CS420 Project1

Bohao Tang

February 20, 2018

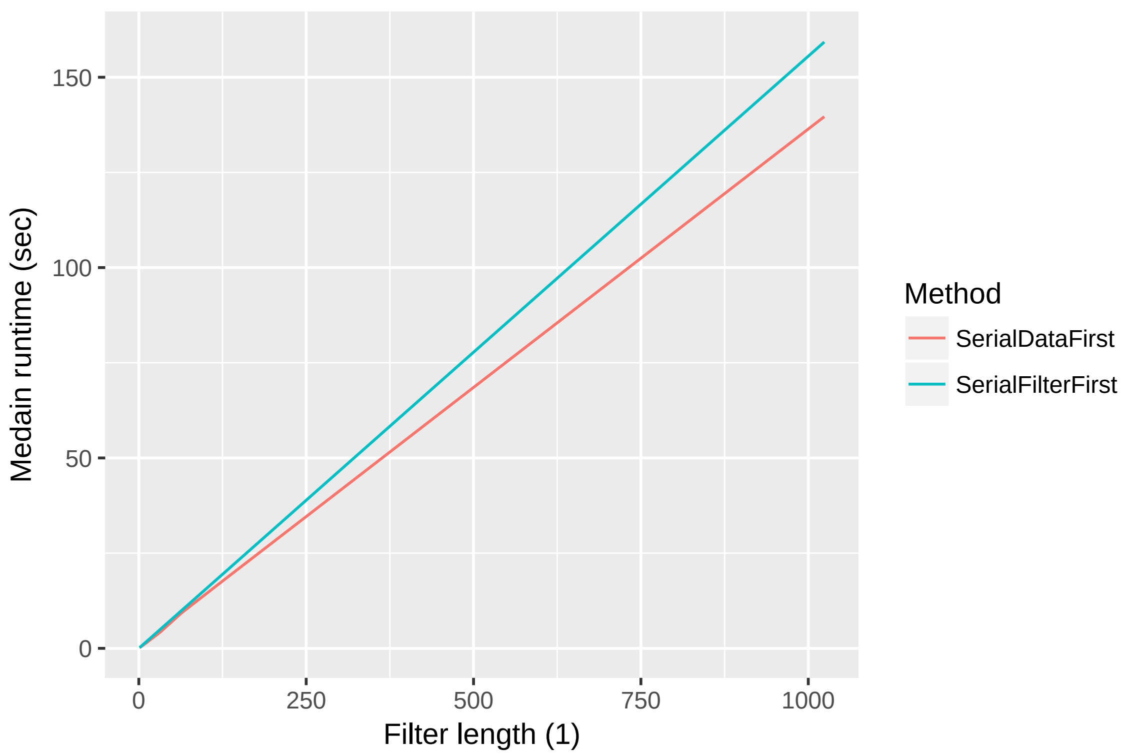
Introduction

Here's the writing part of project 1, you can find the original data used in this assignment in `https://github.com/bhtang127/cs420-parallel/tree/master/project1/data`. The source code is handed in and I ran it 20 times in aws c5.2xlarge. I choose the median value of the 20 runs to do the plot and anlysis, and the standard deviation for each data is approximatly between 0.1 and 0.2.

