Reduction algorithm Faculty of computer science
AGH UNI, 2020-2021 ROUSSEAUX Thibaut – 11/01/2021

Introduction to CUDA and OPENCL

Introduction :

The purpose of this lab is to implement an algorithm that performs reduction of a one-dimensional input array.The final output should be the sum of the all elements in the array.

I used the lhcbgpu2 server to do and execute the program, if you can enter and execute student's directory you will fin dit here : /home/thirou_cuda/algo

It can be executed using for exemple : « ./reduction -nb=10000 » nb being the size of the array.

Otherwise the code is accessible on my github here : https://github.com/Swyfee/Reduction_CU/tree/master

Answers to questions :

-Why the reduction algorithm is so vital, start from naming a few problems that represent this type of processing?

Processing a huge array of x values, means that a single thread must reach each value once, meaning x operations. The reduction algorithm is vital as it seriously lower the processing time, especially if the array is immense. Indeed, instead of browsing x times, the reduction algorithm allow to use x/2 threads and reduce the number of steps. If x=32, 16 threads can be used, and there will only be 5 steps vs previously 32, as each thread compares 2 values. We can see that the reduction is already huge even on small data sizes. It only increases as the data size also increases.

Also, if you use a single thread, your program can't execute another part of the code so the lost time is real.

-Analyse the workload of each thread. Estimate the min, max and averagenumber of real operations that a thread will perform(i.e., when a thread calculate a number contributing to the final result). How many times a single block will synchronise to before finishing its calculations (until a single partial sum is obtained)?

I assume that I will take an exemple for answering this question. Let's assume our array size is 128. This means that only 64 threads will have a job. On the first iteration, each thread will sum their index to their index (see the code for the index calcul allowing the « shift »)+stride and store their results in their own index. That means that there is now 64 indexes to sum. Now only 32 threads will have a job, doing the sum of their index with their index+32. This goes on until only one thread will have one sum to do which will give the results. For this exemple, the threads 33 to 64 will only have 1 operation, the 1rst thread will have 6 operations and:

32 threads have 1 operation, 16 threads have 2 operations, 8 threads have 3 operations, 4 threads have 4 operations, 2 threads have 5 operations and 1 thread have 6 operations.

The average will so be (1/32+2/16+3/8+4/4+5/2+6/1)/6=1.67 operations/thread.

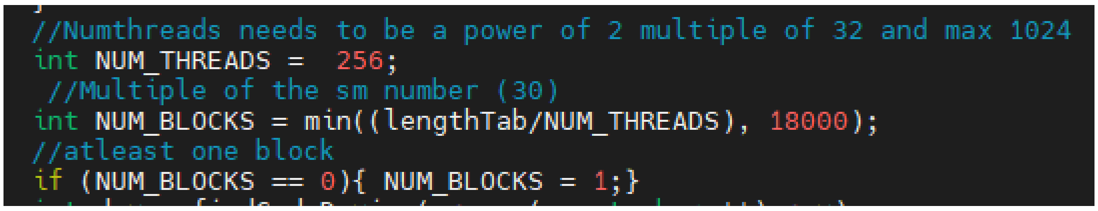
Comparing to a single threaded application which would have : 128/1 : 128 operations/thread.

A block will synchronize the number of maximum required operations +1.

-Describe the possible optimisations to the code?

I am thinking of a better occupency as the impact of the array size over the number of threads and blocks in my code is quite low. As the constraints over the thread number is high I fixed it to 256 (multiple of 32, power of 2, <1024) and for the number of blocks I make it increase as the data size increases to a maximum number (18000), multiple of 30 (number of SM).

I ended up with those variables.



```
//Numthreads needs to be a power of 2 multiple of 32 and max 1024
int NUM_THREADS =  256;
 //Multiple of the sm number (30)
int NUM_BLOCKS = min((lengthTab/NUM_THREADS), 18000);
//atleast one block
if (NUM_BLOCKS == 0){ NUM_BLOCKS = 1;}
```

The number of blocks is so not ALWAYS a multiple of 32 and the number of threads not ALWAYS the Datasize/2. I assume that threads will however be assigned but not used so the drop of performance musn't be that high, and I actually am getting pretty good results. In my case, the maximum array size for the algorithm to work is (18000*256)*2= 9 216 000 as the total number of thread must be minimum arraysize/2, so it also can be modified to handle higher numbers.

-Is the use of atomic operations advisable in this case?

It is useless to use atomic operations in the case, as two threads will never change the value of the same memory space at the same time. We however have to use __syncthreads() ; as operations of other threads must be over before doing the sum on the next iteration. The result couldn't be reliable otherwise.

-Think in a more abstract way about the reduction algorithm. Does the operation being parallelised need to be commutative?

I would say first that there is no need for the operation to be commutative. The program will work whatever the operation is but in my opinion, a commutative operation will be more reliable, easy to set up (compute) on large datasizes and verify as the results will be reliable.

We could compute a substraction using the reduction :

1 2 3 4

3 5 1 8

(3-5)-(1-8) =-2-7=-9

It is just horrible to setup on large scale.