CSC367 Assignment 2 Report

Farzaneh Tabandeh
Sitao Wang

Part 1 Implementation

1.1 Sequential algorithm

We implement this filter algorithm in two phases. The first phase is to apply the filter for each pixel and gather the minimum and maximum calculated value. The second phase is to normalize each pixel based on the minimum and maximum value.

To achieve the minimum cache miss rate and maximum data locality, we carefully choose the order to access the pixel. Because the array is stored in row major, thus we access the pixel in row order, i.e. iterate each row and for each row iterate over each pixel.

1.2 Sharded algorithm

We implement all three parallel sharded algorithms in four phases. The first phase is to have each thread get assigned a portion of the shards of the image based on their id. Then each thread applies the filter for the pixels of its own portion and gathers pixels' minimum and maximum calculated value. The second phase is for each thread to wait for all threads to finish processing their assigned shards by making use of a shared barrier. The third phase is to calculate the global minimum and maximum value of pixels calculated by all threads. The fourth phase is for each thread to normalize the pixels of their portion based on the global minimum and maximum value.

1.3 Work Pool

The work pool consists of the worker threads and the work queue. Each worker thread runs an infinite while loop to fetch tasks from the work queue until the work queue is empty.
The work queue is not implemented as a queue, as it could take up a lot of space. Instead, each task is represented by its task id. All the computations are divided into two kinds of tasks: filter computation and normalization. Each task works on one chunk of data, decided by the chunk id. The task id equals the chunk id for filter computation and chunk id + number of chunks for normalization tasks. When one thread fetches a task, a task id is returned and the task id is incremented for the next task. Because multiple threads could fetch the tasks at the same time, we use mutex to synchronize the fetching.
One challenge for the work pool is synchronization between phases: no normalization task should start before all filter computations are finished. Our task id representation only guarantees that normalization tasks start after all filter computations have started,

thus we need extra methods to fulfill the requirement. We add a counter for the number of tasks completed, which is also synchronized by the same mutex for fetching the task, and a conditional variable that is signaled when the counter reaches the number of chunks, i.e. all normalizations have completed. When a thread fetches a normalization task but the completion counter is smaller than the number of chunks, then it will wait on the conditional variable. We could guarantee the order of execution through this conditional variable.

Part 2 Experiments

2.1 Experiment methods

2.1.1 Fixed image and filter while varying number of cores/threads

In this experiment we consider the image $very\_big\_sample$ (with size 1024×1024) and the filter 3×3 as constants and we analyze the performance by varying the number of cores/threads. We start by running the experiment by one single thread/core and we double it for each next run. This experiment is done over 10 executions for all 5 methods and we draw its results to measure two metrics, runtime and number of L1 cache misses in log scale.

2.1.2 Fixed image and cores/threads number while varying filters

In this experiment we fix the image $very\_big\_sample$ (with size 1024×1024) and 8 as the number of core/threads to be constants and we analyze the performance by varying the filters. We run the experiment on all filters 3×3 , 5×5 , 9×9 , 1×1 one by one.
This experiment is done over 10 executions for all 5 methods and we draw its results to measure two metrics, runtime and number of L1 cache misses in log scale.