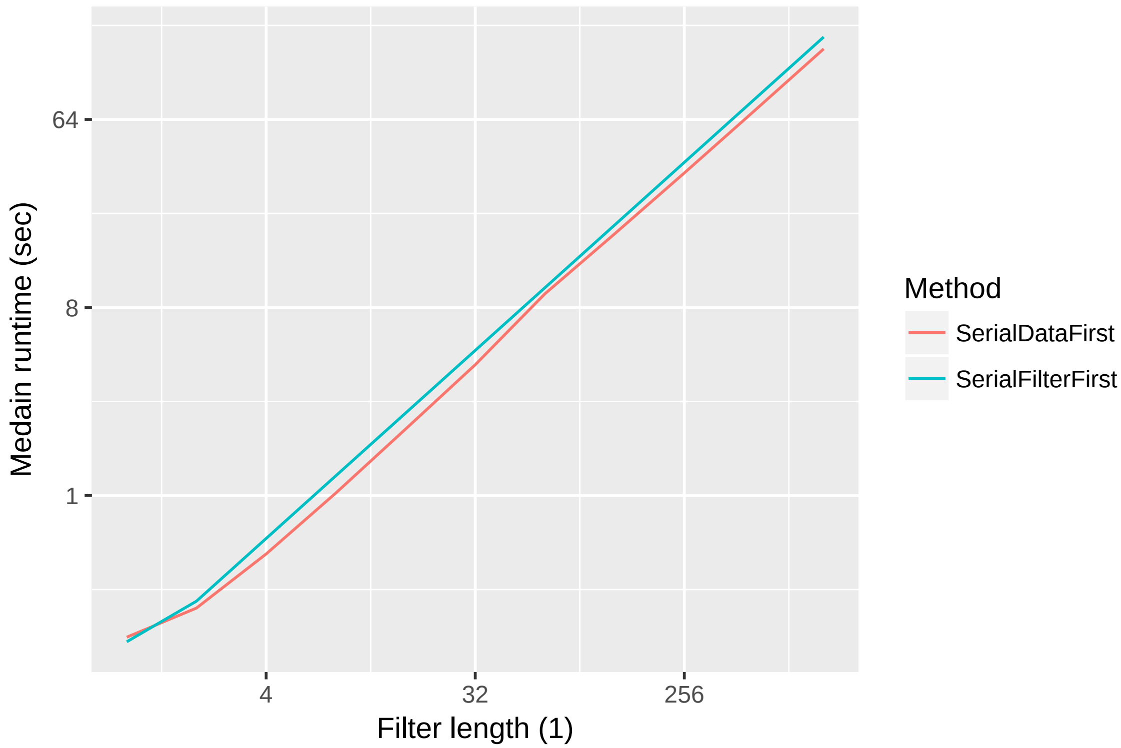
Loop Efficiency

a.

Here is the plot of runtime vs filter size:



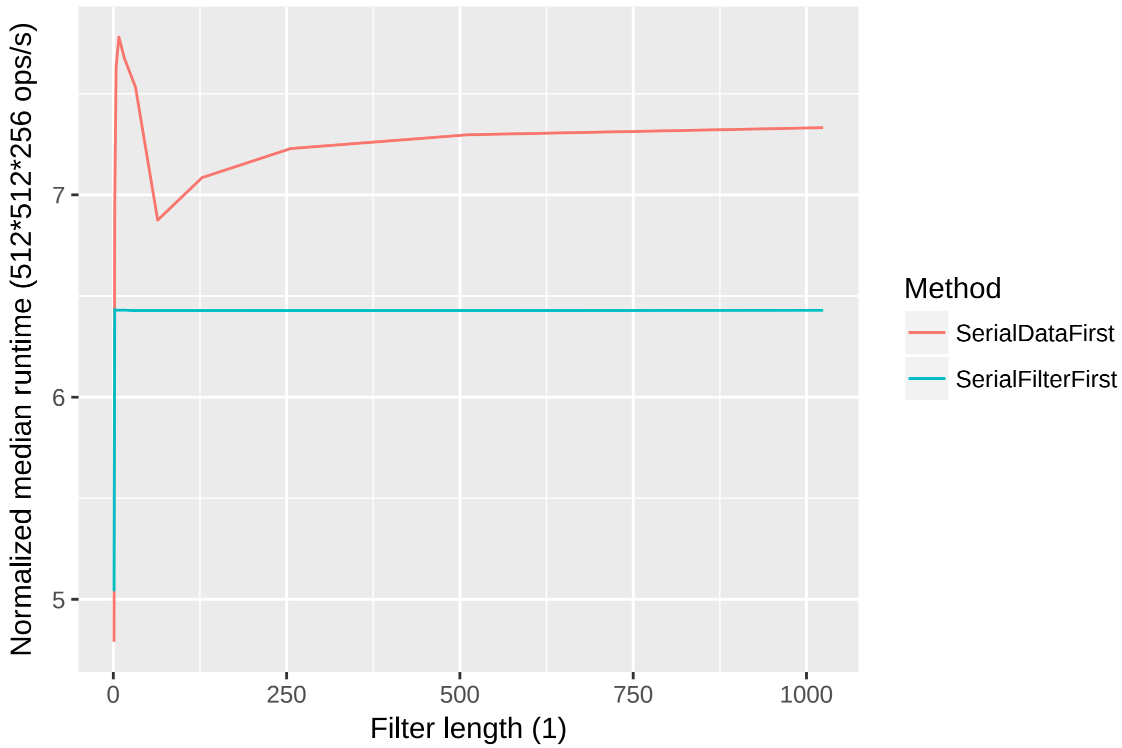
And here a log-log version plot of it:



