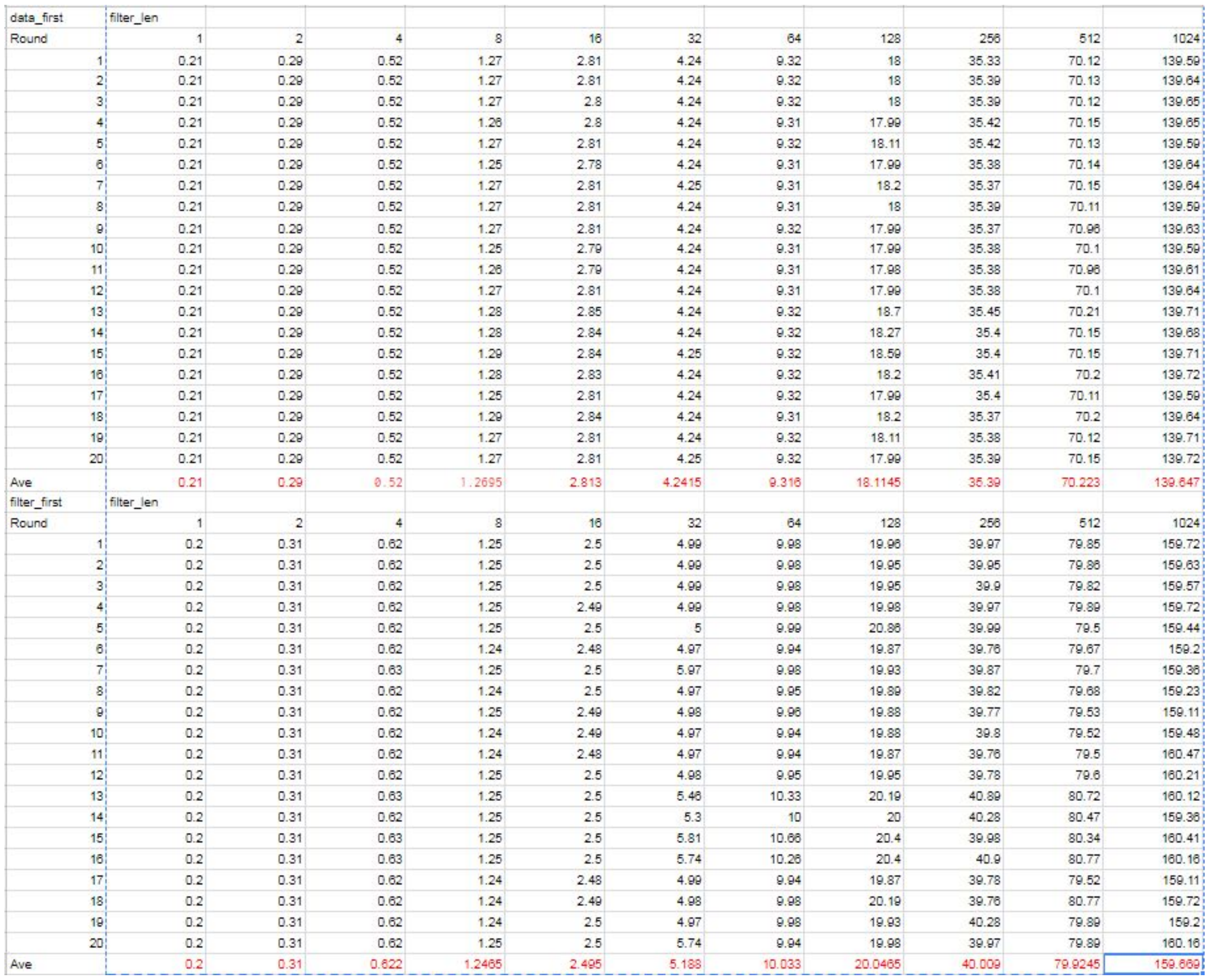
Parallel Programming
Assignment #2: OpenMP Filter

