CSC367 Winter 2021
Assignment 2 Report

Shuting Zhang zhan4145
Muyuan Cao caomuyua

1 Implementation

1.1 The main helper functions

Apply2d() is a straightforward but very important function since it will be called repeatedly in other functions. It processes a single pixel and returns the value of the processed pixel.

normalize_pixel() normalizes a pixel given the smallest and largest pixel in the image.

1.2 Sequential Method

The sequential method calls apply_filter2d(), which is a function that iterates through each pixel in the image, row by row, applies the filter on each pixel by calling apply2d(), and updates the global max pixel and the global min pixel accordingly. Then it calls normalize_pixel() to bring all pixels to the [0, 255] range.

1.3 Horizontal Sharding

The horizontal sharding method calls apply_filter2d_threaded() first to create threads and assign the corresponding work implemented in horizontal_sharding() to each thread. Then, in horizontal_sharding(), each thread processes a set of consecutive rows. Each thread gets assigned a roughly equal number of rows, with the exception of the last thread, which possibly gets more rows due to the fact that the total number of rows might not be divisible by the number of threads. After each thread finishes processing the rows assigned, by applying apply2d() on each pixel, and collecting the partial max and min pixel, it tries to get a mutex to update the global max and/or min. Then it waits for all other threads to finish their jobs by calling pthread_barrier_wait(). After all the threads finish processing their assigned rows, the global max and min pixels are available, the threads proceed to perform normalization on their assigned rows.

1.4 Vertical Sharding: Column Major

Similar to horizontal sharding, the vertical sharding column major method calls apply_filter2d_threaded() first to create threads and assign the corresponding work implemented in vertical_sharding_column_major() to each thread. Unlike the horizontal sharding method in which each thread gets assigned a set of consecutive rows, each thread processes a roughly equal number of consecutive columns in a manner such that it processes all the pixels in a column before moving on to the next column. Then the function updates the global max and min pixel and does normalization.

1.5 Vertical Sharding: Row Major

This method and its implementation, vertical_sharding_row_major(), is very similar to the previous one---each thread processes a roughly equal number of consecutive columns, but in a manner such that it processes first all the pixels from the first row of every column in its subset of columns, then moves on to the pixels from the second row of its columns, and so on.

1.6 Work Pool

In the work pool method, the image is divided in square tiles of size chunk x chunk. The tiles are placed in a queue. Each thread takes one tile of pixels from the queue, processes it, and proceeds to grab another one from the queue. In order to implement this abstraction, this method first invokes create_work_queue(), which enqueues a linked list in which each node stores location information for a single tile: the start and end rows, and the start and end columns. Then, in work_pool(), each thread locks the linked list, grabs a node, stores the location information locally, and unlocks the linked list. With the location information, the thread processes each pixel within the tile and collects partial max and min pixels. Then it proceeds to grab and work on another tile. After the threads finish processing all the tiles, the threads iterate through the same linked list again to grab tiles and normalize the grabbed tiles.

2 Experiments

2.1 Comparing the Performance of Different Methods

2.1.1 General Idea

In this experiment, we will run the sequential algorithm and the parallel methods for the hard coded image. For each run, we varied the number of threads (1, 2, 4, 8) which served as the independent variable. And we will compare the L1 cache misses and the running time of processing the image among the different methods.

2.1.2 Experiment Setup

In the command line, call make run will invoke perfs_student.py, which will generate every graph for this report. Specifically, the function graph() collects the data, both perf and timing, needed for this experiment and generates 2 graphs accordingly. Moreover, this experiment is repeated 10 times for more stable and accurate results.