2.1.3 Fixed filter and number of cores/threads while varying images

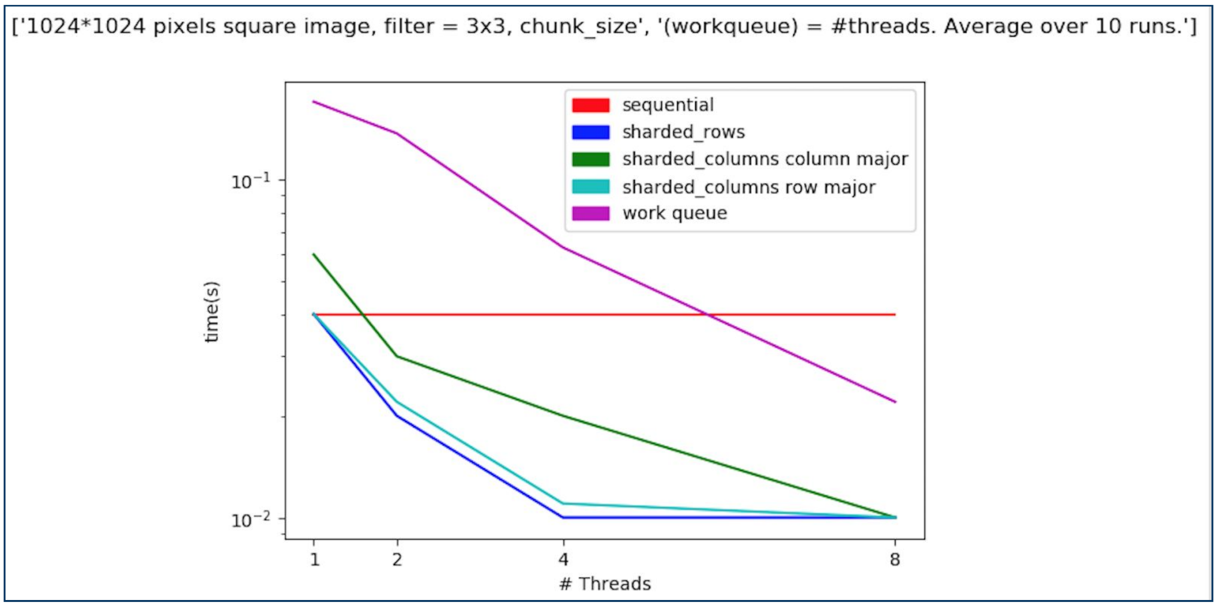
In this experiment we fix the filter 3×3 and 8 as the number of core/threads to be constants and we analyze the performance by varying the images. We run the experiment on images $very\_big\_sample$ (with size 1024×1024) , $very\_tall\_sample$ (with size 1024×1) one by one. This experiment is done over 10 executions for all 5 methods and we draw its results to measure two metrics, runtime and number of L1 cache misses in log scale.

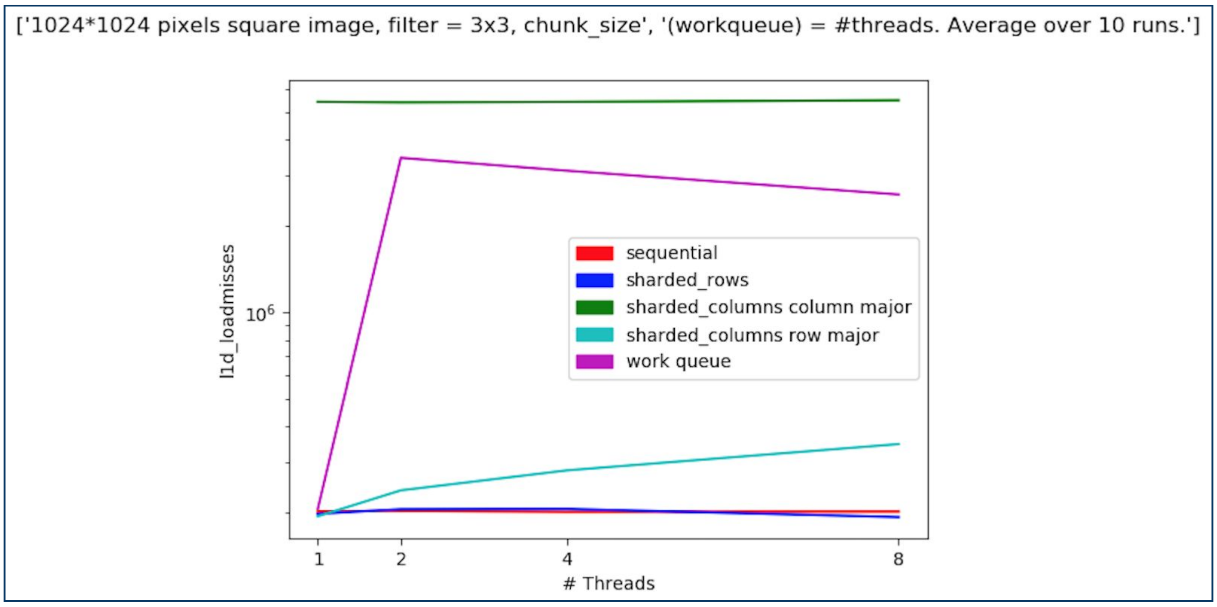
2.1.4 Fixed filter and image while varying chunk size for work pool

In this experiment we fix the filter size 3×3 and image size 1024×1024 . To have further insights into the behavior of the work pool, we change the chunk size and analyze the performance. To have a complete view of the work pool performance, we run the experiments from 1 to 8 threads.
This experiment averaged over 10 executions and we drew the results to measure two metrics, runtime and number of L1 cache misses in log scale.