Youzhe Dou

Part 1: Loop efficiency

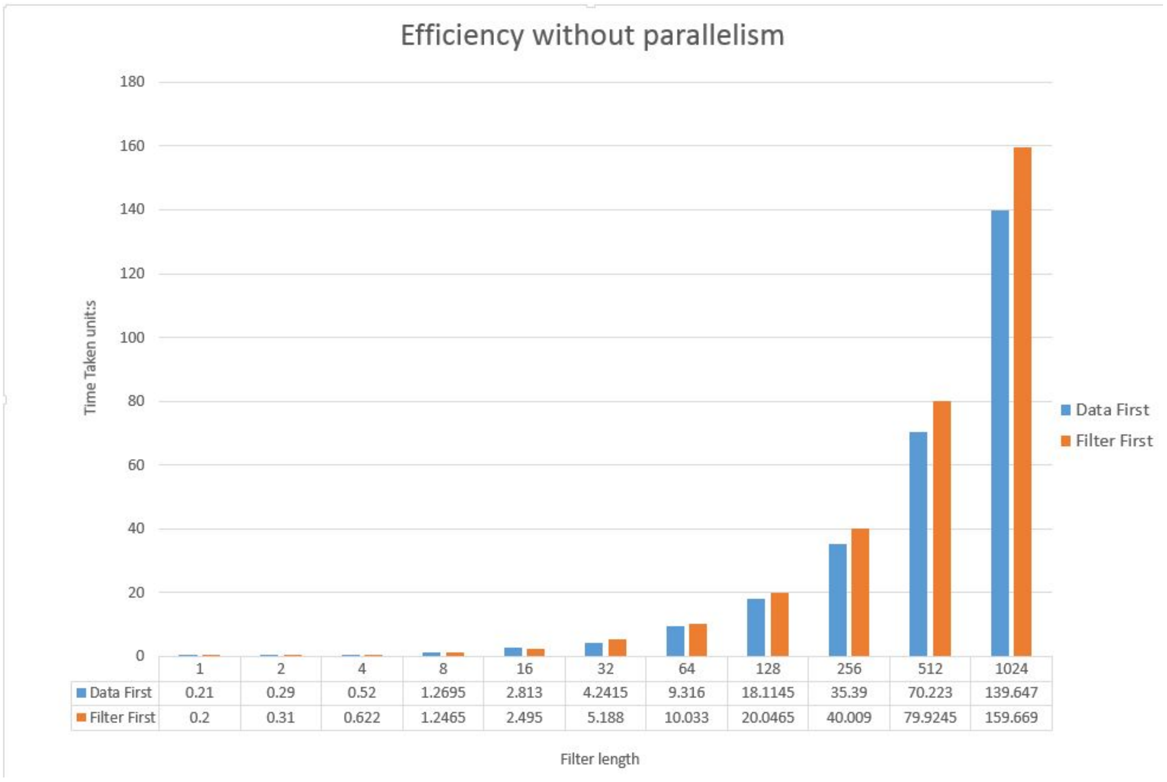
1. Run SerialFilterFirst() and SerialDataFirst() with filter_length (from 1 to 1024) 20 times. Remove outliers and get the averages.

DATA_LENGTH = 512*512*256
FILTER_LENGTH = (1, 2, 4, 8, 16, 32, 64, 128, 512, 1024)
Time unit: sec