2.1.3 L1 Cache Misses of Different Methods

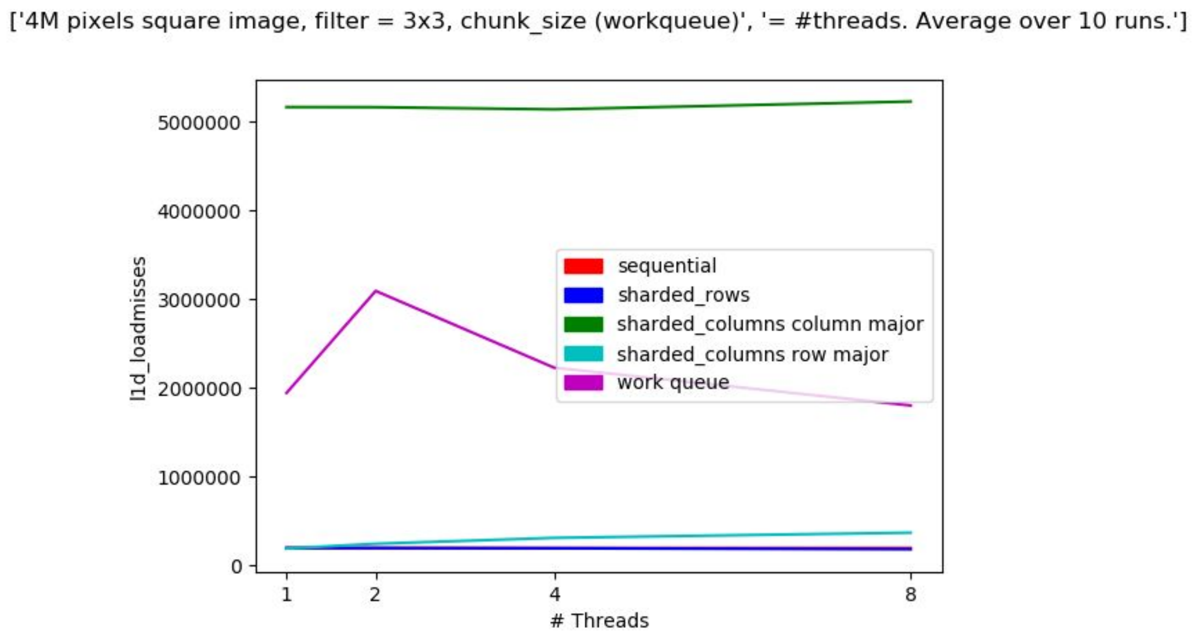

Figure 1: Comparing the Performance of Different Methods---L1 Cache Misses

The above figure 1 demonstrates each method's L1 cache misses.

The line representing the vertical sharding column major method is the one on the top of the graph with the most cache misses. This agrees with what we have learned in class, that matrices are stored row by row in cache. Supposed the image we have processed is of size N * N, then if we traverse the matrix column by column, the pixel we access each time is N pixels apart, which is a very poor special locality. As a result, the cache misses for the vertical sharding column major method is very high.

The three lines on the button each represents the sequential method, the horizontal sharding method, and the vertical sharding row major method. They function in a similar manner such that they all traverse the pixels row by row. As discussed above, matrices are stored row by row in cache. The pixel we are going to process next is likely to already be in cache. Therefore, these three methods have low cache misses.

Regarding the line that represents the work queue method, we made three observations:
O_1: when the number of threads / chuck size is 1, it has relatively low cache misses, but still higher than the three methods on the bottom of the graph

O_2: it has relatively high cache misses when the number of threads / chuck size is 2
O_3: The cache misses gradually becomes lower as the number of threads / chuck size become larger (i.e. 4 and 8)
For O_1: although the chuck size is 1 and the work queue method works basically the same as the other three row-major methods---processes each pixel row by row, it has extra cache misses as it also needs to access the work queue.
For O_2 : the size of each work chunk is 2 * 2. While the 2 pixels on each row have good special locality, the special locality is bad when the method finishes processing the 2 pixels on the first row and proceeds to the second.
For O_3: although the special locality is still poor when the work queue method finishes processing one row and moves on to the next, the rows are longer (i.e. 4 and 8), which means the method does not need to switch rows as often.

