A Parallel Median Filter Algorithm

DVA336 project, part 2.2

Albert Bergström - abm13007@student.mdh.se
Sebastian Lindgren - sln13009@student.mdh.se

January 19, 2017

1 Introduction

In this project we chose a problem in the field of image processing algorithms. We have implemented a median filter algorithm both sequentially and parallel with CUDA. After doing this we studied the speedups we got when we ran the two algorithms. We did this on problems of different sizes, which showed that which is fastest depends on the problem size.

2 Sequential algorithm

Both the sequential and the parallel version starts the same, by reading a windows bitmap image (bmp) and placing the binary read pixel values into a struct called pixel. The struct consists of red, green, blue values and the average value of the three colors. Since bmp stores the colors in reversed order (blue green red) they are switched for easier read.

The sequential filter consists of 3 for loops. The first 2 loops through each pixel(x,y) in the image. The third loop goes through the pixels around the current pixel (the neighbours). If the place checked is outside the image its discarded and the next place is checked. If its inside then the pixel is placed inside a temporary array in a sorted fashion(smallest to biggest). When the whole area of neighbors has been placed in the array the new pixel is chosen. If the array contains an odd number of pixels, the one in the middle is chosen. But if the array contains an even number of pixels, the average value of the two middle pixels is used as the new pixel.

The filter area size that gets checked corresponds to a defined filter size. If the filter size is 1 then the area that gets checked is 3x3. This calculation is done to determine the area: Area size = $(filtersize * 2 + 1)^2$. When each pixel has been calculated the new matrix is returned. It is then saved as Output.bmp for both versions.

3 Parallel algorithm

The parallel version works similar to the sequential one but with a crucial difference. Instead of iterating the width and height it initiate a CUDA kernel with the appropriate amount of threads. By doing it this way the operations for each pixel can be done by the appropriate thread instead of being done in a loop. By dividing the process over many threads only a small piece of the sequential code has to be run on each CUDA core.

To divide the problem we use a 2D grid and 2D block. The 2D block is set as (x:16, y:16). The grid is set by (x:(width+15)/16, y:(height+15)/16). This gives us a good amount threads for each image size. Other than this there is the overhead of the CUDA program in the form of for example allocating memory space on the GPU and transferring data from RAM to the shared graphics memory.

4 Test results

To test what kind of speedup we get we try processing a 500x500 pixel (small image), 1000x1000 pixel (medium image) and 2000x2000 pixel (large image) on both the CPU and GPU. When looking at the difference in time between the sequential and the parallel solution we see that the CUDA solution is several times faster on larger images. Additionally the CUDA solution doesnt suffer as much when increasing the filter size.

- Filter size 1: 3x3-1=8 neighbours
- Filter size 2: 5x5-1=24 neighbours
- Filter size 3: 7x7-1=48 neighbours

Table 1: Sequential solution

Seq	Filter size 1	Filter size 2	Filter size 3
Image size: 500x500	0.062s	0.288s	1.000s
Image size: 1000x1000	0.199s	1.185s	3.689s
Image size: 2000x2000	0.876s	4.224s	13.518s

Table 2: CUDA solution

CUDA	Filter size 1	Filter size 2	Filter size 3
Image size: 500x500	0.312	0.454s	0.536s
Image size: 1000x1000	0.457s	0.595s	1.235s
Image size: 2000x2000	0.740s	1.580s	3.124s

The tests were conducted on a DELL E5550 with an i5-5300U processor and a Nvidia 830M graphics card.

5 Discussion

In the future there are some improvements that could be made to the algorithm. For example could the data structures be modified. Instead of using integers, unsigned chars could be used to reduce the Pixel-array sizes and thus enable slightly bigger images to be processed by the CUDA cores.

There could also be interesting results if we for example ran the filter several times on the output of the filter. Or if we added other filters and combined their effects. This could for example take an image, first smooth it and then perhaps add color inversion or whatever we wish.

Lastly in the discussion we must note that we are very impressed by the speedups available when working with CUDA cores. Problems that could take an unfeasible amount of time on a CPU would become manageable if they are data-parallel enough and done on a GPU. In the future more applications should target the GPU for big and parallel tasks.