data_first	filter_len										
Round	1	2	4	8	16	32	64	128	256	512	1024
1	0.21	0.29	0.52	1.27	2.81	4.24	9.32	18	35.33	70.12	139.59
2	0.21	0.29	0.52	1.27	2.81	4.24	9.32	18	35.39	70.13	139.64
3	0.21	0.29	0.52	1.27	2.8	4.24	9.32	18	35.39	70.12	139.65
4	0.21	0.29	0.52	1.26	2.8	4.24	9.31	17.99	35.42	70.15	139.65
5	0.21	0.29	0.52	1.27	2.81	4.24	9.32	18.11	35.42	70.13	139.59
6	0.21	0.29	0.52	1.25	2.78	4.24	9.31	17.99	35.38	70.14	139.64
7	0.21	0.29	0.52	1.27	2.81	4.25	9.31	18.2	35.37	70.15	139.64
8	0.21	0.29	0.52	1.27	2.81	4.24	9.31	18	35.39	70.11	139.59
9	0.21	0.29	0.52	1.27	2.81	4.24	9.32	17.99	35.37	70.96	139.63
10	0.21	0.29	0.52	1.25	2.79	4.24	9.31	17.99	35.38	70.1	139.59
11	0.21	0.29	0.52	1.26	2.79	4.24	9.31	17.98	35.38	70.96	139.61
12	0.21	0.29	0.52	1.27	2.81	4.24	9.31	17.99	35.38	70.1	139.64
13	0.21	0.29	0.52	1.28	2.85	4.24	9.32	18.7	35.45	70.21	139.71
14	0.21	0.29	0.52	1.28	2.84	4.24	9.32	18.27	35.4	70.15	139.68
15	0.21	0.29	0.52	1.29	2.84	4.25	9.32	18.59	35.4	70.15	139.71
16	0.21	0.29	0.52	1.28	2.83	4.24	9.32	18.2	35.41	70.2	139.72
17	0.21	0.29	0.52	1.25	2.81	4.24	9.32	17.99	35.4	70.11	139.59
18	0.21	0.29	0.52	1.29	2.84	4.24	9.31	18.2	35.37	70.2	139.64
19	0.21	0.29	0.52	1.27	2.81	4.24	9.32	18.11	35.38	70.12	139.71
20	0.21	0.29	0.52	1.27	2.81	4.25	9.32	17.99	35.39	70.15	139.72
Ave	0.21	0.29	0.52	1.2695	2.813	4.2415	9.316	18.1145	35.39	70.223	139.647
filter_first	filter_len										
Round	1	2	4	8	16	32	64	128	256	512	1024
1	0.2	0.31	0.62	1.25	2.5	4.99	9.98	19.96	39.97	79.85	159.72
2	0.2	0.31	0.62	1.25	2.5	4.99	9.98	19.95	39.95	79.86	159.63
3	0.2	0.31	0.62	1.25	2.5	4.99	9.98	19.95	39.9	79.82	159.57
4	0.2	0.31	0.62	1.25	2.49	4.99	9.98	19.98	39.97	79.89	159.72
5	0.2	0.31	0.62	1.25	2.5	5	9.99	20.86	39.99	79.5	159.44
6	0.2	0.31	0.62	1.24	2.48	4.97	9.94	19.87	39.76	79.67	159.2
7	0.2	0.31	0.63	1.25	2.5	5.97	9.98	19.93	39.87	79.7	159.36
8	0.2	0.31	0.62	1.24	2.5	4.97	9.95	19.89	39.82	79.68	159.23
9	0.2	0.31	0.62	1.25	2.49	4.98	9.96	19.88	39.77	79.53	159.11
10	0.2	0.31	0.62	1.24	2.49	4.97	9.94	19.88	39.8	79.52	159.48
11	0.2	0.31	0.62	1.24	2.48	4.97	9.94	19.87	39.76	79.5	160.47
12	0.2	0.31	0.62	1.25	2.5	4.98	9.95	19.95	39.78	79.6	160.21
13	0.2	0.31	0.63	1.25	2.5	5.46	10.33	20.19	40.89	80.72	160.12
14	0.2	0.31	0.62	1.25	2.5	5.3	10	20	40.28	80.47	159.36
15	0.2	0.31	0.63	1.25	2.5	5.81	10.66	20.4	39.98	80.34	160.41
16	0.2	0.31	0.63	1.25	2.5	5.74	10.26	20.4	40.9	80.77	160.16
17	0.2	0.31	0.62	1.24	2.48	4.99	9.94	19.87	39.78	79.52	159.11
18	0.2	0.31	0.62	1.24	2.49	4.98	9.98	20.19	39.76	80.77	159.72
19	0.2	0.31	0.62	1.24	2.5	4.97	9.98	19.93	40.28	79.89	159.2
20	0.2	0.31	0.62	1.25	2.5	5.74	9.94	19.98	39.97	79.89	160.16
Ave	0.2	0.31	0.622	1.2465	2.495	5.188	10.033	20.0465	40.009	79.9245	159.669