2.1.4 Timing of Different Methods

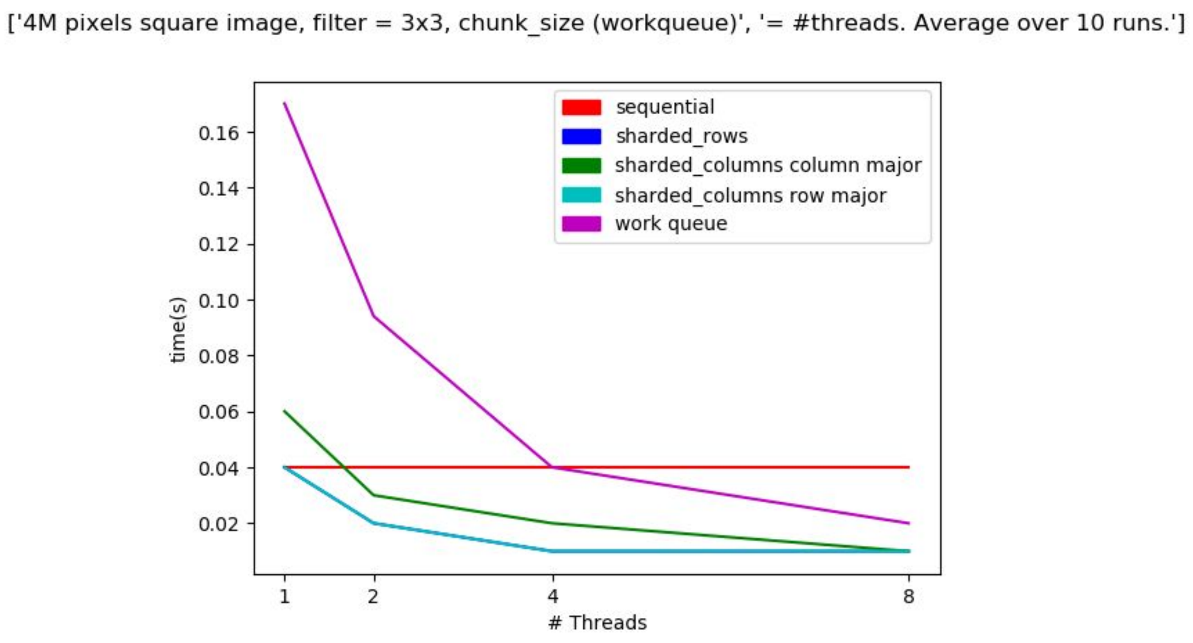

Figure 2: Comparing the Performance of Different Methods---Timing

The line for the sequential method is a straight line as it does not have parallel implementation. As the number of threads increases, it becomes the slowest one as expected.

The other methods all become faster as more threads are available; the time they use, as the number of threads doubles, becomes approximately half as before.

The cache load misses, on some extinct, also reflects the performance of a method. A method is more likely to have poor performance if it has higher cache misses since the processor would need to spend more time accessing data in slower memory. This agrees with what we observe from Figure 2: the work queue method and the vertical sharding column major

method take longer time than the horizontal sharding method and the vertical sharding row major method.
The reason that the work queue method has even poorer performance than the vertical sharding column major method might be because it has additional overhead introduced by extra synchronization regarding accessing the shared work queue.

2.2 Work Pool Method with Different Chunk Sizes

2.2.1 General Idea

In this experiment we test the work pool method for a variety of chunk sizes of 1x1, 2x2, 4x4, 8x8, 16x16, and 32x32, in order to explore the role chunk size plays in image processing.

2.2.2 Experiment Setup

The perfs_student.py generates every graph for this report. Specifically, the function graph2() collects the data needed for this experiment and generates 2 graphs accordingly. Moreover, this experiment is repeated 10 times for more stable and accurate results.

