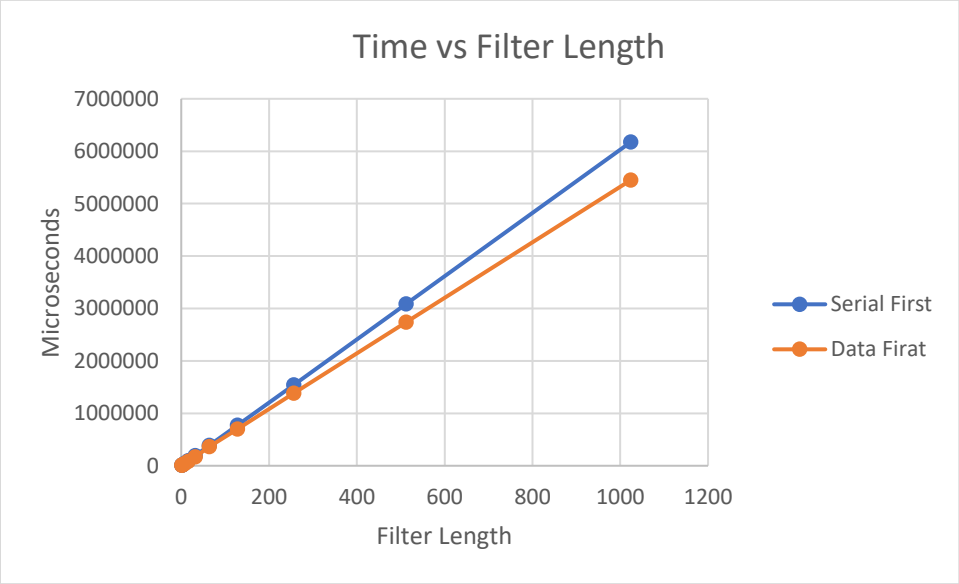
Jason Zhang

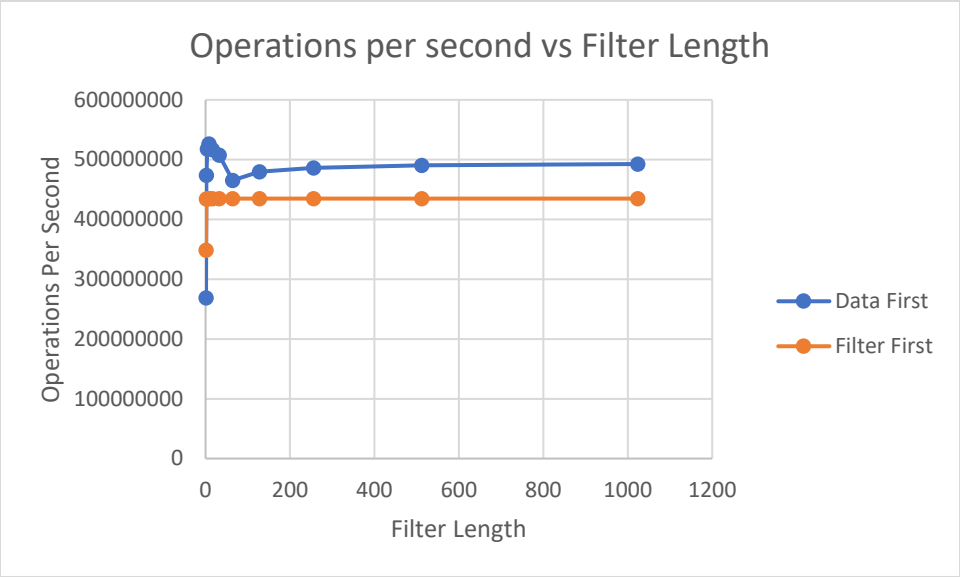
Loop Efficiency

1

a.



b.


Operations per second vs Filter Length
Operations Per Second
600000000
500000000
400000000
300000000
200000000
100000000
0
0
200
400
600
800
1000
1200
Filter Length
Data First
Filter First

c.

