COMP5811 Parallel and Concurrent Programming: Coursework 3

December 5, 2016

1 Dividing input into thread blocks

The first choice that has to be made is regarding the number of dimensions of the grid and the blocks. Since the program operates on two dimensional images it makes sense for both the grid and blocks to also be two dimensional. The second choice is for specifying how exactly the image is going to be assigned to thread blocks. The way threads operate on the image is that each thread in a block handles one pixel and each block handles a rectangular block of pixels of the image. For processing a rectangular block of the image we set $blockDim.x = blockDim.y = 8/16/32$ because as the problem is laid out we can assume that the width and height of the image is a multiple of 32. The benefits of the different block dimensions are discussed in the last paragraph.

Before explaining the choice for grid dimensions we must note that the input of the program is read from the '.pbm' file format. In that file format the image is laid out row by row. This means that after reading the file the resulting array contains all the rows of the image consecutively. What this means is that if a row of the grid does not handle area on the image as wide as one row then a one dimensional index i would not represent the same x, y coordinates in the image array and in the grid. Thus we always set the $gridDim.x = image.width/blockDim.x$. This is reasonable because we can alway assume that that will not be more that the maximum $gridDim.x$ dimension the GPU supports, considering the size of the image. For small enough images it is possible to set $gridDim.y = image.height/blockDim.x$, so that the grid covers all of the image. For testing on larger images though we cannot cover all of the image with the grid, so we must allow for one thread to manipulate multiple pixels by introducing a stride. For demonstrational purposes the $gridDim.y$ value is set to 1 so that it may shown how the stride works.

2 Optimizations

2.1 Texture memory

Each thread makes use of six neighbouring data cells. This means that multiple memory adresses (close in the 2D grid) are requested more than once from global memory. Thus we make use of the read only Texture Memory that is able to cache memory fetches. More specifically we make use of 2D Texture Memory that is designed to exibit 2D spacial locatily, i.e. memory calls for adjacent addresses in the 2D texture memory grid are cached when they are requested.

2.2 Shared memory

In this problem threads do not interact with one another directly, but caching what each thread has computed in shared memory prior to writing to global memory improves memory coalescing.

2.3 Block Dimension and Bank collisions

It is not possible to completely avoid bank collisions unless we only use 32 threads per block, because each thread writes in a separate element of the shared memory array and there are only 32 banks. Using only 32 threads per bock to avoid all bank collisions results in poor performance due to underutilisation of each block. For any larger number of threads bank collisions are unavoidable simple due to the pigeonhole principle regardless of whether we use int (32 bit) or char (8 bit) or boolean (1 bit) as the data type for the shared memory array.

Choosing the block dimensions to be 32×32 yields a worse running time than using block dimensions of 16×16. That is possibly due to an increased amount of bank collisions. Using 8×8 also results in worse performance, probably due to underutilisation as mentioned in the previous paragraph.