Graphics Processing Units - Fall 2018
Homework 2

Daniel Rivera Ruiz
Department of Computer Science
New York University
drr342@nyu.edu

1. Assume the following piece of code is running on G80:

```
1  #define VECTOR_N 1024
2  #define ELEMENT_N 256
3  const int DATA_N          = VECTOR_N * ELEMENT_N;
4  const int DATA_SZ         = DATA_N * sizeof(float);
5  const int RESULT_SZ       = VECTOR_N * sizeof(float);
   ...
6  float *d_A, *d_B, *d_C;
   ...
7  cudaMalloc((void **)&d_A, DATA_SZ);
8  cudaMalloc((void **)&d_B, DATA_SZ);
9  cudaMalloc((void **)&d_C, RESULT_SZ);
   ...
10 scalarProd<<<VECTOR_N, ELEMENT_N>>>(d_C, d_A, d_B, ELEMENT_N);
11
12 __global__ void
13 scalarProd(float *d_C, float *d_A, float *d_B, int ElementN)
14 {
15     __shared__ float accumResult[ELEMENT_N];
16     //Current vectors bases
17     float *A = d_A + ElementN * blockIdx.x;
18     float *B = d_B + ElementN * blockIdx.x;
19     int tx = threadIdx.x;
20
21     accumResult[tx] = A[tx] * B[tx];
22
23     for(int stride = ElementN /2; stride > 0; stride >>= 1)
24     {
25         __syncthreads();
26         if(tx < stride)
27         accumResult[tx] += accumResult[stride + tx];
28     }
30     d_C[blockIdx.x] = accumResult[0];
31 }
```

   (a) How many threads are there in total?
   There are ELEMENT_N$= 256$ threads per block and there are VECTOR_N$= 1\,024$ blocks, so the total number of threads is $1\,024 \cdot 256 = \mathbf{262\,144}$.

   (b) How many threads are there in a warp?
   The number of threads per warp is fixed for all Nvidia GPUs and it is equal to $\mathbf{32}$.

(c) How many threads are there in a block?
As stated in (a), the number of threads per block is $\mathbf{256}$.

(d) How many global memory loads and stores are done for each thread?
According to the *CUDA C Programming Guide*, `__global__` function parameters are passed to the device via constant memory, which means that lines 17, 18 and 19 are only accessing either constant memory or registers. Therefore, per thread there are only two load operations (on line 21, one for `A[tx]` and one for `B[tx]`) and one store operation (on line 30 for `d_C[blockIdx.x]`).

(e) How many accesses to shared memory are done for each block?
- On line 21 there is one write operation to shared memory (`accumResult[tx]`) per thread, so $\mathbf{256}$ in total.
- Within the `for` loop, line 27 is executed by half of the threads on the first iteration. For further iterations the number of threads that execute the line is halved. Each execution of this line requires two accesses to shared memory (one read of `accumResult[stride+tx]` and one write to `accumResult[tx]`). Therefore, the total number of accesses associated to the `for` loop is $2 \cdot (128 + 64 + 32 + 16 + 8 + 4 + 2 + 1) = \mathbf{510}$.
- Finally, on line 30 each thread reads `accumResult[0]`, which results in $\mathbf{256}$ accesses.
- The total number of accesses to shared memory per block is therefore $256 + 510 + 256 = \mathbf{1\,022}$.

(f) How many iterations of the for loop (Line 23) will have branch divergence? Show your derivation.
Branch divergence is calculated per warp, and the condition for divergence is introduced by line 26: `if (tx < stride)`. Therefore, as long as `(stride % WARP_SIZE) = 0` there will not be divergence within the warp. In the previous exercise we noticed that stride$(i) = 2^{7-i}, i \in [0, 7]$, and since we know that `WARP_SIZE = 32`, it is clear that the first 3 iterations (stride $= \{128, 64, 32\}$) won't have divergence and the last 4 (stride $= \{16, 8, 4, 2, 1\}$) will.

(g) Identify an opportunity to significantly reduce the bandwidth requirement on the global memory. How would you achieve this? How many accesses can you eliminate?
The accesses to global memory on line 21 are already optimized and cannot be avoided:
- Each thread is reading one element from A and one from B.
- There's no point on storing these values in shared memory because they will only be used once.
- Consecutive threads read consecutive addresses in memory, so the memory access is coalesced.