It is more efficient to have data first in the outer loop because the filter size is much smaller than the data length and thus it can fit in smaller cache. This also means that we will experience far less cache misses with the filter in the inner loop as opposed to data in the inner loop, and therefore we will waste less cycles trying to fetch data from memory.

d.

The trend seems to increase drastically in terms of operations per second as we increase through the filters from 0 to 64. However larger filter lengths seem to have a level off between 400000000 and 5000000000 operations per second and this turns our drastic increase into a constant number of operations. This is most likely due to hardware restrictions in that the CPU can only handle so many operations per second. Thus once we reached the upper limit for operations per second, the graph naturally leveled off as we increased our filter length.

Loop Parallelism

1.

a.

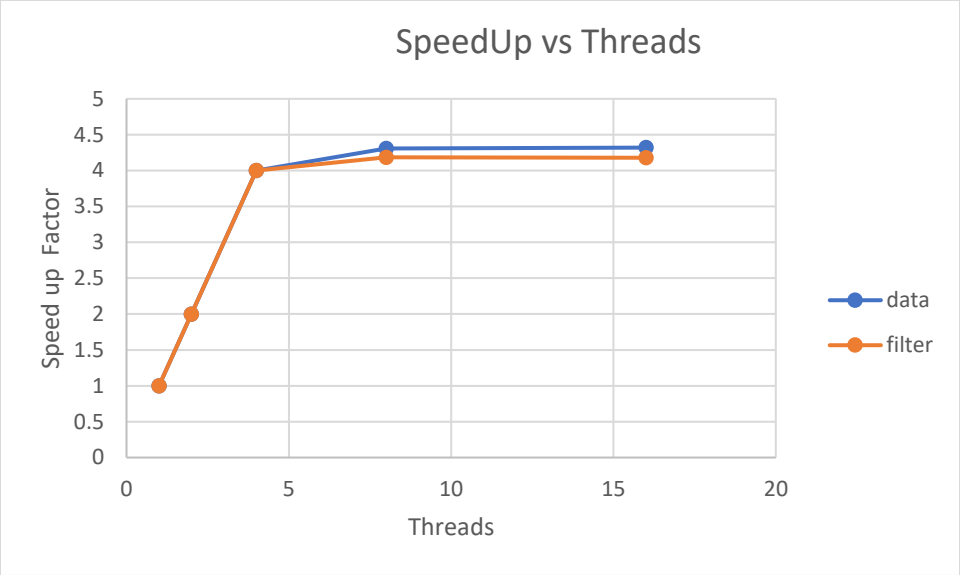
```
void parallelFilterFirst ( int data_len, unsigned int* input_array, unsigned int*
output_array, int filter_len, unsigned int* filter_list )
{
    /* Variables for timing */
  struct timeval ta, tb, tresult;

  /* get initial time */
  gettimeofday ( &ta, NULL );
  #pragma omp parallel for
  /* for all elements in the filter */
  for (int y=0; y<filter_len; y++) {
    /* for all elements in the data */
    for (int x=0; x<data_len; x++) {

      if (input_array[x] == filter_list[y]) {

        output_array[x] = input_array[x];
      }
    }
  }
  gettimeofday ( &tb, NULL );

  timeval_subtract ( &tresult, &tb, &ta );

  printf ("Parallel filter, %lu, %d, %d\n", tresult.tv_sec*1000000 +
tresult.tv_usec, filter_len, omp_get_max_threads());
}
```

b.

```
/* Function to apply the filter with the filter list in the outside loop */
void parallelDataFirst ( int data_len, unsigned int* input_array, unsigned int*
output_array, int filter_len, unsigned int* filter_list )
{
    /* Variables for timing */
  struct timeval ta, tb, tresult;

  /* get initial time */
  gettimeofday ( &ta, NULL );
  #pragma omp parallel for //schedule(dynamic, 4)
  /* for all elements in the data */
  for (int x=0; x<data_len; x++) {
    /* for all elements in the filter */
    for (int y=0; y<filter_len; y++) {

      if (input_array[x] == filter_list[y]) {
        output_array[x] = input_array[x];
      }
    }
  }

  gettimeofday ( &tb, NULL );

  timeval_subtract ( &tresult, &tb, &ta );

  printf ("Parallel data, %lu, %d, %d\n", tresult.tv_sec*1000000 +
tresult.tv_usec, filter_len, omp_get_max_threads());
}
```

2.

a.