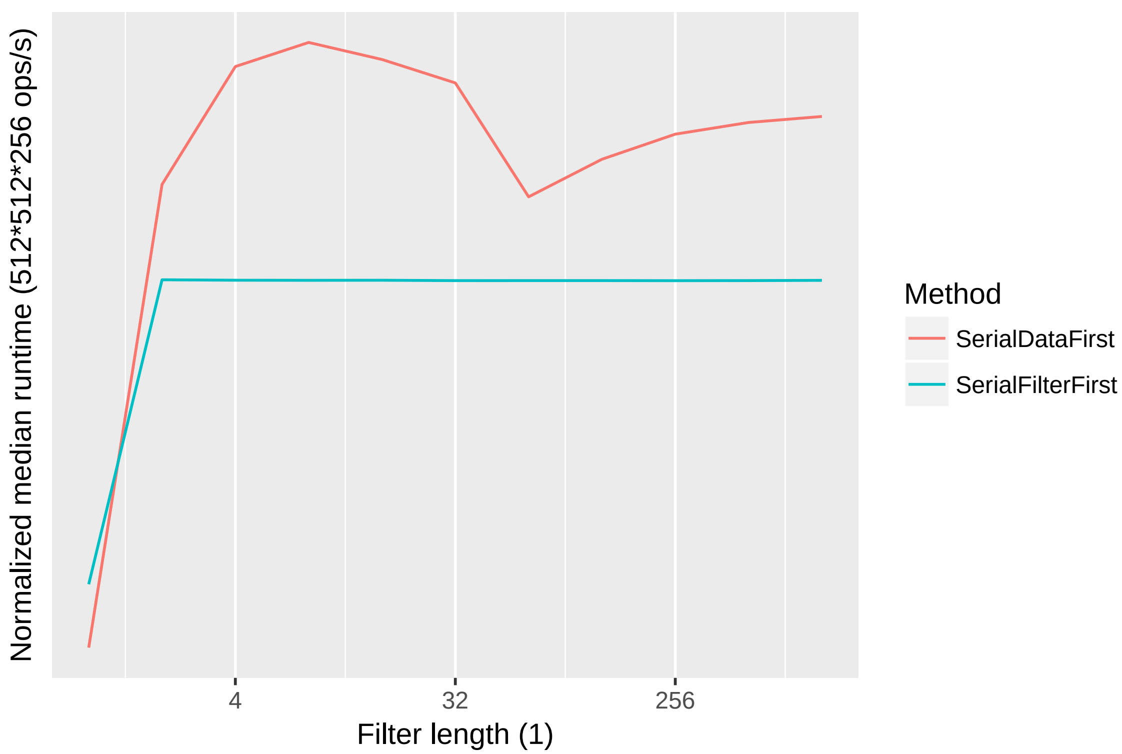
We can see that when filter length bigger than 1, function `serialDataFirst` costs less time.

b.

In the implemented functions, total number of operations is approximatly `filter_length * data_length`. Since `data_length` doesn't change, we normalized the runtime by `filter_length / runtime`. Then here is the plot of normalized runtime vs filter length.



And here is a log-log version plot:



We can see here that when `filter length` > 1, function `serialDataFirst` can run more operations per second.

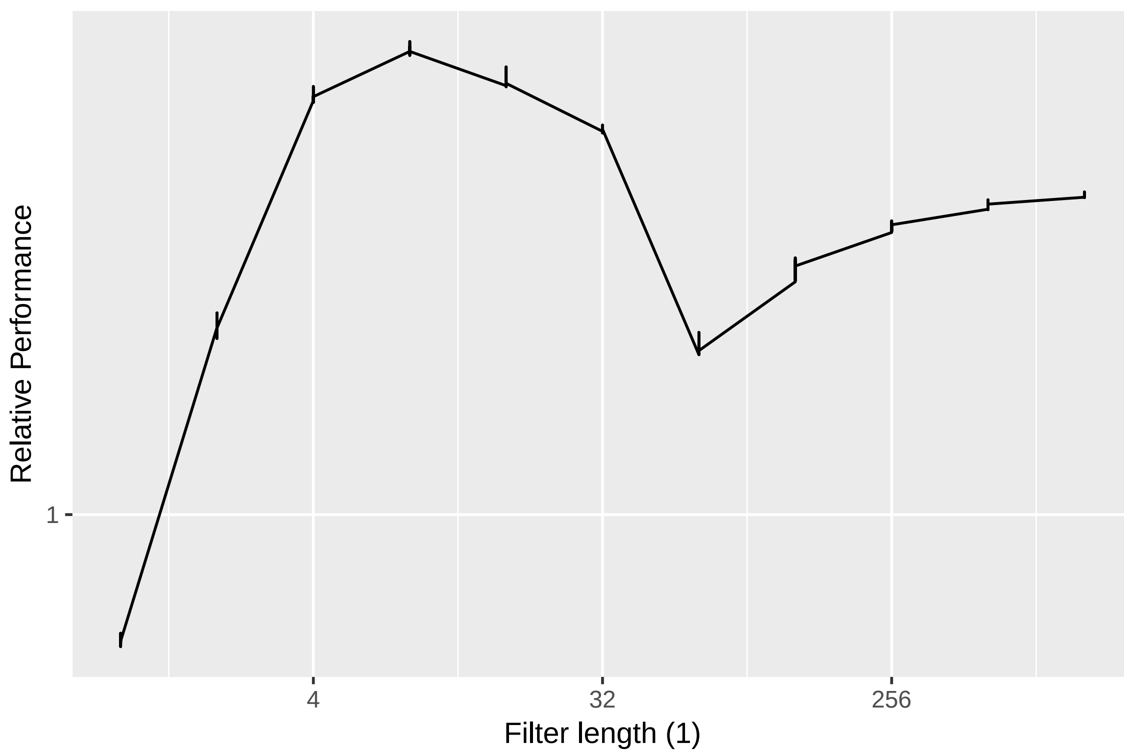
c.