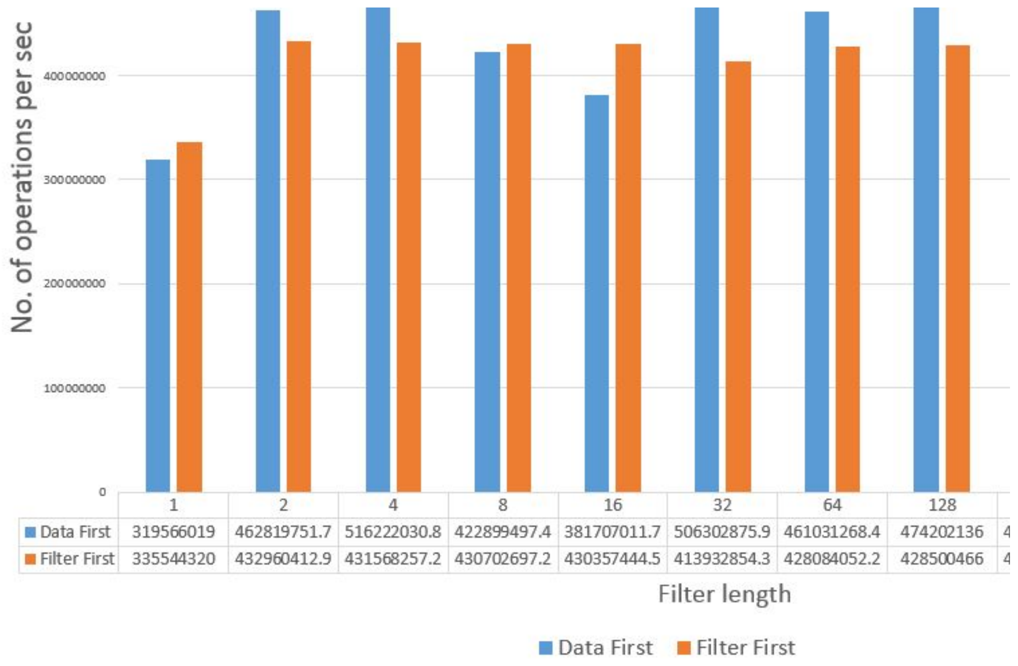
a.



	1	2	4	8	16	32	64	128	256	512	1024
■ Data First	0.21	0.29	0.52	1.2695	2.813	4.2415	9.316	18.1145	35.39	70.223	139.647
■ Filter First	0.2	0.31	0.622	1.2465	2.495	5.188	10.033	20.0465	40.009	79.9245	159.669

b. Normalise the time to filter length using the equation:

$$No.of\ operations\ \ = \ DATA\ LENGTH * FILTER\ LENGTH/run\ time$$



	1	2	4	8	16	32	64	128	
🟦 Data First	319566019	462819751.7	516222030.8	422899497.4	381707011.7	506302875.9	461031268.4	474202136	4
🟧 Filter First	335544320	432960412.9	431568257.2	430702697.2	430357444.5	413932854.3	428084052.2	428500466	4

c.

It is generally more efficient to have data in the outer loop. It can be observed that when data is in the outer loop, less time is taken to complete the task and number of operations per seconds is higher.
The reason is that, there is a memory hierarchy (L1, L2,L3, SSD, memory, disk...) from faster access but less capacity (L1) to slow access but large capacity (disk). When we want to access some data, we will start by looking at L1 cache, if the data is not there, we will go down the hierarchy. Each time the data was not there, there is a cache miss, and each time the data is there, there is a cache hit. Apparently, more cache hit will result in faster operations.

```
Code 1:
for(int i = 0; i < 1000; i++){
    for(int j = 0; j < 1000000; j++)
         ...
}

Code 2:
for(int i = 0; i < 1000000; i++){
    for(int j = 0; j < 1000; j++)
         ...
}
```

Compare two nested loops above, the inner for loop will be accessed more frequently and inner loop with large number will have less cache miss because the cache line will load a bunch of adjacent data that might be accessed in the future. On the contrary, a smaller range in the inner loop will result in more frequent cache miss and this increases the time taken to complete the task.

d.

The total number of operations is the product of DATA_LEN and FILTER_LEN. When the size of filter doubles every round which means that the number of operations also doubles. From the graph in part a, the time taken doubles as well. Therefore, the relative performance stays the same as filter size increase. This can be proven by the “almost constant” numbers of operations as well.

Part 2: Loop Parallelism

1.

a.

```
#pragma omp parallel for
for (int y=0; y<filter_len; y++) {
  for (int x=0; x<data_len; x++) {
    if (input_array[x] == filter_list[y]) {
      output_array[x] = input_array[x];
    }
  }
}
```