2.2.3 L1 Cache Misses

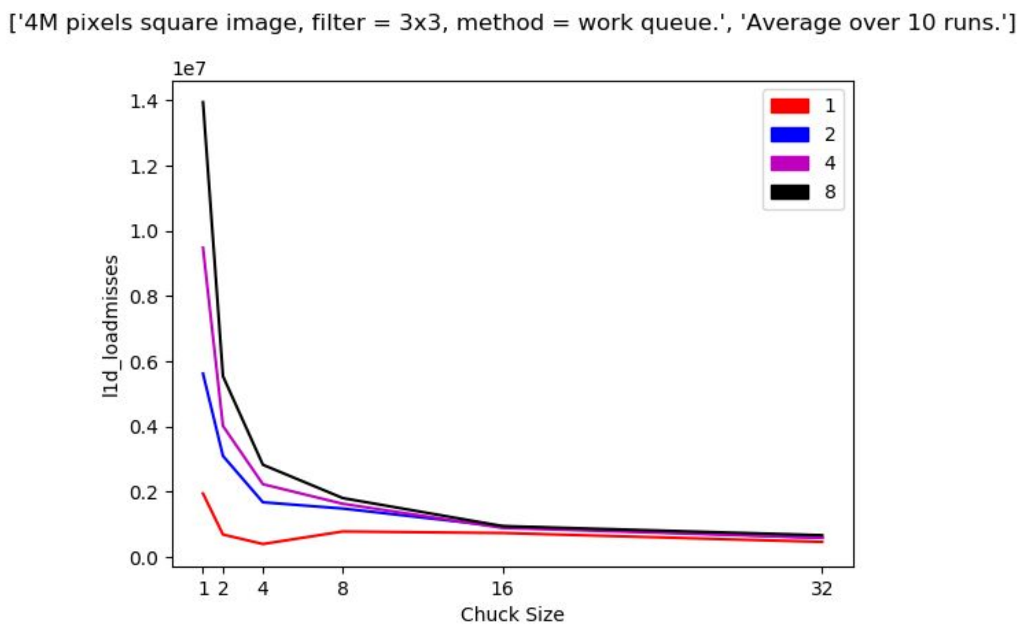

Figure 3: Work Pool Method for Different Chunk sizes --- L1 Cache Misses

As demonstrated by Figure 3, when the chuck size is fixed, the more threads we have, the more cache misses happen. This is because, as we employ more threads, the tiles of pixels each thread grabs would be further apart, causing poorer special locality.

Another observation is that as the chuck size becomes bigger, the number of cache misses decreases. This is because, as the chuck sizes become larger, the cache will be able to store more nearby pixels which could be used in the nearby future, leading to better temporal and spatial locality.

