GPU Practical 3: Matrix Transposition

Matrix Transposition

The aim of this practical is how to transpose a matrix optimally on a GPU. At first glance this may seem to be fairly straight forward, but it actually requires a a fair degree of understanding of memory access patterns.

This practical is based on `http://developer.download.nvidia.com/compute/cuda/sdk/website/C/src/transposeNew/doc/MatrixTranspose.pdf` and the techniques found therein.

Starting from a naïve approach, we take into account the following criteria:

- Reducing block size and increasing operations per thread
- Coalesced global read/write
- Shared memory bank conflicts

and demonstrate that in each case they improve the performance of the operation.

Framework

Open `MatrixTranspose.cu`. This contains the framework for this practical. This performs the following steps:

- Create a random $N \times N$ matrix `a` and copy it to the GPU.
- Create a destination matrix `aT`
- Perform and time a simple copy from `a` to `aT` via a kernel
- Perform and time a transpose, storing result in `aT`
- Check the result
- Display the effective bandwidth on the device

The reason for timing a basic copy is that, in an ideal world, a transpose should take the same time (after all, it's just copying values from one array into another). We aim to get as close as possible to the limit suggested by the copy when optimizing the transpose operation.

Improve the copy first

Firstly, notice that the copy kernel only copies a single element at once. This suggests a relatively large overhead of computing indices, launching threads, and so on as compared to the actual memory copy.

Try modifying the kernel (and the calling structure) so that each thread copies 8 elements of the matrix instead. This should be done by dividing the matrix into 8 separate blocks by row. So, we expect that each kernel contains a loop from 0 to 7 such that iteration 0 deals with elements `a[i][j]` where $0 \leq i < N/8$, iterations 1 deals with those s.t. $N/8 \leq i < 2 * N/8$, etc.

Naïve approach

Implement the one-line trivial transpose that comes from switching the indices in both of the copy kernels.

Compile and run the code. You will find that it has very low bandwidth compared to the copy. What is likely to be the problem? (There aren't many possibilities!)

Shared memory

By now, you should have realised that as well as *reading* in a coalesced fashion from global memory, you also need to *write* in a coalesced fashion.

Work out an approach based on splitting the matrix into tiles in shared memory. Use a different thread within the thread-block to write as to read, so that all the loads and stores are coalesced.

Shared memory padding

Depending on how you implemented the use of shared memory in the previous section, you may have done it so that you end up with shared-memory bank-conflicts. Try padding your shared memory so that loads/stores from the same column do not clash.

Other things to try

You will notice that the size of the matrix has been hard-coded at 1024×1024. Try adapting the code for other sizes of matrix, in particular ones that are not integral multiples of the tile size. See whether similar bandwidths can be attained, or whether you need to optimize the algorithm differently depending on the remaining tile-size.

You could also adapt the code for non-square matrices, and for matrix elements other than 4-byte floats.