CSCI-GA.3033-008

Graphics Processing Units (GPUs): Architecture and Programming

Fall 2013 – Dec 17th, 2013 (90 minutes)

NAME: **ID:**

- This exam contains 5 questions with a total of 40 points.
- The exam is open book/notes but no electronic devices.
- If you have to make assumptions to continue solving a problem, state your assumptions clearly.
- You answer on the question sheet. You can use extra white papers if you want.

1. As a CUDA programmer, how does knowing about the concept of warps help you, especially that warps are transparent to the programmer?

- In the kernel invocation, chose a number of threads that divides evenly with the number of threads in a warp. Otherwise, you will not make the best use of underlying hardware.
- Knowing about warps helps writing your code in a way to reduce branch-divergence as much as you can.
- Knowing about warps helps you access the memory in a way to increase the opportunities of memory coalescing.

2. Suppose we have a compute bound application with enough parallelism. Discuss the pros and cons of each the following two strategies: (i) more blocks per gird and less threads per block (ii) more threads per block but less blocks per grid. Assume the total number of threads is fixed.

(i) + smaller blocks can make better use of SM resources: when several blocks are assigned to the same SM.
- less opportunities to synchronize among threads: because synchronization is done among threads of the same block and here block size is small
- If block size turned out to be less than a warp-size then there is a performance loss here.

(ii) + more opportunities for sharing among threads
- Large blocks may not make the best use of resources in an SM because the granularity of assignments to SM is a block granularity.

3. Suppose an NVIDIA GPU has 8 SMs. Each SM has 32 SPs, but a single warp is only 16 threads. The GPU is to be used to add two arrays element-wise. Assume that the number of array elements is 2^{24}. Let t denote the amount of time it takes one thread (yes, just one) to perform the entire calculation on the GPU. The kernel code is shown below (num_threads is the total number of threads in the whole GPU):

```
__device__ void prob(int array_size) {
        int tid = threadIdx.x + blockIdx.x * blockDim.x;
        for ( int i=tid; i<array_size; i += num_threads )
                result[i] = a[i] + b[i];
        }
```

(*a*) What is the amount of time it takes if we use one block of 16 threads?

Execution time is t/16.

(b) What is the amount of time it takes if we use two blocks of 8 threads each?

Execution time is t/16.

(c) Justify why the above two answers are similar/different.

The total number of threads is 16 in both cases. The one-block configuration would run on one SM and would be able to use 16 cores, the two-block configuration would run on two SMs and would be able to make use of 8 SPs each, for a total of 16.
Since there are enough SPs for all threads and none of the threads will have to wait for another thread (i.e. threads are independent) , the rate of computation will be the same as the work is divided evenly among threads, the execution time in both cases will be t/16.

(d) Assume that 256 threads are enough to keep all SPs in the SM busy all the time. What is the amount of time it would take to perform the computation for one block of 1024 threads? Justify.

There would be no difference in performance in configurations from 256 to 1024 threads per block. Then the time is t/256.

(e) Repeat question (d) above but with two blocks of 512 threads each.

Since there are two blocks two SMs will be used and so the time will be based on 512 threads, so the execution time is t/512.

4. Given the following code (assume balls are already in GPU global memory and the variables balls_per_thread and delta are defined elsewhere) :

```
struct Ball {
      float position;
      float velocity;
};
struct Ball* balls;  /*
__device__ void update(float delta) {
           int start=(threadIdx.x+blockIdx.x*blockDim.x)* balls_per_thread;
            int stop = start + balls_per_thread;
            for ( int i=start; i<stop; i++ )
                     balls[i].position += delta * balls[i].velocity;
}
```

If we assume that all threads are assigned to one block, explain **two** enhancements to speed-up the above code. You don't need to write code but show the parts that need enhancement, explain why they need enhancement, and what is your fix, and why your fix is actually better than the original.

Solution:

Memory access is most efficient when consecutive threads access consecutive locations, so that coalescing can be used by the hardware. In the code above consecutive threads will access memory addresses that differ by a factor of (balls_per_thread * sizeof(Ball)), which is not the best thing.
Two things can be done to fix that.

First, the array index will be computed so that it is equal to threadIdx.x + something, where something is the same for all threads in a block.
For example:

```
int start = threadIdx.x + blockIdx.x * blockDim.x;
for ( int i=0; i<balls_per_thread; i++ ) {
   int idx = start + i * num_threads_per_block;
   then you access the balls here using idx as index}
```

Second, the data will be reorganized from an array of structures, to just two arrays.

5. The line of code below checks for a special case to avoid calling an expensive square root. Describe a situation in which it makes sense for CUDA to do that, and a different situation when it makes no sense (meaning it would be faster to do the square root all the time). Assume that 50% of the time d is equal to 1.

```
if ( d == 1 ) s = 1; else s = sqrt(d);
```

Solution:

It makes sense if there is no branch divergence, meaning that for all warps the branch is either always taken or always not taken.
If there is branch divergence then the code will run more slowly than code always performing the square root since within a warp the execution time will be the sum of both paths through the if.