3.2.4 Timing

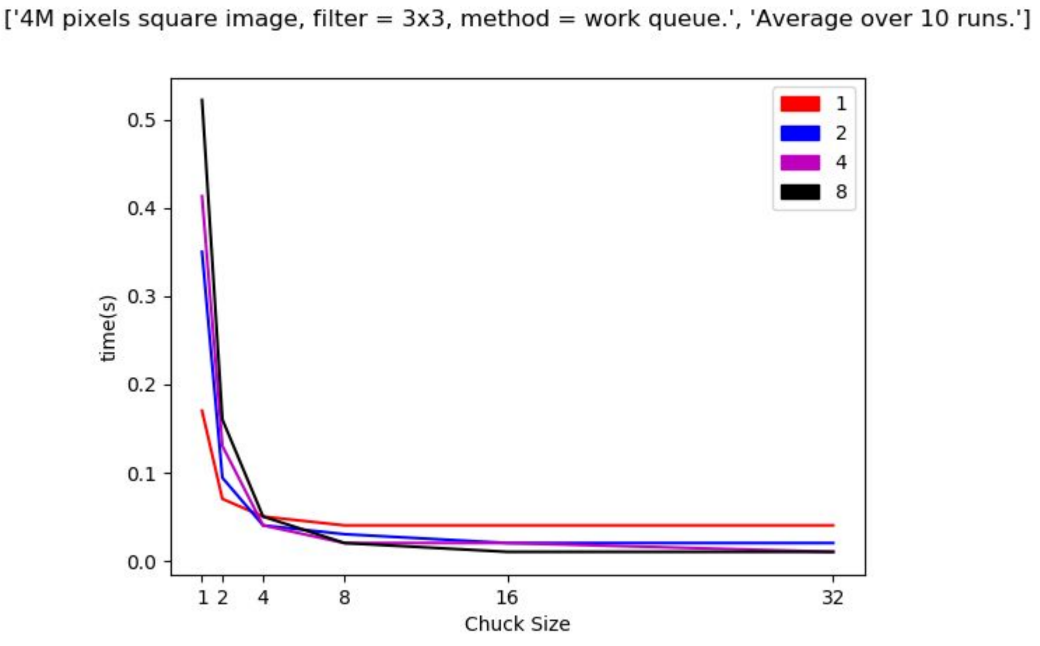

Figure 4: Work Pool Method for Different Chunk sizes --- Timing

The method performs better when the chuck size is larger since, as we have observed in Figure 3, larger chuck size leads to less cache misses.

When the chuck size is really small, one for instance, the performance of the work pool method is very poor. This is due to the overhead introduced by the synchronization for accessing the shared work queue. The more threads we employ, the more threads would need to wait for the mutex to become available, causing more overhead.

The lines eventually become smooth since cache misses will become stable when the chuck size is large enough, which agrees with the observation from Figure 3.

2.3 Impact of Different Filter Sizes

2.3.1 General Idea

In this experiment, we will vary the filter size (1x1, 3x3, 5x5, 9x9) for each of the methods and analyze the impact of different filter sizes

2.3.2 Experiment Setup

The perfs_student.py generates every graph for this report. Specifically, the function graph3() collects the data needed for this experiment and generates 2 graphs accordingly. Moreover, this experiment is repeated 10 times for more stable and accurate results.