2.2 Experiment results

2.2.1 A fixed image of size 1024×1024 pixels, a fixed 3×3 filter and various core numbers

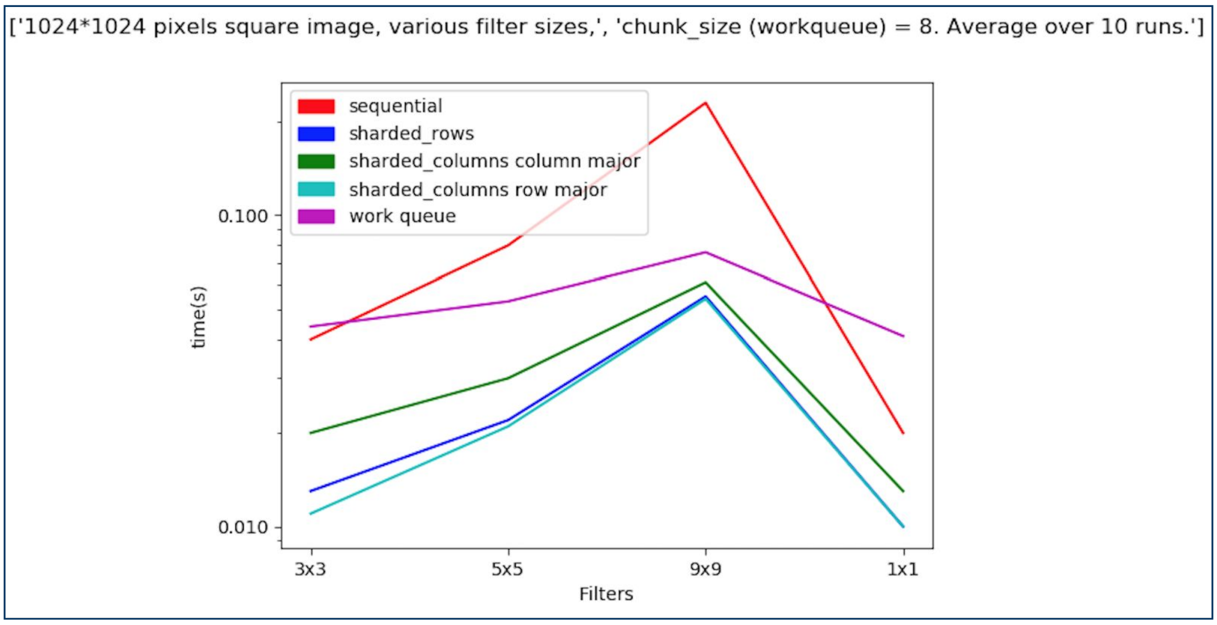


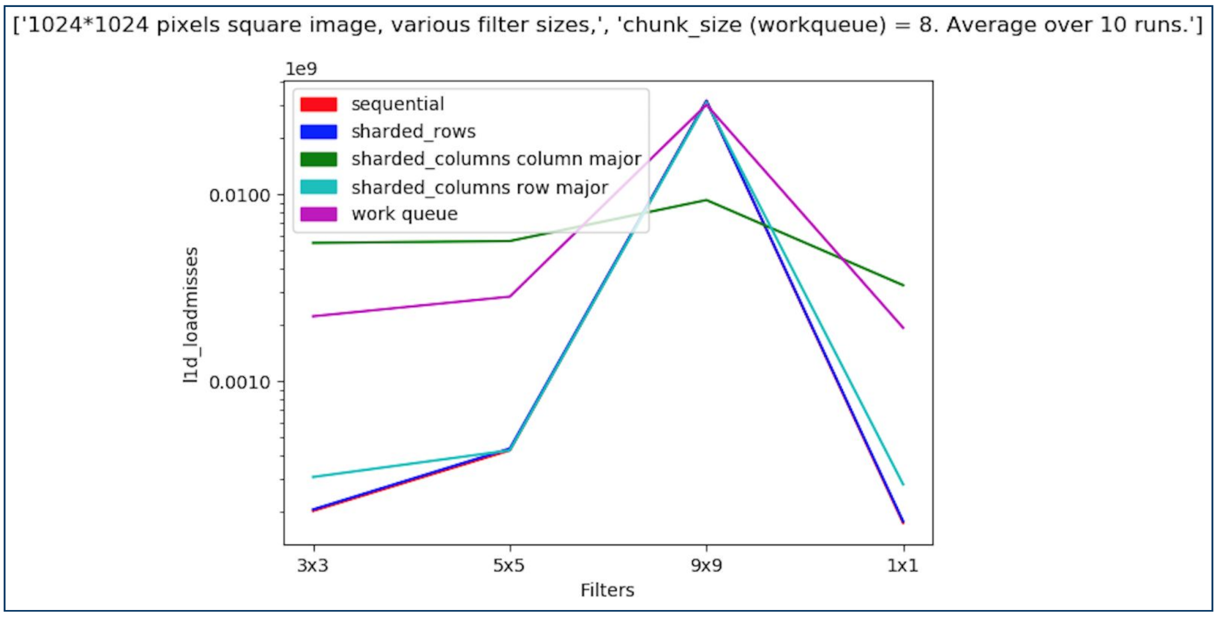


As shown in the visualization, predictably the runtime of the sequential algorithm is constant at 0.04 sec over the increase of the number of cores. As the sharded_rows method processes the image row by row and as the result is able to make use of the spatial cache locality very well(similar to the sequential), it has the lowest runtime among all. The increase of threads/cores shows a drastic decrease in runtime for this method as more chunks of image are parallzied. One point about this parallel method is that it's runtime remains

constant when the thread number goes up from 4 to 8 and this shows the ultimate minimum runtime possible for processing this image is at 0.01 sec.
The sharded_columns row major method shows the same decreasing trend as the sharded_rows', however its runtime overall is greater than that method as expected. Although each thread in the sharded_columns row major method processes its portion of work row by row which is beneficial in terms of spatial cache locality, sharding the image based on column causes the threads to not to be able to make full use of their fetched cache line and this in turn causes more cache misses compared to sharded_rows method. Also this way of sharding causes the overhead of false sharing since threads would fetch the same cache line.