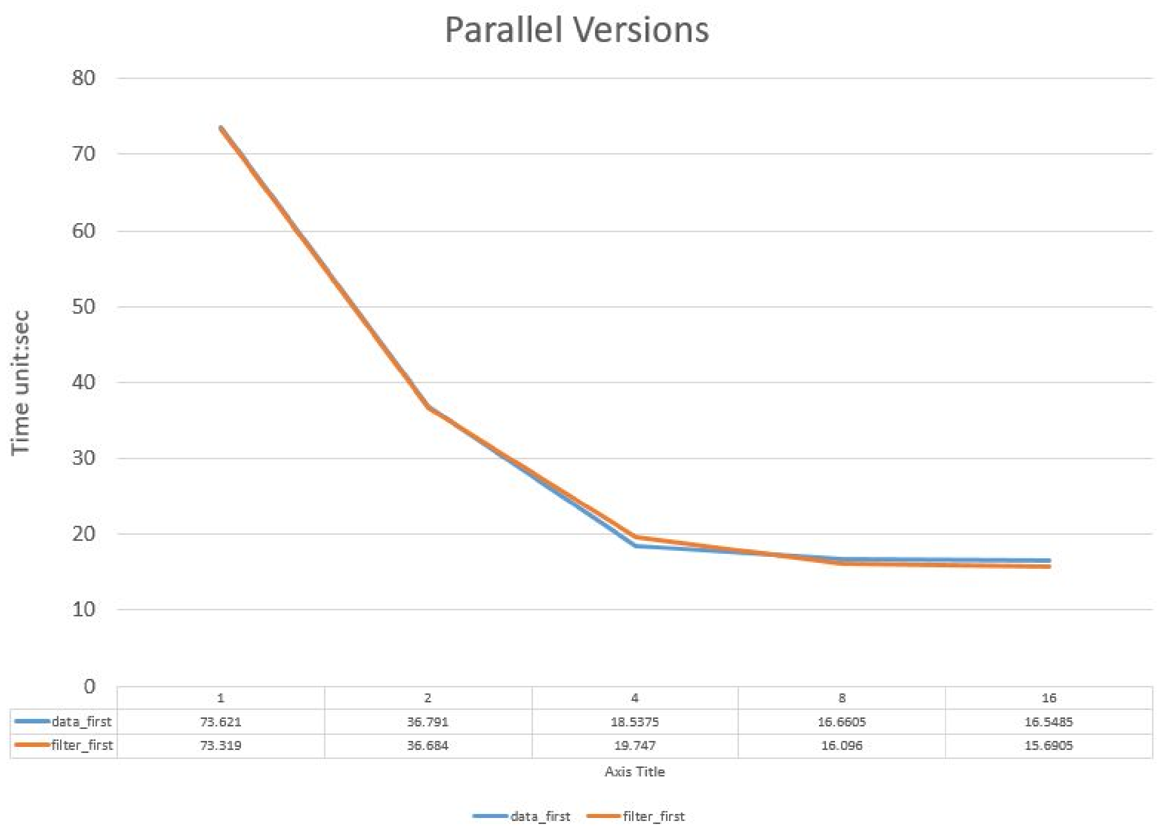
b.

```
#pragma omp parallel for
for (int x=0; x<data_len; x++) {
  for (int y=0; y<filter_len; y++) {
    if (input_array[x] == filter_list[y]) {
      output_array[x] = input_array[x];
    }
  }
}
```

2.

a. Code in filter.c

```
#define NUM_THREADS 16// 1 2 4 8 16
omp_set_num_threads(NUM_THREADS);
int num_threads = omp_get_num_threads();
int tid = omp_get_thread_num();
int step = filter_len/num_threads;
#pragma omp parallel for
for (int x=tid*step; x<tid*step+step; x++) {
  for (int y=0; y<filter_len; y++) {
    if (input_array[x] == filter_list[y]) {
      output_array[x] = input_array[x];
    }
  }
}
```



	1	2	4	8	16
data_first	73.621	36.791	18.5375	16.6605	16.5485
filter_first	73.319	36.684	19.747	16.096	15.6905

b.
The speedup is relatively linear with up to 4 thread (2-4 threads can be considered as sublinear). When the number of thread deployed is higher, there is no observable improvement. The speedup is because that more than one threads are running parallely and they are working on an independent proportion of the total work. However, each thread has to be run on one core with some setup cost. When the number of thread is more than the number of cores, some thread will be executed sequentially even through they are meant to run parallely.

Using command: lscpu

```
Core(s) per socket:    4
Socket(s):             1
```

We can see that there are 4 cores per socket and 1 socket on AWS c5.2xlarge. That means there are only 4 cores that we can use and therefore speed up curve should be tailed off when thread number equals to 4. This is same as my deduction using experiment result above.