2.3.3 L1 Cache Misses

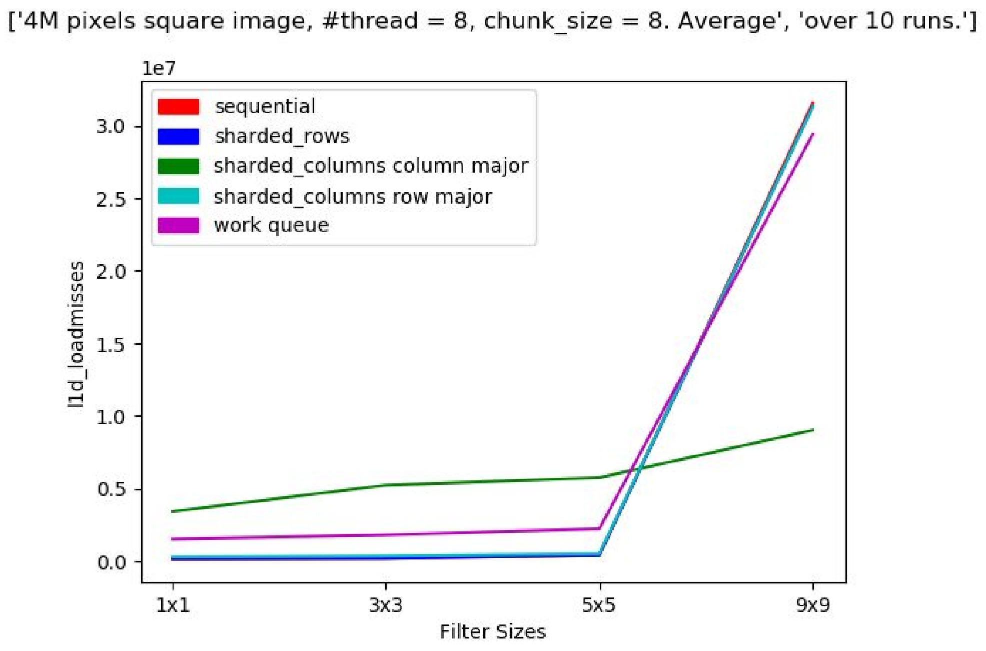

Figure 5: Impact of Different filter sizes --- L1 Cache Misses

From Figure 5, we can observe that as the filter size increases, the number of cache misses increases, as expected. When the filter size becomes larger, more rows need to be traversed in order to process a single pixel.

Another observation is that the 4 row major methods (sequential, sharded rows, sharded columns row major, work queue) have relatively smooth increasing L1 cache misses until the 5 x 5 filter, after which they all experience a drastic increase in L1 cache misses. This is probably because the methods with the 5 x 5 filter hit the limit of the cache. With filters bigger than 5 x 5 (i.e. 9 x 9), the pixels and the filter needed to process a single pixel might not be able to fit in the cache.

The third observation is that, in the column major method, filter size does not affect the cache misses too much. This should be because when we calculate the current pixel, we store the next pixels (on other rows) in cache. Then, it's kind of a cache hit when the column major method moves to the next pixel in the same column and computes by using its neighbours.

2.3.4 Timing

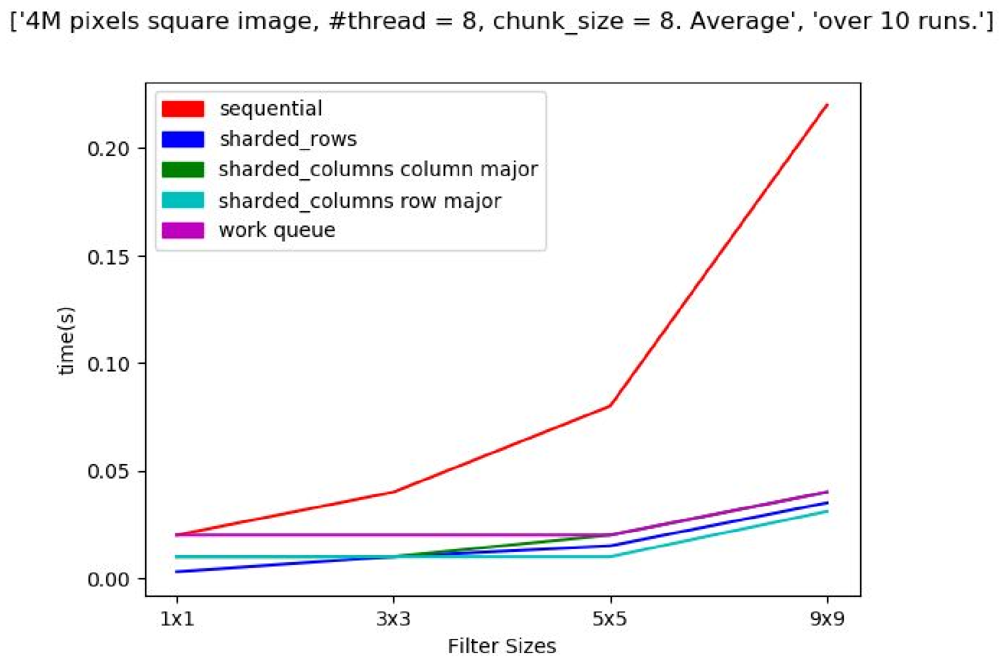

Figure 6: Impact of Different filter sizes --- Timing

From Figure 6, we can observe that as the filter size increases, the time cost increases, which is as expected. As discussed in 2.3.3, when the filter size becomes larger, more rows need to be traversed and more calculations need to be done to process a single pixel. The drastic increase in time when the filter size becomes bigger than 5 x 5 can, to some extent, be explained by the drastic increase in L1 cache misses. This is, again, because when the filter becomes larger than 5 x 5, the pixels and the filter needed to process a single pixel might not be able to fit in the cache. The sequential method becomes significantly slower than the parallel strategies when filter size becomes larger is because the parallel methods contributed the work to 8 threads, while the sequential method does not have this parallel implementation.

