HPCDA Report - CUDA Track

Andrea Ortenzi
Giuseppe Montanaro

This report aims to describe the work we have done during the development of the project for the HPCDA contest.

Implementation 0 - Standard

The first implementation we have built is based on a simple parallelized version of the standard algorithm. Firstly we convert the matrix representation from COO to CSR in order to optimize the memory occupation and reduce the number of operations during the execution phase. The kernel applies a thread to each row in the matrix and it calculates the PersonalizePageRank and puts the result in the result array. The accuracy of the implementation is 100% and the average execution time is ~1600ms.

Implementation 1 - Float Precision

This implementation is an evolution of the previous one. Basically, we changed the precision of the values from double to float, since CUDA is able to process single precision numbers much faster than double precision ones. The results are quite impressive, the accuracy is still 100% and the average execution time is ~1000ms.

Implementation 2 - Taking Advantage of Shared Memory

This implementation is based on a different approach, it starts from a preprocessing on the COO representation of the graph. We wanted to put the vector ppr in shared memory because it is the most used in the algorithm. Since the vector is too big compared to the dimension of the shared memory (48kB), we decided to split the vector in chunks and put these in shared memory. In order to accomplish this task a block must use only the respective chunk. So, we splitted the matrix in *n* sets of columns that will be managed by *n* GPU blocks. Moreover, these sub elements are as long as a multiple of 32 in length so that a single warp can manage the execution; indeed, read operations from the global memory are done in a batch of 32 memory cells at time. Basically in this way we are reading just one time for each set. The execution phase is basically the same it: calculates the values working in the respective chunks. The results of this implementation are not bad at all: accuracy at 90% and average execution time ~800ms.

Implementation 3 - Kernel Splitting

This implementation comes from the #1. It implements 3 kernels, the first one calculates the value of the dangling factor in the PPR formula, since it is always the same at every iteration, so it can be calculated once and used during the iterations. Moreover, this kernel uses the coalesced access in order to make the threads in the same warp access contiguous space in memory. A second technique used here is the shuffle down that allows to directly update the result calculated by the single warp. The second kernel is almost the same, the only difference is that it calculates another vector that contains the squared difference between the calculated value and the actual one. This vector is used by the last kernel that basically calculates the sum of the element in an array in order to have the total euclidean error. The results of this implementation are encouraging: accuracy still at 100% and average execution time ~800ms.

Implementation 4 - Putting All Together!

The last implementation basically mixes everything we have done till now. The results are definitely interesting: accuracy still at 90% and average execution time ~600ms. So, compared to the standard implementation we have an improvement of ~1000ms.