DSSC - HPC - Parallel Programming

Assignment 4

Patrick Indri

February 6, 2020

Introduction

The aim of this assignment is to solve an exercise concerning CUDA programming. The folder `Ex_6` contains the implementation of the exercise. The required task is to perform a matrix transpose in CUDA.

Exercise 6

The `Ex_6` folder contains a `makefile` to compile the code and the script `run.sh` to compile and run it. It should be stressed that GPU support is required in order to properly run the code.

The two proposed implementations transpose the matrix using block of threads where each block transposes a 32×32 tile. The code accepts the number of threads per block as argument. In particular, a naive and an *efficient* implementations are provided, where the efficient one aims at avoiding large strides across global memory exploiting the GPU shared memory.

The naive implementation performs the transposition simply swapping the indices of the input and output vectors. This results in large strides across global memory, reducing the overall bandwidth for this kernel.

On the other hand, the optimised version exploits shared memory to optimise memory access. Once the 32×32 tile buffer is allocated on the `__shared__` memory, the input matrix can be continuously read and copied into the buffer. Then, the indices are recalculated on the shared memory and the results are written, continuously, on the output matrix. A block-wise barrier synchronisation, put after the input matrix reading, assures that all the threads in the block have completed the reading and the buffer transposition phases before writing to output.

The following matrix summarises the results, showing the bandwidth for different numbers of threads per block for a matrix of size 8192×8192:

Threads per block	Naive bw (GB/s)	Optimised bw (GB/s)
64	16.88	52.15
256	28.27	101.45
512	59.61	97.44
1024	75.59	79.14

The optimised implementation scores consistently better. Moreover, the results are coherent with the recommendation to use thread blocks with fewer threads than elements in a tile, so that each thread transposes multiple matrix elements. The table shows that 256 threads per block (i.e., each thread transposes 4 matrix elements for a tile) provides the best results.