2.4 Impact of Different Images

2.4.1 General Idea

In this experiment, we will explore how different image widths affect the performance of various methods. We will run the sequential algorithm and the parallel methods for our own images. The total number of pixels is fixed to be 1M pixels, we only vary the width(and the corresponding height) and check the performance of different methods. The width is ranging from 1 to 32K, since from the last assignment we know that Scinet's L1 cache is of size 32KB.

Note that it's inevitable to have serious I/O operations if we want to use our own created images. But the image size is fixed in this experiment, we still can find some trends from the perf result.

2.4.2 Experimental Setup

For this experiment, graph4() in perfs_student.py will be called. We will use the new images created by pgm_creator.out, rather than the hard code images. With the fixed image size 1M, for each run we will vary the image width size (1, 8, 16, 64, 512, 1024, 4096, 32768). And we will compare the L1 cache misses and the running time of processing the image among the different methods.

2.4.3 L1 Cache Misses of Different Methods

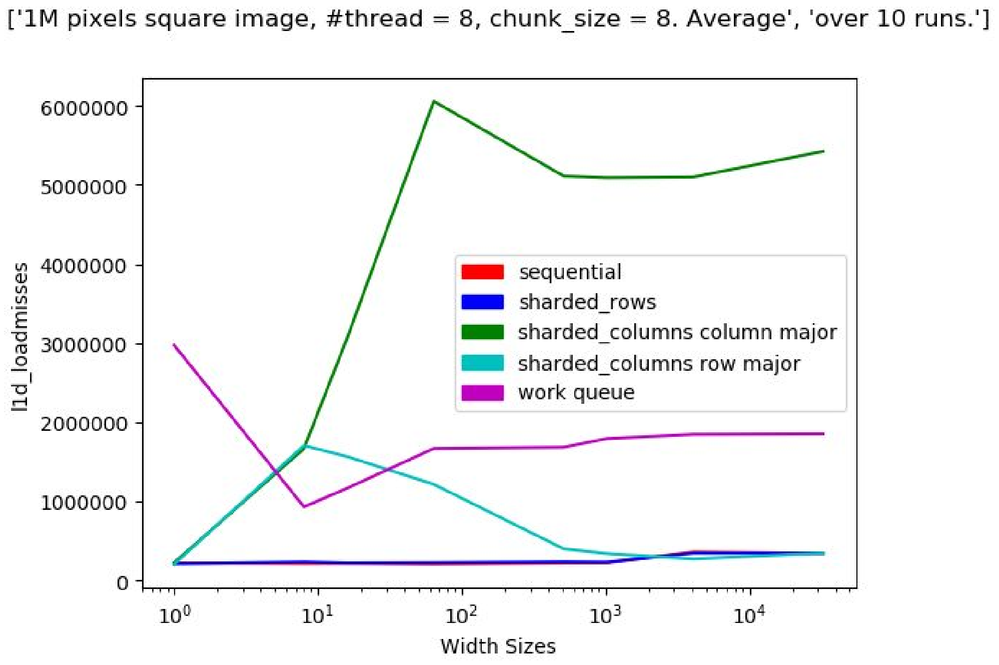

Figure 7: Comparing the Performance of Different Methods---L1 Cache Misses

Values on the x-axis are different image widths, and L1 Cache Misses are displayed as y values.
Based on Figure 7, we made some observations:
O_1: Similar as what we observed in experiment 2.1, the sequential algorithms and sharded_rows have the lowest cache misses. And sharded_columns column major performs worst (when image width is not equal to 1). When the image width is small, the column major method performs well. That's because when width sizes smaller than threads' number, the column major method accesses the matrices as similar as other methods do.