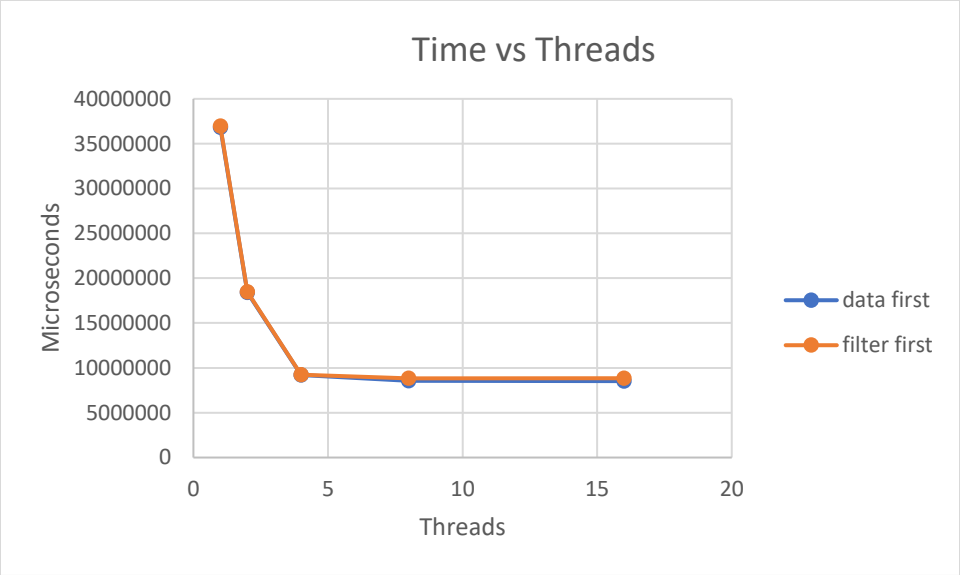

SpeedUp vs Threads
Speed up Factor
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
0
5
10
15
20
Threads
data
filter

b.

From our speedup plots we can see that until about 8 threads, the more threads we used, the faster our programs were. However, after 8 threads, our speedup tailed off. This is due to the fact that we are not utilizing more threads to speed up our program and 8 threads is our maximum. This makes sense because we were using a c5.2xlarge instance which only has 8 virtual CPU cores for us to use. This means that we will be cap to how many threads we can run in parallel and in this case 8 threads is about our max.

3.

a.



From this plot you can see that data first is slightly faster than filter first. This is most likely due to the fact that our data length is much larger than our filter length. Thus by having our data be the outer loop and filter in our inner loop, we are able to fit our filter length in a smaller cache size which means we will not experience as many cache misses and won't have to waste cycles trying to retrieve data from memory. If we use data in our inner loop, we won't be able to fit such a large size into cache and thus we will experience more cache misses and will have to waste more time fetching data which means the program will take longer to execute.

b.

p values for Data first: (using equation (S = 1/((1-p) + (p/s))))