The sharded_columns column major method with one single thread has obviously a bigger runtime than the sequential as shown. This is clearly due to a high level of cache misses as shown in the visualization below. With increase in threads its runtime decreases to a good degree, although its time complexity overall is still much greater than the other two parallel methods talked about above. The sharded_columns column major method has the highest level cache miss amog all methods.
One observation is that all of these 3 parallel methods have the same runtime when the number of threads is 8 which shows we could get the best runtime with 8 cores regardless of the method(among these 3) used.
The work queue method has a drastic greater runtime overall regardless of the number of threads used for processing the image. The reason for that is certainly the overhead of synchronization between threads while communicating with the work queue. We can observe this overhead more clearly when the thread number is just one. Then overhead of using synchronization primitives has caused the time complexity to be much higher than the equivalent case in the sequential method (0.16 sec compared to 0.04 sec).

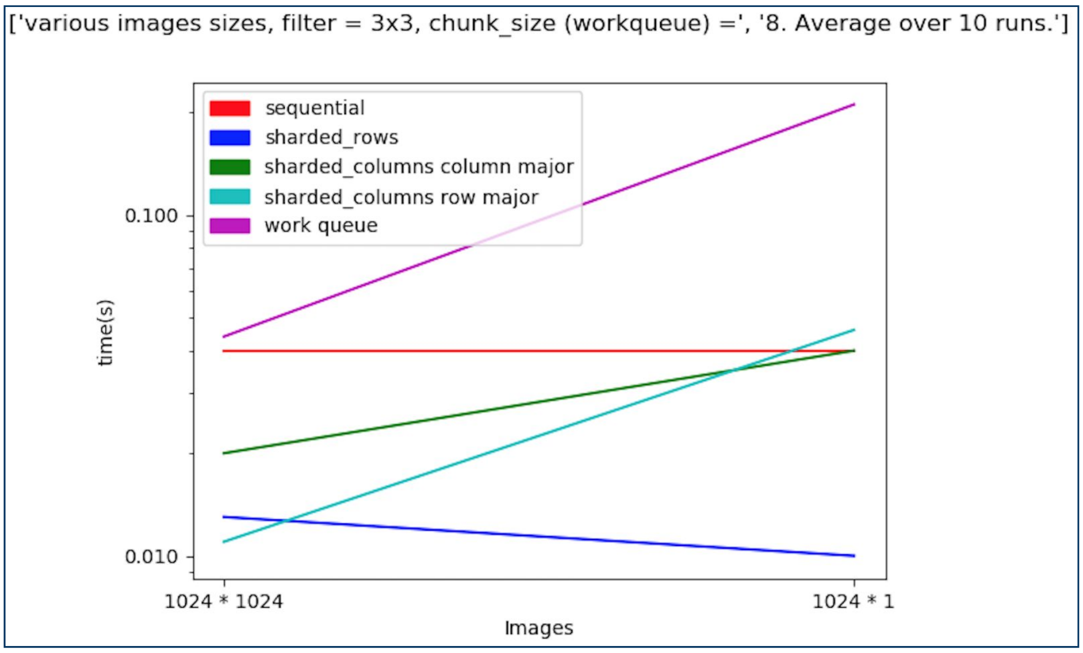
2.2.2 Fixed image of size 1024×1024 pixels, fixed 8 cores and various filters