O_2: Sharded_columns row major method doesn't perform well when image width is approximately the same or smaller than the threads numbers(8). It should because there is no big difference between sharded_columns row major method and sharded_columns column major method, when the image width is approximately the same or smaller than the threads numbers.

O_3: For the work queue, it has the largest cache misses when image width is 1, has the smallest cache misses when image is 8, and stays at 2000000 for the following width sizes. For the work queue method, when image width is equal to the chunk size, it actually accesses the matrices 8 rows by 8 rows. Then the tiles of pixels each thread grabs would be less apart, spatial locality is utilized well. Also, with the fixed chunk size(8), when image width is 1, we actually only use 1 column of the square tiles. In other words, the size of tiles(be really used) becomes smaller. When we have the same amount of threads and the same image size, the smaller tiles/chunks, will lead to larger cache misses, as we discussed in the second experiment.

2.4.4 Running Time of Different Methods

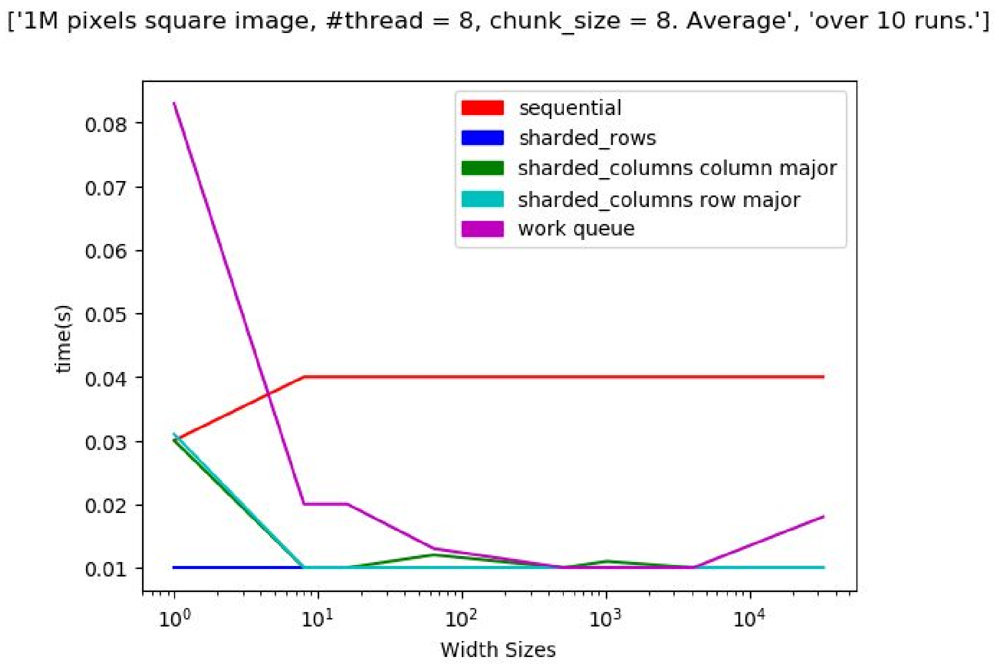

Figure 8: Comparing the Performance of Different Methods---Timing

Values on the x-axis are different image widths, and timing results are displayed as y values. Based on Figure 8, we made some observations:
O_1: Different image widths do not affect the time cost too much except when the image is only 1-pixel wide.
O_2: For the sequential algorithm, the time cost when image width is 1 is smaller than other time costs. While for parallel methods, the time cost when image width is 1 is much larger than other time cost (except sharded_rows).

We think this result will take place when the image's width is approximately the same or less than the filter width. In this case, spatial locality is utilized poorly. When we are computing the value of a pixel, the cached data can no longer be re-used by some other computations in the future.