Assignment 4

Report

Dominique Roduit (234868)

1 – Explain how you implemented GPU_array_process and its optimizations

I inspired myself from the provided file « example.cu » and the lectures hand-on for the main steps :

- allocate the input/output arrays on device
- copying arrays from host to device
- Invoke kernel
- copying array from device to host
- Deallocate the resources on device and host.

The only difference with « example.cu » where they just needed to organize the thread blocks into 1D arrays of threads, was that we had to define the number of blocks and threads per block in a dim3 variable in order to organize the thread blocks into 2D arrays of threads. It’s one of the main difference.

Kernel

To implement the kernel, I reused nearly the same CPU baseline provided. The difference is that we don’t need the loops because we can simply define the loop index-like as seen in class with :

```
int i = (blockIdx.x * blockDim.x) + threadIdx.x;
int j = (blockIdx.y * blockDim.y) + threadIdx.y;
```

We use these variables in a condition to avoid processing the output of the middle hot core and the borders. However, since we not depend only on the threadId but also of the position in the block, all the threads in a warp will follows the same path, and thus, we won’t suffer of thread divergence.

Optimizations

My optimizations reside more in the tunning of the kernel parameters « Number of blocks » and « Threads per blocks », since I don’t suffer from thread divergence, and don’t suffer from bank conflicts since I don’t use the shared memory (I attempted to use the shared memory with _syncthreads() but failed since my output file was full of 0s). Also, we don’t have any loop to unroll.

« It is common to recommend :

- *running more threads per multiprocessor*
- *running more threads per thread block*

Motivation : this is the only way to hide latencies
*But … **Faster codes run at lower occupancy**. Maximizing occupancy, you may lose performance »*

Nvidia advice : https://goo.gl/9cyhLe

Provided the values in the lecture slides, we can find that the GPU has :

- 15 Sms
- 16 blocks per SM
- 4 warps per block
- 32 threads per warp

For a length of 100, the performances are acceptable with almost any parameters, so we focus on the square length of 1000.

If we want to follow the Nvidia advice above, we have to use the fewest threads possible. Then we try to maximise the use of the blocks and to have the fewest thread possible in every threads.

To do so, I started to fixed the number of thread per block to 32 and I tried to decrease this number progressively. The number of blocks on both dimensions is picked according to the length and the number of threads per blocks as :

```
int nBlks = ceil((double)length/thrsPerBlock);
```

In decreasing the number of threads per block, I noticed that the performance became better and better. For 8 threads per blocks, the performance was the best. Indeed, If we have 8 threads per blocks on both dimensions, we will then have ceil(1000/8) = 125 blocks on both dimensions of the grid. Thus, 125x125 thread blocks of 64 threads each = 125*125*64 = 1’000’000. It’s exactly the total elements contained in the square of length 1000. We run at lower occupancy.

Actually, I had many problems in defining these parameters. Firstly, I made a lot of calculations by hand to maximize the occupancy, and indeed the performances was surprisingly good (about 0.28s), but the big problem was that my output matrix was full of zeros in running with a square length of 1000. So I prefered to get a correct result with higher running time rather than a breathtaking running time with no result for the square length of 1000. I really took a lot of time to tune and try different parameters.

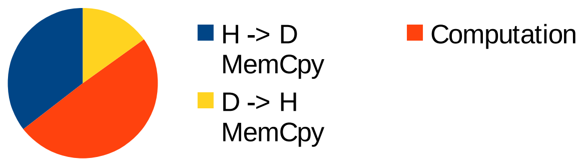
2 –

a.

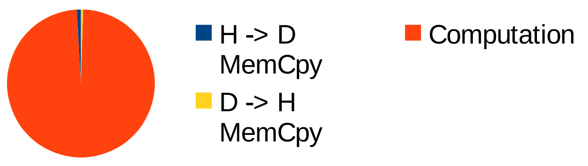
Length	Iterations	Reference CPU function	Optimized CPU version (assignment 2)	Optimized GPU version
100	10	0.000145s	0.000512s	0.1685s
100	1000	0.02117s	0.01137s	0.1819s
1000	100	0.1473s	0.03449s	0.227s
1000	10000	21.67s	4.175s	5.596s

b. Present graphically the breakdown of the GPU runtime for copying data from host to device, computation, and copying data from device to host.

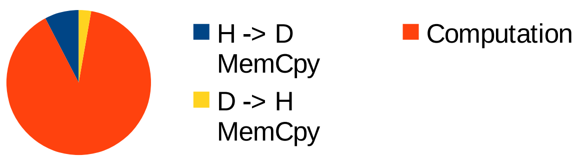
100, 10
Host to Device MemCpy takes 9.584e-05s
Computation takes 0.0001335s
Device to Host MemCpy takes 4.08e-05s



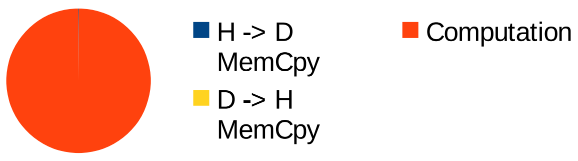
100, 1000
Host to Device MemCpy takes 9.67e-05s
Computation takes 0.01093s
Device to Host MemCpy takes 4.221e-05s



1000, 100
Host to Device MemCpy takes 0.004736s
Computation takes 0.05542s
Device to Host MemCpy takes 0.001689s



1000, 10000
Host to Device MemCpy takes 0.004759s
Computation takes 5.42s
Device to Host MemCpy takes 0.002112s



c. Analyze and explain the results of parts a and b.

In part a) we see that we have a good speedup for the optimized GPU version compared to the baseline CPU single-threaded implementation, especially for the biggest square length and the most iterations. However, the running time with GPU is globally worse than with the CPU optimized version, which is in my opinion not normal. I probably failed somewhere.

The part b) shows clearly that the running time is mostly consumed by the computations. This is in a sens normal since our algorithm is rather memory intensive and that « Computations » include the whole execution of the kernel where all the memory accesses are done. But it also mean that we could probably improve the kernel. I think that we could use the shared memory to improve the running time significantly, but I failed to get correct results in my output matrix in doing it so I prefered to provide a working implementation, rather than a fast one that doesn’t work.