3.
a.
From the graph in 2a, the speed for the two parallel version is almost the same and Filter_first version is slightly faster.
For parallel versions, the outer loop is divided into several blocks and each thread is only handling a small portion of data. This results in similar cache hit rate and therefore the speed is very similar.

b.
Using Filter first version:
Using the formula, $S = 1/(1-P+P/s)$
1 - 2 thread: 1.998 = 1/(1-P+P/2)   P=1
2 - 4 threads: 1.8575 = 1/(1-P+P/4)   P=0.62
4 - 8 threads: since there are only 4 cores, S should be 4. 1.227 = 1/(1-P+P/4)   P=0.25

c.
The speedup from 1 to 4 threads is almost optimal (close to 2 when the threads number doubles). When we are using more than 4 threads, the speed up is non-ideal. This is mainly associated with startup cost. The system we are using only have 4 core so that only 4 threads can be running parallely. If there are more than 4 threads, the rest will be assigned to one of the core and wait for the previous thread to finish. This increase the total startup cost.
Since each thread is handling different portion of data and there is no overlap, there will not be any interference cost. Furthermore, we need all threads to finish before we can run the check_data step and there will be some skew cost since not all threads will finish at the same time. However, since we are dividing work evenly and there is only one parallel part, the skew cost is very insignificant.

Part 3: An Optimized Version

a. Loop unrolling

```
//unrolling
  omp_set_num_threads(NUM_THREADS);
  int num_threads = omp_get_num_threads();
  int tid = omp_get_thread_num();
  int step = DATA_LEN/num_threads;
  #pragma omp parallel for
    for (int x=tid*step; x<tid*step+step; x+=8) {
    for (int y=0; y<filter_len; y+=8) {
      if (input_array[x] == filter_list[y]) {
        output_array[x] = input_array[x];
      }
      if (input_array[x+1] == filter_list[y+1]) {
        output_array[x+1] = input_array[x+1];
      }
      if (input_array[x+2] == filter_list[y+2]) {
        output_array[x+2] = input_array[x+2];
      }
      if (input_array[x+3] == filter_list[y+3]) {
        output_array[x+3] = input_array[x+3];
      }
      if (input_array[x+4] == filter_list[y+4]) {
        output_array[x+4] = input_array[x+4];
      }
      if (input_array[x+5] == filter_list[y+5]) {
        output_array[x+5] = input_array[x+5];
      }
      if (input_array[x+6] == filter_list[y+6]) {
        output_array[x+6] = input_array[x+6];
      }
      if (input_array[x+7] == filter_list[y+7]) {
        output_array[x+7] = input_array[x+7];
      }
    }
  }
```

Number of operation per loop	1	2	4	8
Ave time (s)	15.56	6.9	3.12	1.54

Loop unrolling works greatly in reducing the time taken. The trend is that the time taken halves when the if statement in the for loop doubles because we are reducing the number of loop and increasing the number of operations in one loop. Theoretically , we can remove the loop and write out all operations directly and this will produce the fastest results. But the readability will be low.

One important thing is that in order for the for loops to finish the entire data, the amount of increment has to be the factor of data_length and filter_length. Otherwise, the check data will fail.

b. Custom Scheduling

Dynamic: time taken (unit: s)

```
#pragma omp parallel for schedule(dynamic,BLOCK_SIZE)
```

BLOCK_SIZE	No BLOCK_SIZE	1	10	100	1000	10000
Filter first	15	15	16	21	73	73
Data filter	17	17	16	17	17	16

For dynamic scheduling, data are allocated to threads at run time according to the chunk size. They will get another chunk of data once they finished their current data. The default chunk size is 1 so that NO BLOCK_SIZE column and BLOCK_SIZE=1 column has same results. It can be observed that dynamic scheduling generally does not improve the performance and make things worse when the outer loop is filter. This is because when a thread get its data, the inner loop is huge so that there will be more cache miss rate. In addition, there are startup cost associate with each thread.

Static: time taken (unit: s)

```
#pragma omp parallel for schedule(static,BLOCK_SIZE)
```

BLOCK_SIZE	No BLOCK_SIZE	1	10	100	1000	10000
Filter first		15	16	19	73	73
Data filter		17	17	17	17	17

For static scheduling, data are divided into equal portion according to thread number. It can be observed that static scheduling generally does not improve the performance and make things worse when the outer loop is filter.

Since all iterations need almost same amount of time in our code, the default scheduling can be optimal and therefore custom scheduling does not improve the overall performance.