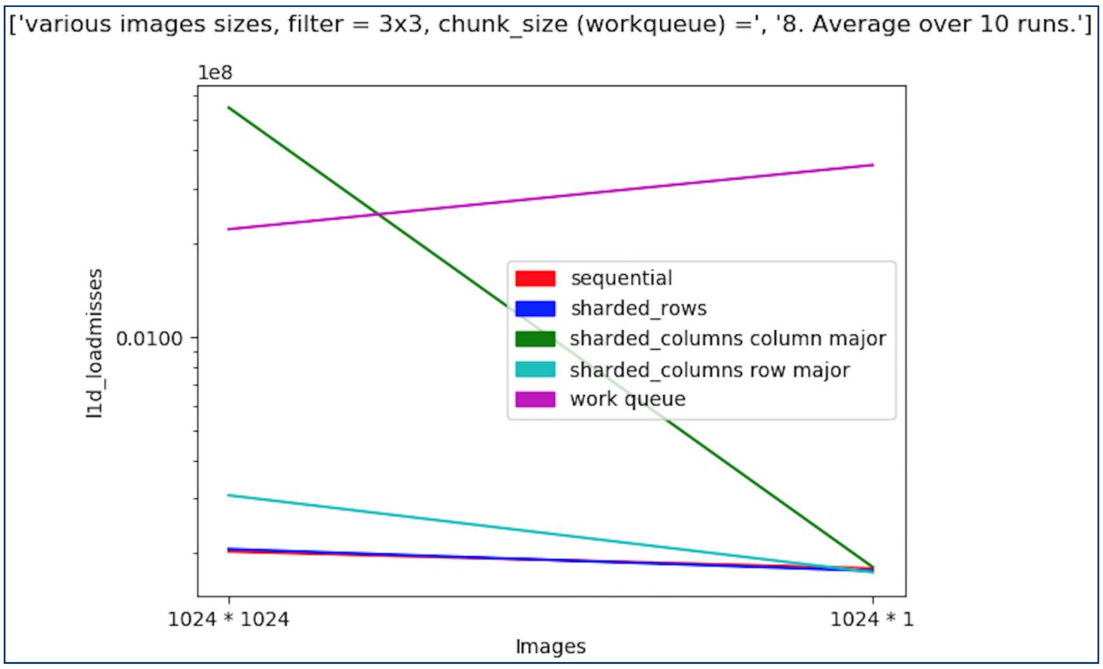




As shown in the above graphs, size of the filter applied on the image has an important role in the runtime of every method. Overall the greater the size of the filter, the greater the runtime is. This is because processing every pixel needs more access to neighbouring pixels and probably more cache line fetching. Specifically, the results shows that the filter size of 9×9 causes the runtime of all methods to peek at the highest level.
Sequential and sharded_rows show exactly the same runtime behavior over the change of filters.
The sharded_rows and sharded_columns row major methods have similar runtime when the filter size is greater than 3×3. The reason for that, as shown in the L1 cache visuation, is that the number of cache misses of these two methods are equally high as both have to fetch new cache lines for processing a bigger area of the image. Bigger filters certainly also cause overhead of false sharing as threads have to have access to a greater number of neighbours of a pixel each time.
Interestingly, sharded_columns columns major method has lower cache misses for the 9×9 filter compared to all other methods. As the memory is row major, every pixels' row neighbours will be fetched in a cache line automatically. In the case of this method specifically, the thread fetches the columns around every pixel one time, but then while it goes down a column to process every pixel, it has the row and column surrounding that pixel already fetched and it actually gets to access them all. Infact in this method, since each specific thread works on a column, the fetched rows from previous processing are used again while the thread processes the pixels downward.
Work_queue method shows a lower cache misses than sharded_columns columns major in the smaller filter sizes.
This experiment also shows that the overhead of synchronization is much bigger than the cache miss overhead, as Work_queue does poorly in terms of runtime among all methods.