threads	p value
1	0
2	1.000001
4	0.999815
8	0.877228

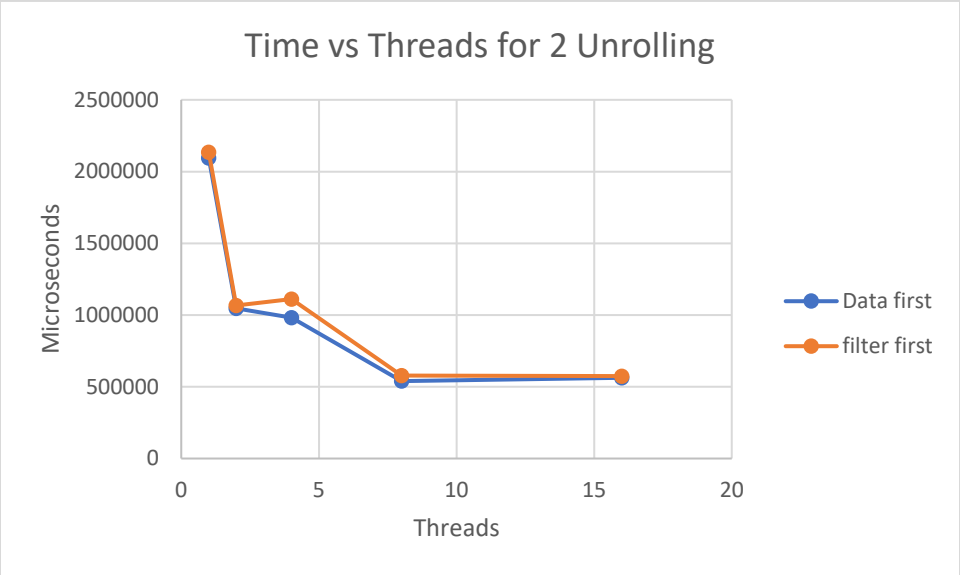
c.

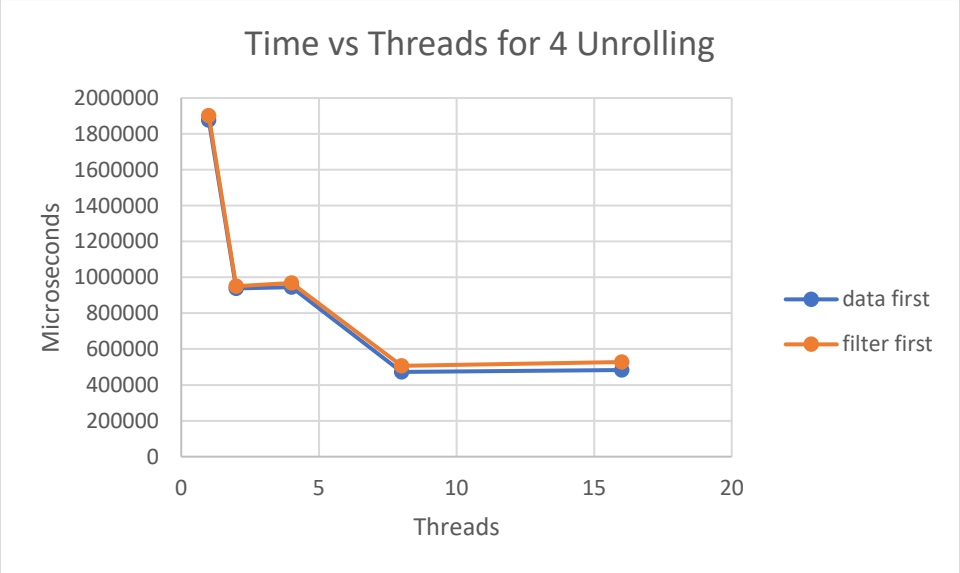
I think interference and start up costs attribute the most to the non-ideal speedup. This is because multiple threads are accessing the same data and there might be issues when multiple threads try to read/write to that same data. This will cause hang ups and delay the running time. Startup costs are negligible. Skew won't play as much of a role since each thread runs independently of the other threads.

