but accesses to global memory are very slow. Therefore, when programming a kernel, it is best to try to avoid global memory accesses.

c)

Syncthreads acts as a traditional barrier between all the threads in a block, meaning that when syncthreads is called then none of the threads in the block where it is called will continue until every thread in the block has reached the syncthreads call. Threadfence halts the current thread until all previous writes to shared and global memory are visible to other threads, but it does not affect any other threads than the one calling the function.

2

a)

Parallel reduction can be implemented using a tree structure by first an element from threads in groups of two, and adding these two elements and then sending these to the first elements of the two. Then the same step is repeated for groups of 4, then 8 and so on.

3

a)

A divergent branch, or warp divergence, is when the kernel includes a predicate condition that causes a branch in the program in a warp where not all threads in the warp follow the same path through a branch. Put into simple terms, a warp divergence is when the 32 threads in a warp are not all set to do the same thing. For example, given an array of pixels where every other pixel should be set to blue and every other should be set to green, the following code would cause a divergent branch:

```
if (threadId % 1 == 0) {
        pixels[threadId] = blue;
}
else {
        pixels[threadId] = green;
}
```

A way to avoid warp diversion in this example is to rearrange my data so that the pixels that need to be set to blue are aligned, and the pixels that need to be set to green are aligned. That way, one warp would set all of its pixels to green and another would set all of its pixels to blue.

b)

Sample 1:

This program says that if a thread has an ID greater than 8, than it executes “my_function1”, otherwise it executes “my_function2”. This program has warp divergence, because for every block the first 8 threads will execute “my_function2”, and the rest will execute “my_fuction1”.

Sample 2:

This program says that if a block has an odd block ID, all the threads in that block execute “my_function1”, otherwise the threads in a block execute “my_function2”. This program has no problems with warp divergence because all the threads in a block execute the same task, which means all the threads in the same warp execute the same task.

Sample 3:

This program states that if a thread's thread ID is greater than 64, then it enters the it executes "my_function1", otherwise it executes "my_function2". This means that threads 0 to 64 will enter the if block, and threads 65 to 127 will enter the else block. In other words, the first element of warp number three will enter the if block, and the rest of the elements in the block will enter the else block. This means that the program has warp divergence.

4

Sample 1:

This program states that if a thread's ID is greater than 131, then it executes "my_function1", then syncthreads, then "my_function2". Otherwise, the thread executes "my_function2", then syncthreads, then "my_function1". This will cause a deadlock, because some elements in a warp will get through the if statement, while others will not. Note that I'm assuming that the threads need to hit the same syncthreads statement to continue, and that they will deadlock even if they hit different syncthread statements.

Sample 2:

In this program, if a thread's block id is greater than 2 then it executes "my_function1", then syncthreads, then "my_function2". Otherwise, it executes "my_function2, then syncthreads, then "my_function1". This program can not cause a deadlock, because all the threads in a block will either enter the if statement or the else statement.

Sample 3:

What happens in this program is that if "my_function1" returns true for a given thread, and if that thread's block ID is greater than 2, the thread executes "my_function2" and then waits for all the other threads in the block at syncthreads Finally, the thread executes "my_function3". If the thread's block ID is not greater than 2, then it executes "my_function3" directly after entering the while loop. This program can potentially deadlock because if some threads in the block don't get through the clause after the while statement, and some do, then the threads who entered the while loop will wait for the threads who didn't ininitely at the call to syncthreads. In other words: the program deadlocks if the function "my_function1" can return different values for threads in the same block.

5

Shared memory can be used to speed-up our program to blur an image because each block can store their part of the image in shared memory with a halo. This will speed up the time it takes to read the different pixels. Because this program only runs with one iteration, there is no need for a border exchange.

Part 2, Code

1

See blur_cuda.cu.

2

See kmeans_cuda.cu.

Description of implementation

Implementing a parallelized version of the k-means clustering alorithm has two steps:

1. Paralellizing the step where centroids for the clusters are calculated.
2. Paralellizing the step where points are set to a cluster.

I chose to perform the calculation of new centroid values by assigning each cluster a thread. This implementation requires a large amount of clusters compared to points to become efficient. The kernel function that performs this calulcation in my program is called new_centroids(). The first step of the function (line 153) is checking that the thread ID doesn't exceed the number of clusters. This line, in addition to the addition of 1 on line 243 when the grid size is set, is intended to make my program more robust to different kinds of input values.

Each thread iterates through all the points and checks if it belongs to the current cluster. If it does then it's values are added, and at the end of the function the values for the x and y for the current cluster are divided by the number of points in the cluster. A central assumption when choosing this implementation is the assumption I made that the process of checking if a point belongs to a cluster is very fast compared to the process of adding the coordinates of a point to a cluster.

The step where points are set to a cluster was implementing by assigning one thread to each point. Then, quite similarily to the naive version, the distances to each cluster is compared for each point.