Dean Kopping

KPPDEA001

Assignment 1 report

Methods:

Parallelisation algorithm: In order to parallelize the filtering operations for the mean and median filter, I have made use of the fork-join parallelisation algorithm in order to access each pixel in a given window width until all the pixels in the image have been reached and used to determine the mean or median value of the sample window width of pixels.

The algorithm uses a “divide and conquer” approach by recursively forking the given operation into two independent threads until it reaches the sequential cut-off after which the algorithm joins the threads back together. The algorithm is effective as it allows the programme to perform the filtering operations on different sections of the image concurrently and therefor results in a major speed up.

Validation: In order to validate the algorithms, I ran my parallel filters and my sequential filters for a range of different window widths, beginning with 3x3 and ending with 15x15. I completed this for three images of different sizes and recorded the time that was taken to complete the filtering for every one of these runs. After analysing the operation times, it was clear to see the parallel median filter as well as the parallel mean filter were both significantly quicker then their sequential versions for every image at every different window width. I also insured that both mean filters and both median filters produced an identical image and the speed up was not attributed to a difference in output.

Timing: In order to accurately time the filtering operations of all programs I made use of the currentTimeMillis() function this function returns the current time in milliseconds.

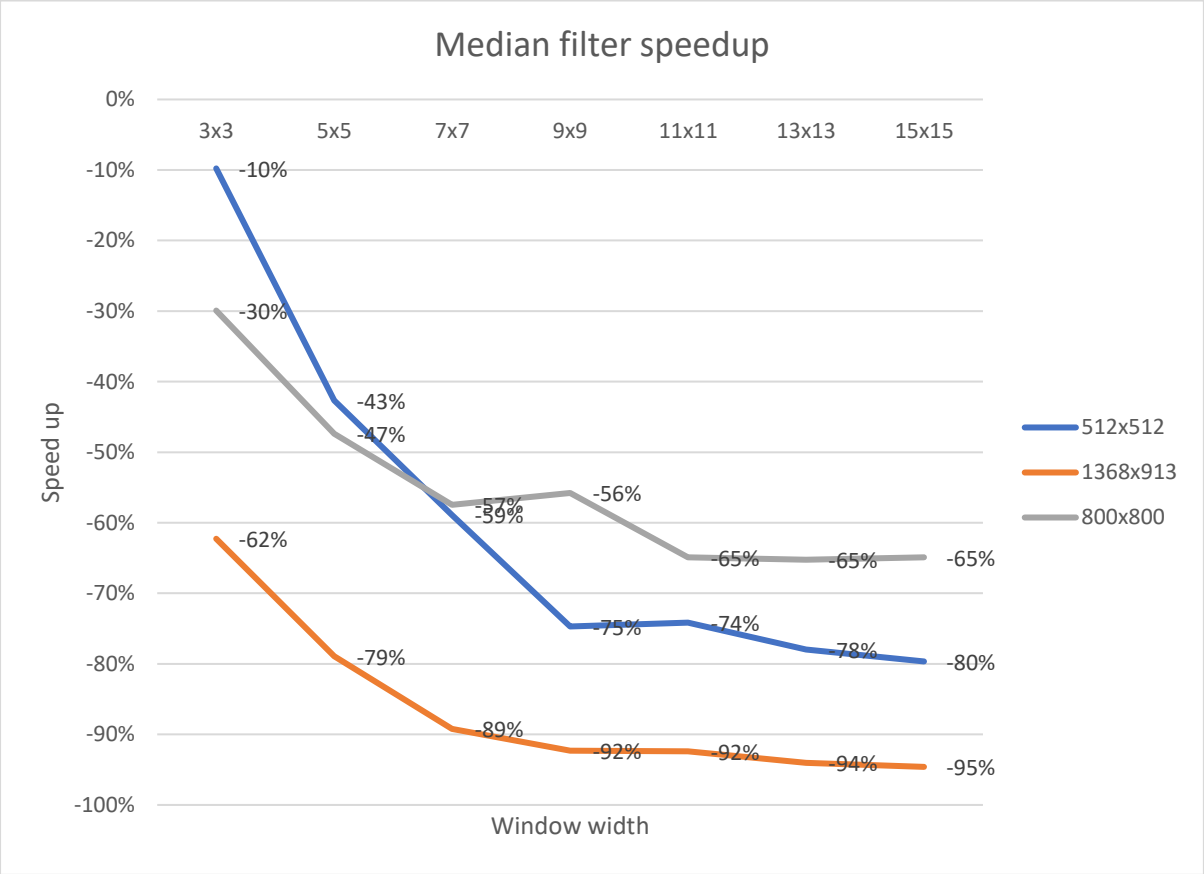
For the sequential programs the function was called before the filtering method and the time was recorded, the function was then called again after the filtering method was completed and the two values were subtracted to determine the time taken to perform the filtering.