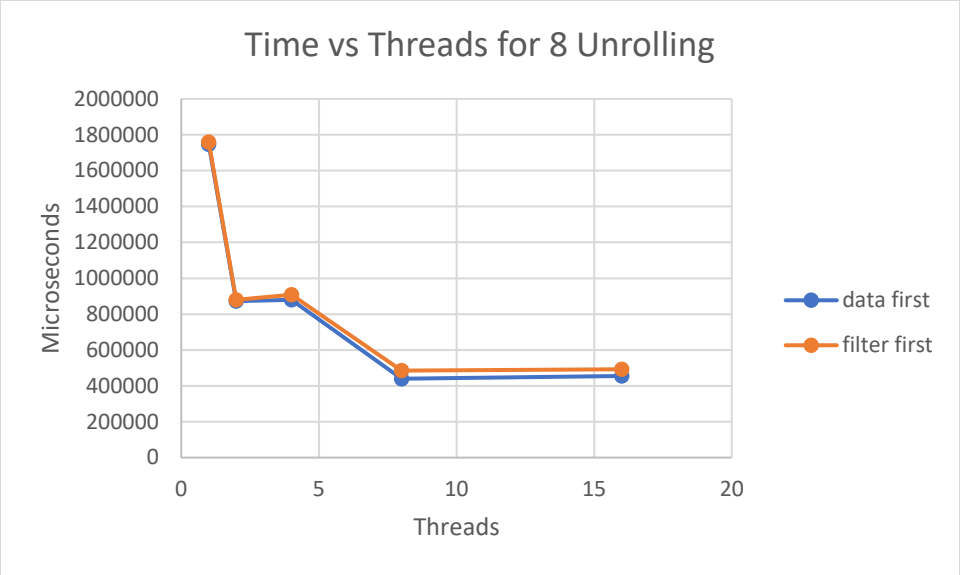
An Optimized Version

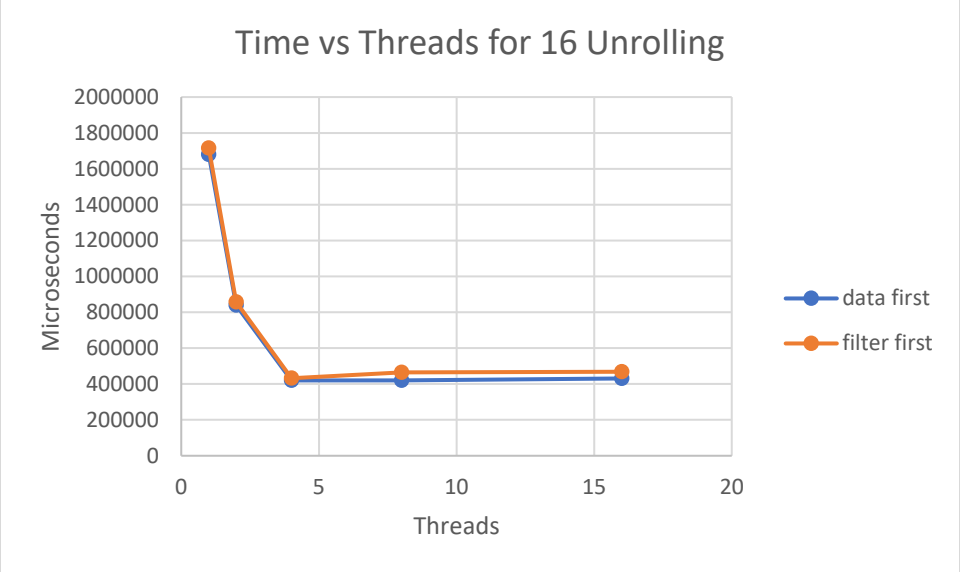
1.

a.