It is more efficient to have data in the outer loop. Because here `data length` is very big `512*512*256` and `filter length` is relative tiny at most `1024`. So it is likely that when you fix a filter element and scan the data, data can not be totally read into memory(cache) meanwhile the filter can when you fix a data point and scan the filter. That will cause a different operation speed and a waste of time from I/O. So putting data out side will be faster.

But when filter size is equal to 1, serialFilterFirst is faster. This is because of the loop overhead. When you put data outside you will keep jumping statment between inner and outer loop, and this will slow down the program.

d.

We divide time of `serialFilterFirst` by that of `serialDataFirst` to get the relative performance, and the log-log plot is like below.



The trend is complex, we can say that the relative performance is grossly follow an oscillation decline. As `filter length` goes up, it will be more and more near to `data length` and therefore we can suppose relative performance tend to be 1, which creates a trend of decline. But locally, when you increase `filter length` by 1, the total jump of statments between inner and outer loop increased by 1 for `serialFilterFirst` vs 0 for `serialDataFirst`. This create a locally increase trend of relative performance. So the total trend is an oscillation decline.

Loop Parallelism

1.

a.

See the code

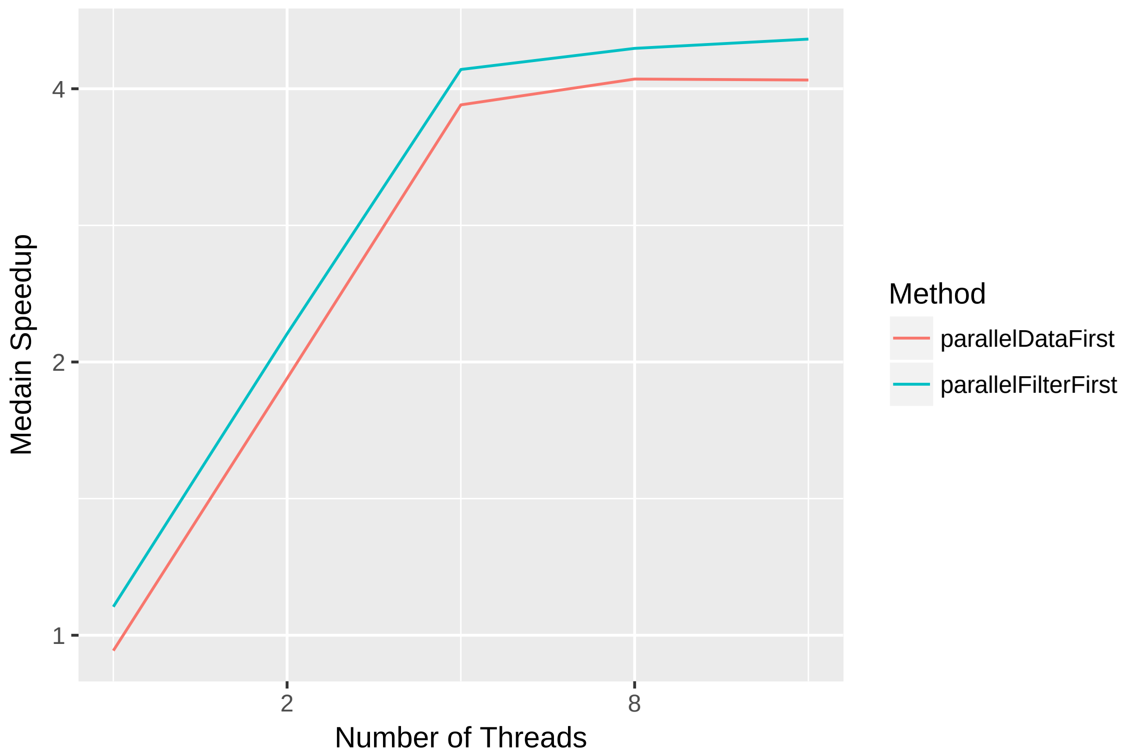
b.