On line 30, however, the writing to global memory is very inefficient because all threads within the block are writing the same value (`accumResult[0]`) to the same location in global memory (`d_C[blockIdx.x]`), effectively overwriting it 256 times. To avoid this, line 30 can be replaced with the following:

```
1 if (tx < WARP_SIZE) {
2     d_C[blockIdx.x] = accumResult[0];
3 }
```

By doing this small change, we reduce the number of writes to global memory per block from 256 to 32, while avoiding branch divergence within the warps of the block.

2. What factors can make two threads corresponding to two different warps but of the same block take different amounts of time to finish? To get full credit, write at least 3 factors.
   - *Conditional statements*. Even though branch divergence (within a warp) can be avoided by a well designed condition, different execution times for different warps will be almost inevitable, unless the instructions within both branches of the condition take similar amounts of time to execute.
   - *Memory access*. If the memory being accessed by one of the threads is relevant to the whole application, it is more likely to be cached. Under this circumstances, the execution of this thread will be considerably faster compared to a thread that has to constantly fetch data from global memory.
   - *Hardware considerations* It might be the case that the hardware associated to a given thread (SP) is inherently slower due to small variations in the manufacturing process of the device that cannot be avoided nor predicted.

3. What is the difference between shared memory and L1 cache?
   While both memories share the same on-chip physical location that considerably reduces latency compared to global memory access, the main difference between shared memory and L1 cache is the level of flexibility that they offer to the programmer.
   In the case of L1 cache this flexibility very limited, since the caching policies are unknown and hard to predict. Additionally, one line of the L1 cache is usually 128 bytes, so if the memory access of the application is not coalesced to make the most out of the line, cache misses will become more frequent.
   In contrast, the programmer has full control over the contents of shared memory and he makes it explicit within his code. This means that he can bring data from global memory into shared memory at will (as long as it fits) and at any level of granularity (non-coalesced access is possible, although discouraged if performance is to be maximized). Since shared memory is available to all the threads within a block, it facilitates and speeds up communication and data transfer among them, making it a very powerful resource when used properly.

4. Can memory be coalesced for threads in a warp yet not-coalesced for threads in a different warp of the same block?
   It is possible. The following portion of code shows an example of this:

```
__global__ void kernel(float * A, float * B, int n) {
    if (threadIdx.x < WARP_SIZE) {
        B[threadIdx.x] = A[threadIdx.x];
    } else {
        B[threadIdx.x] = A[n * threadIdx.x];
    }
}
```

   This kind of behaviour might not be very useful nor desired in most circumstances, but it is a simple instance where threads within the first warp of the block ($0 \leq$ `threadIdx.x` < 32) will have coalesced memory access, and all other threads will not (assuming that $n \neq 1$).

5. Suppose you want to write a kernel that operates on an image of size (400×900) pixels. You want to assign one thread to each pixel. Your thread blocks are square in geometry, and you try to use the maximum number of threads per block possible on the device. The maximum number of threads per block is 1024. How would you select the grid dimensions and block dimensions of your kernel?
   Let N be the size of the squared blocks $B = (N \times N)$, and let $W = \lceil \frac{400}{N} \rceil$ and $H = \lceil \frac{900}{N} \rceil$ be the dimension of the grid $G = (W \times H)$. The number of unused threads U at the device level is given by the following expression:

$$U = \left(WHN^2 - 400 \cdot 900\right) + \left(WH(-N^2 \bmod 32)\right)$$

   Where the first part accounts for the threads that will be unused if the blocks don't fit exactly the size of the input grid, and the second part accounts for the threads within each block that will be unused if the number of threads per block is not a multiple of the warp size (32).
   So the problem is reduced to finding the value of N that minimizes U:

$$N^* = \underset{1 \leq N \leq 32}{argmin}(U)$$

   If we simply tabulate for all possible values of N, we find that $N = 8$ yields the optimal solution with $U = 1\,600$, $W = 50$ and $H = 113$.
   This result shows that as a rule of thumb it is better to optimize for the number of threads within a block to be a multiple of the warp size, even if that means generating some extra threads in the grid. This makes sense because at least there is a guarantee that each block will fully span a certain number of warps.
   Trying to optimize for the blocks to fit exactly the dimensions of the input data is much more risky: at first glance it will look as if there were no unused threads within the grid, but under the hood as many as 31 threads can be idle each time the warp scheduler schedules the last warp of a block.