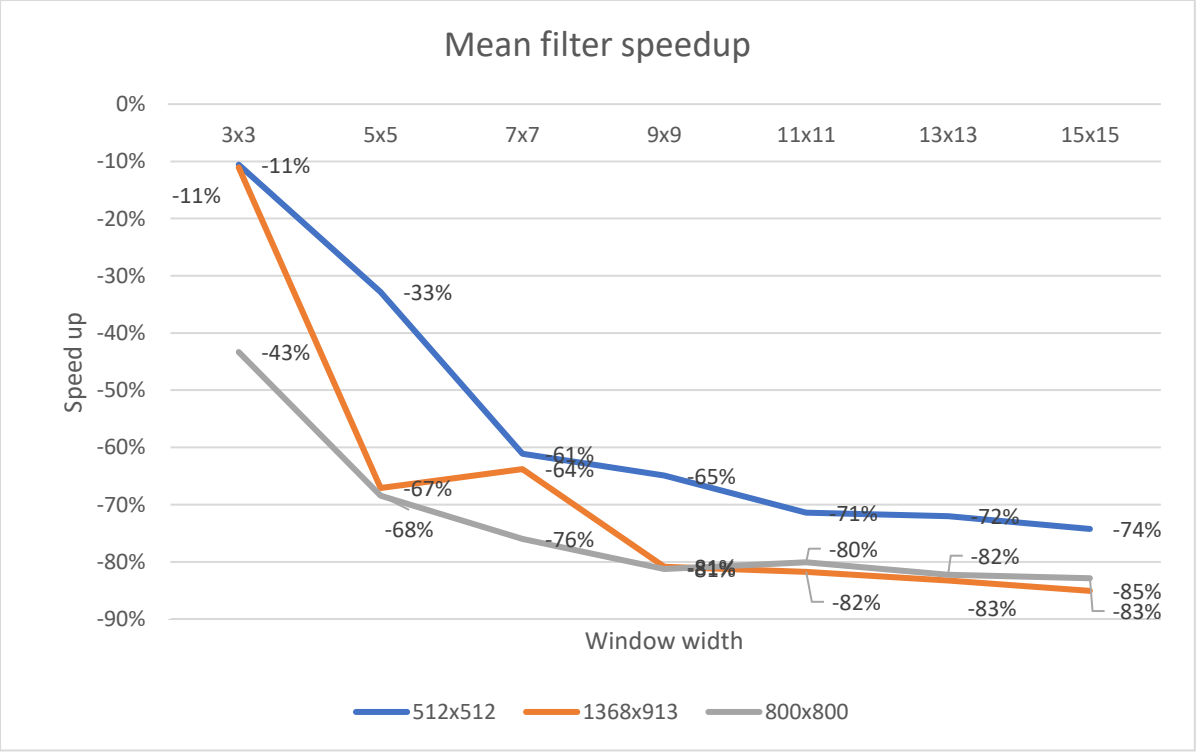
For the parallel programs the currentTimeMillis() function was called before and after the invoke() function and similarly the values were then subtracted.

Optimal serial threshold: For maximum speed up, it is essential to give enough work to each thread but not too much that the program becomes the same as its serial version. It is important not to make too many threads as making threads can become expensive and result in a decline of speed. Therefor the algorithm needs to be told when to stop forking the threads and begin joining them. In order to find an optimal serial threshold, I manually ran my programs multiple times with different cutoffs, starting with a very large cut off and a very small cut off and eventually narrowing it down to a value which made my program optimal and resulted in a good speed up from the serial programs.

Machine architecture: Both machines used in testing had four cores and produced very similar results.

Results:


Median filter speedup
0%
-10%
-20%
-30%
-40%
-50%
-60%
-70%
-80%
-90%
-100%
Speed up
3x3
5x5
7x7
9x9
11x11
13x13
15x15
-10%
-43%
-75%
-74%
-80%
-30%
-47%
-57%
-59%
-56%
-65%
-65%
-65%
-62%
-79%
-89%
-92%
-92%
-94%
-95%
512x512
1368x913
800x800
Window width


Mean filter speedup
0%
-10%
-20%
-30%
-40%
-50%
-60%
-70%
-80%
-90%
Speed up
3x3
5x5
7x7
9x9
11x11
13x13
15x15
-11%
-11%
-33%
-61%
-64%
-65%
-71%
-72%
-74%
-43%
-67%
-68%
-76%
-81%
-80%
-82%
-82%
-83%
-85%
-83%
Window width
512x512
1368x913
800x800

Sequential cut off: The optimal sequential cut off for both parallel algorithms was 80. This was the value that was found to result in the best speed up for both algorithms.

The best performing data set size was found to be the largest image which was 1368x913. For the parallel mean filter the algorithm saw a maximum speed up of 85% with a window width of 15x15. For the parallel median filter, the same image saw a speed up of 95% with a 15x15 window width which was the maximum speed up. The parallel programs however do perform well for all image sizes and window widths tested with the best performance happening with the largest window widths.

The parallel algorithms are very similar and both have some unexplainable anomalies. For the median parallel algorithm on the smaller window widths, the speed up is ranked from smallest image to largest however as the window width increases this becomes less predictable. The smallest image and the second smallest image trade places in speed up percentage. The largest image however is consistently seeing the best results.

The mean parallel filter algorithm is generally less predictable and there is an anomaly for the window width of 7x7. For some reason at this window width the speedup decreased and we see a sharp peak in the graph. The best speed up for both graphs is seen when the window width is large, and the image is large.

For a machine with four cores the best possible speed up would be four times faster, this however is never usually the case as creating threads has a cost.

Conclusion:

In conclusion, it is worth using parallelization to tackle this problem in java.

For a machine with four cores the best possible speed up would be four times faster, this however is never usually the case as creating threads has a cost. Although the best speed up seen from the algorithms was 95% it is definitely worth using parallelization as the programs which used it were consistently faster than their sequential counterparts.

When an optimal sequential cut off is found the programs speed up is highly noticeable and can save time and resources on the machine.