See the code

2.

a.

Here is the plot for speed up.



b.

Here's the data for absolute runtime:

```
##   num_threads parallelFilterFirst parallelDataFirst
## 1           1            74.09158          72.95728
## 2           2            37.07546          36.56992
```

```
## 3             4              18.96089             18.27166
## 4             8              17.97234             17.11399
## 5            16              17.55461             17.15548
```

The speedup increase approximatly linear before threads 4, slightly increase between 4 and 8, and tailed off after 8. This is strong scaling because the total problem size never change. This program is run on c5.2xlarge which has 8 cores, so ideally speedup curve will increase linearly up to 8 threads because it has 8 independent computing units and then curve will be sharply tailed off.

3.

a.

For the absolute time, `parallelDataFirst` is still faster because the problem mentioned above(Loop Efficiency 1.c.) still holds. But as the plot above, `parallelFilterFirst` have a higher speedup, maybe this is because filter size is small and when you devide them you will reduce the jumps of statment more than deviding larger data. Or because that filter is small, so deviding them will make them able to be held in higher cache and therefore higher speed.

b.

We use data from `parallelDataFirst` to estimate `p`. Here is the speedup data for num threads < 8:

```
##   num_threads            Method Median_Speedup
## 1           1 parallelDataFirst      0.9616693
## 2           2 parallelDataFirst      1.9185380
## 3           4 parallelDataFirst      3.8398678
## 4           8 parallelDataFirst      4.0996148
```

Then we use a Least Square method to decide the `p`, the we got `p = 0.92238`.

```r
p = sum((1-1/Median_Speedup)*(1-1/num_threads)) / sum((1-1/num_threads)^2)
p
```

```
## [1] 0.92238
```

c.

The non-ideal speedup is mainly due to the startup cost. Because here the loop carrier dependencies don't exist and when we divide the outer loop evenly, the smaller tasks will be approximatly of same size (and are independent to each other). So the interference and stew effect are tiny in this problem.

An Optimized Version

a.

Here's the runtime[sec] data for loop unrolling type of funcion (`LUFilter1st` and `LUData1st`), num_threads=16, filter_length=512, you can compare them to data frame above.

```
##   LUFilterFirst LUDataFirst
## 1      12.39503    12.25481
```

We can see a significant speedup (about 5 seconds faster). This is because when we do loop unrolling, we significantly reduce the total amount of jump statments between loops, also this will reduce the startup cost since we now have a less outer loop.

b.

Here's the runtime[sec] data for dynamic scheduling type of funcion (`DynamicFilter1st` and `DynamicData1st`), num_threads=16, filter_length=512, you can compare them to data frame above.

```
##   DynamicFilterFirst DynamicDataFirst
## 1           17.51156         17.55338
```

We can see that there's no obvious speedup and a tiny slowdown for `DynamicData1st`. This is because in this problem, we don't have many influence from interference and skew, so dynamicly divide the task will not make much difference, only a tiny increase of startup cost. And since data length is far more large than filter length, startup cost for `DynamicData1st` will increase much more.

Then here's the runtime[sec] data for static scheduling type of funcion (`StaticFilter1st` and `StaticData1st`), num_threads = 16, filter_length = 512, block size = 128, you can also compare them to data frame above.

```
##   StaticFilterFirst StaticDataFirst
## 1          26.46272        17.29101
```

We can see that there's no obvious speedup or slowdown for `StaticData1st`, and an obvious slowdown for `StaticFilter1st`. This is because the block size is 128 and the filter length is 512, so when you put filter in the outer loop, this scheduling can only divide the task into 4 pieces, which is inefficient in a thread number of 16. But since the data length is very large, there will be no influence if you put data in the outer loop.