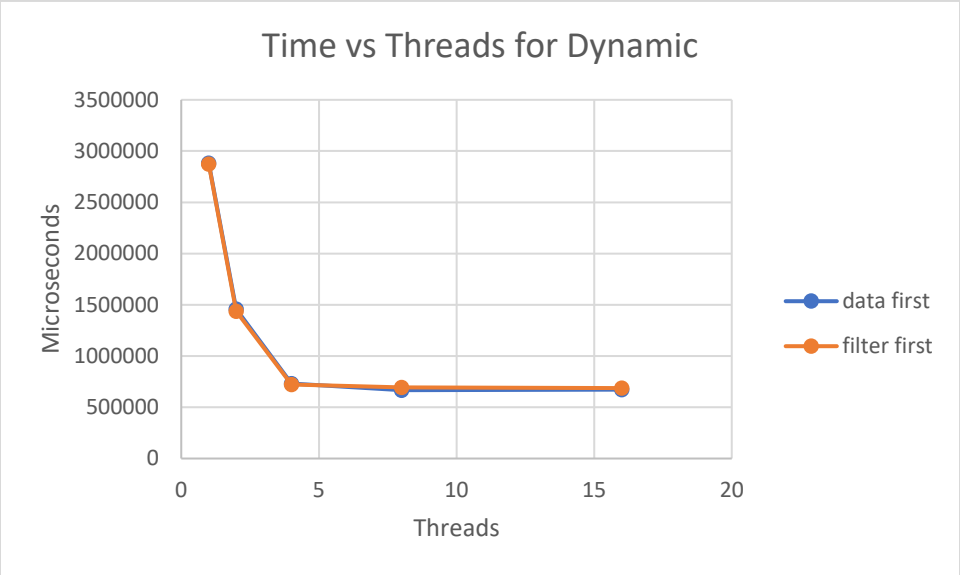


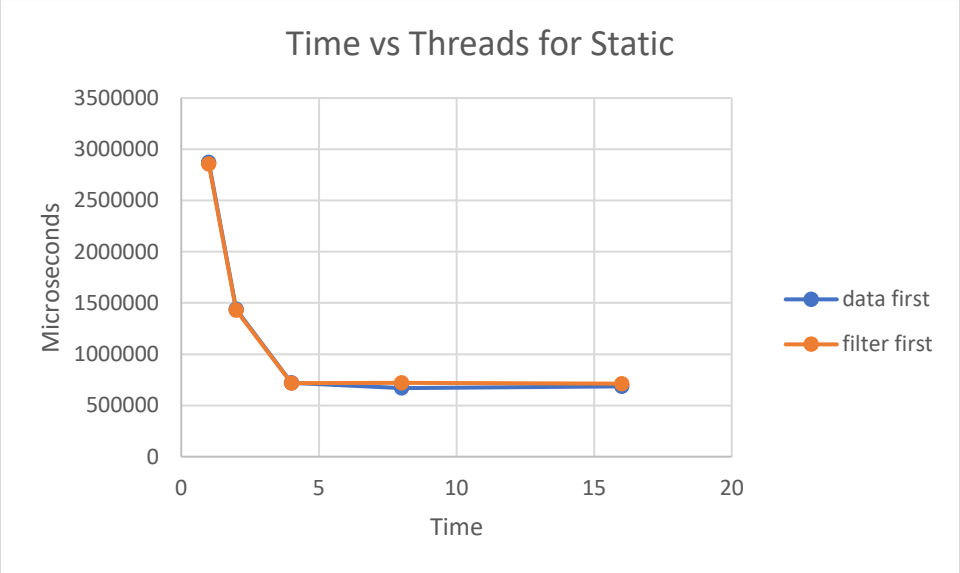


I see improvement with each unrolling. With a higher unrolling level my total runtime is reduced by seconds. The run time is improved because with more unrolling we begin to eliminate the need for loop checking and branch predictions. The overhead associated with wrong branch predictions is eliminated and therefore we get a faster runtime.

b.

Both use a chunk size of 4





In comparison to our parallel run from earlier there seems to be no improvement when using scheduling. For both static and dynamic it looks like the run times are similar to the unscheduled parallel runs. For static it will try to evenly divide the number of loop runs and schedule them to available threads. In this case, when we run static it is similar to just running our for loop sequentially because each thread will wait until the previous thread finishes its loop iterations before it begins its own. Therefore it will take about the same time as an unscheduled one. Dynamic scheduling will also do much of the same, since our loop depends on the previous iterations to finish before continuing. The threads

that have some loop iterations assigned to it that come after the current thread that is running its loop iterations, will remain idle, thus we will not get any benefits from the scheduling.