2.2.3 Fixed 3×3 filter, fixed 8 cores, and various image sizes

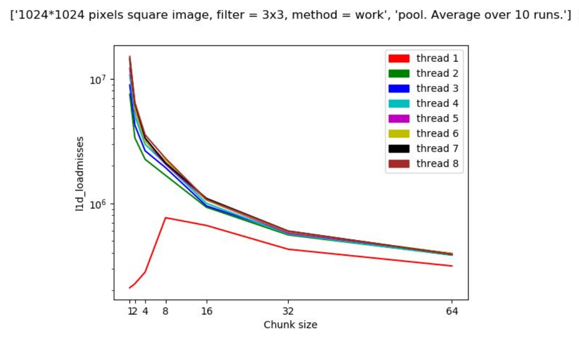


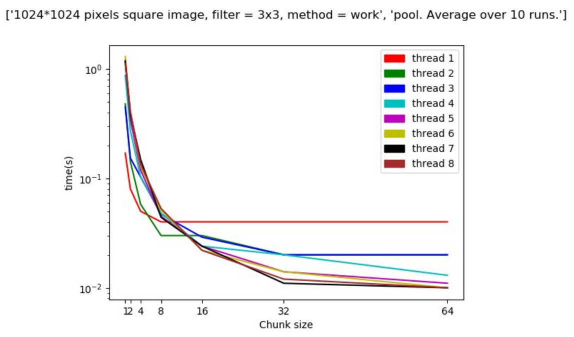


This experiment enables us to compare the tall vs square image results over the methods. The data gathered shows that having taller images (i.e: an image with bigger height compared with its width) removes the effect of parallelization for sharded_columns row major and sharded_columns column_major as the whole image will be processed by one thread. All methods except Work_queue have the same cache miss when the image is tall and the reason is all of these methods behaviour in this case become equivalent to the sequential method. When the image is tall, the only parallel methods are the sharded_rows which

assigns every row(i.e: every pixel) to one thread and the Work_queue. The Work_queue can also distribute the works on the image quite evenly between the threads as its division is based on both height and width. Although these two methods will partition the image among all threads the same, the overhead of synchronization for the queue causes the Work_queue to have a much bigger runtime.
So in terms of runtime, sharded_rows has even best time complexity among all and its runtime is even better when the image is tall, as it makes use of all 8 threads for a single pixel each time which causes the runtime for each thread to be very short. This is the only parallelization method that makes sense to choose over the sequential if the image is tall. This experiment clearly shows the dimensions of the image that needs to be processed will have a great impact on what parallelization method should be used (if any) to get a desired speed up.

2.2.4 Fixed image of size 1024×1024 pixels, a fixed 3×3 filter and various chunk sizes and cores





From the results, we could see that, generally, the cache miss decreases and runtime decreases with the increase of the chunk size. The increased chunk size provides better data locality and thus lower cache miss rate.
There is one exception for this observation, which is the cache miss rate for the single thread. Because for the single thread with chunk size 1, the algorithm accesses the array in the same order as the serial algorithm and has extremely high data locality as it visits the array in the same order as data is stored. When the chunk size is 2, however, each chunk contains data from different rows, which is the source of cache misses.
We could also observe some patterns for thread number. Generally, with increased thread number, the cache miss rate increases. This is because the parallel data access has low data locality, and thus the increase in parallelism causes higher cache misses rate. We could see from the runtime that, for small numbers of chunk sizes, the smaller number of threads has smaller runtime because of the lower cache miss rate. However, as the chunk size goes up, the difference in cache miss rate diminishes because of the poor data locality for each chunk, the efficiency of parallelism dominates and the runtime is smaller for a higher number of threads.