6. 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?
Since the block will fill entirely one warp it can be scheduled all at once. The 16 threads will run in parallel and effectively take the same amount of time t of a single thread (though the cumulative time for all threads will obviously be $16t$). Since there are 2^{24} elements to process and each block processes 16 elements at a time, we need $2^{24}/16 = 2^{20}$ blocks to perform the whole calculation. Each SM can handle 2 blocks (since it has twice the SPs as the size of the warp) and we have 8 SMs, so we can execute $8 \cdot 2 = 16$ blocks at once in the whole GPU. The overall running time for the whole application will therefore be:

$$T_a = \frac{2^{20}t}{16} = 2^{16}t$$

(b) What is the amount of time it takes if we use two blocks of 8 threads each?
In this case each block is wasting half the threads in a warp, because one warp is the smallest scheduling unit and warps cannot span multiple blocks. All other things being the same, we know that the overall running time of the application will be:

$$T_b = \frac{2^{21}t}{16} = 2^{17}t = 2T_a$$

(c) Justify why the above two answers are similar/different.
As explained in the previous answer, half the threads in the whole device are being wasted because the size of the block is too small. Thus, the time required with blocks of size 8 is double the time required with blocks of size 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 $1\,024$ threads? Justify.
One block of $1\,024$ threads will span 64 warps and we will need $2^{24}/1\,024 = 2^{14}$ such blocks to process all the elements in the arrays. Since the device is the same we can still process 16 warps at any given time, which means there is $16/64 = 1/4$ of a block occupying the device. Therefore, the overall running time of the application will be:

$$T_d = \frac{2^{14}t}{1/4} = 2^{16}t = T_a$$

(e) Repeat question (d) above but with two blocks of 512 threads each.
One block of 512 threads will span 32 warps and we will need $2^{24}/512 = 2^{15}$ such blocks to process all the elements in the arrays. Since the device is the same we can still process 16 warps at any given time, which means there is $16/32 = 1/2$ of a block occupying the device. Therefore, the overall running time of the application will be:

$$T_e = \frac{2^{15}t}{1/2} = 2^{16}t = T_a$$

As we can see, the results for (a), (d) and (e) are all the same since all configurations achieve full occupancy of the device. In the configuration for (b) we only reach 50% occupancy and threfore the time required is doubled.

7. 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);
```

The main factor to take into account to decide if executing this line of code is beneficial or not, is branch divergence:

- If the condition is guaranteed to evaluate the same for all threads within a warp, the warps where the condition is true can execute much faster (avoiding completely the square root) and speed up the overall execution time. In this scenario it is favorable to include the condition.
- If the condition will not evaluate the same for all threads within a warp, thread serialization will occur and the overall performance will be affected. In this scenario it is not favorable to include the condition.

Let's suppose for instance that we are trying to access the elements of a matrix `A[i][j]` with blocks of size (32×32). Each row of the block will map to a warp and will access a portion of 32 consecutive elements in `A`. If we can guarantee that half the rows of `A` are filled with ones, we are in the first scenario and using the conditional statement is beneficial. However, if we only know that half the elements of `A` are ones but know nothing about their distribution, then we are in the second scenario and always performing the square root is better.

8. What is wrong with this piece of kernel code? How to deal with it if we need to `__syncthreads()` both in the if-body and else-body (i.e. how to change the code yet preserve the semantic)?

```
1 if {
2     ...
3     __syncthreads();
4 }
5 else {
6     ...
7     __syncthreads();
8 }
```

The problem with this code is that it is prone to deadlocks:

- `__syncthreads()` works at a block level, which means that every thread that reaches this statement, will wait for *all* threads in the block to reach it as well before continuing execution.
- If some threads within the block execute the code inside the `if` statement and others execute the code inside the `else` statement (which is to be expected, otherwise the conditional would be pointless), the first set of threads will be hanging forever on line 3, waiting for the second set of threads who are in turn waiting on line 7.
- At this point no further progress can be made and the kernel will hang forever in a deadlock condition.

To avoid this problem, we have to take the synchronization commands out of the conditional statement, to guarantee that all threads can reach them. Assuming that we are testing for condition C in the original code, the deadlock-safe version would be as follows:

```
1 if (C) {
2     ...
3 }
4 __syncthreads();
5 if (!C) {
6     ...
7 }
8 __syncthreads();
```