CSC367 assignment 2

hongmao.wu(1008134207)

October 2023

A grayscale image comprises numerous pixels, each assigned a value within the range of 0 to 255. To accentuate edges in an image, a Laplacian filter is employed. However, after applying this filter, pixel values may extend beyond the expected 0-255 range. Therefore, it is crucial to normalize the resulting values to ensure they fall within the acceptable range.

Part 1 - Sequential implementation

This paragraph provides insights into the sequential implementation approach. In sequential implementation, a single thread is used to process each pixel in the image. As a result, this method doesn't require the main thread to create new threads. While this approach may be relatively slower compared to multi-threaded methods for pixel processing, it offers several advantages.

One notable advantage is the absence of common issues associated with multi-threading, such as locking contention, synchronization overhead, and cache coherence overhead. Furthermore, due to the absence of concurrent writes to the same cache lines, the occurrence of false sharing is minimal, resulting in fewer cache misses, as illustrated in Figure 1. In addition, the sequential method completely avoids any race condition.

Part 2 - Data Parallel implementation

This implementation divides an image into rows or columns and employs various methods to distribute a roughly equal number of rows or columns to each thread. For later explanations, let's assume the picture has dimensions of N rows and M columns, with a total of X threads available for processing.

Process used by all methods

This paragraph outlines the general steps followed by each method for computing pixel values within a designated block.

The process begins by calculating the start and end indices for the specified block. Next, the filter is applied to every pixel in the block, computing new values that are stored in an array called 'loc_target'. Subsequently, the algorithm identifies the maximum and minimum values within the 'loc_target' array.//

To enable multiple threads to write to the target array without conflicts, multiple locks are used to create critical sessions. A thread then encounters a synchronization barrier which requires every thread to pause and wait until all of them have completed the computation of maximum and minimum values for their respective image blocks.

After leaving the barrier, the program then calculates the global maximum and minimum values, representing the extreme values across the whole image. The next step normalizes every pixel in the 'loc_target' array to ensure that the pixel values fall within the valid range for a grayscale image. Finally, the processed results are written to the output image.

Three different methods to divide an image

This paragraph outlines the process of dividing a picture into rows or columns. The method divides a total of N rows or columns and assigns each thread a workload of approximately (X + N - 1) / X rows or columns. It's important to note that, in some instances, not every thread may receive the full (X + N - 1) rows or columns to process.

For instance, if X equals 7 and N is 16, each thread would be assigned (7 + 16 - 1) / 7 = 3 rows or columns. However, the second-to-last thread would have only 1 row or column to process, and the last thread would receive none. To address this, the program calculates the actual number of threads required to ensure that no thread is left with zero rows or columns to process.

Subsequently, the program creates an equal number of tasks and assigns each thread a task. Each task follows the above process to compute values for the resulting image. This approach guarantees an equitable distribution of the workload among the threads.

This paragraph introduces three methods for processing images. The "Sharded Rows" method divides the original image into rows. In contrast, the sharded columns column major and sharded columns row major methods split the original image into columns.

However, there is a difference in how these methods compute values. In the sharded columns column major approach, values are computed from the top to the bottom and from left to right. On the other hand, the sharded columns row major method computes values from left to right and then from top to bottom. This distinction in processing order can have implications for the results produced by these methods.

Analysis

This paragraph analyzes key factors that impact performance. In the sharded row method, values are computed from left to right and then from top to bottom. This order of computation is advantageous because values on the same row in an array are stored in the same cache line. As a result, the sharded row method has much more cache hits than the other two methods.

Similar reasoning can be applied to explain that the sharded columns row major method would result in more cache hits than the sharded columns column major method. This difference in cache latency between cache hits and misses can significantly impact the overall performance of these methods.

This paragraph explores another critical factor impacting performance. The results displayed in Figure 2 illustrate that, in general, a larger number of threads leads to faster execution times. However, it's essential to note that the execution time doesn't simply halve each time the number of threads doubles, and the performance gains diminish as the number of threads increases.

For instance, consider the sharded columns column major method. With one thread, it takes around 0.035 seconds. While with two threads, it takes around 0.022 seconds. One key factor is locking overhead. To prevent synchronization

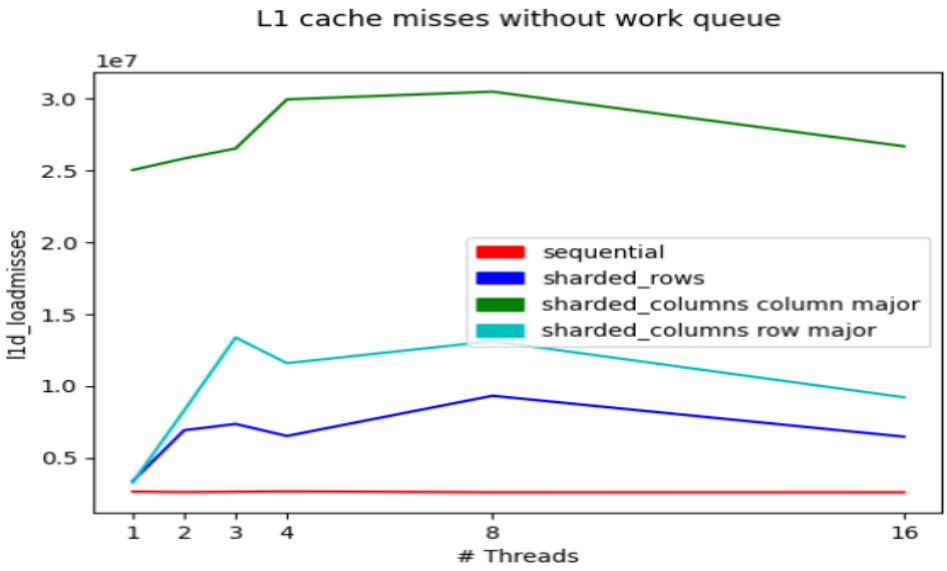

Figure 1: L1 cache misses

issues, a lock can temporarily block a critical section, causing other threads to wait until the thread completes its modifications to a shared resource. This idling time could accumulate can impact performance.

Additionally, the barrier ensures that the fastest thread must pause and wait until the slowest thread reaches the barrier. This waiting time adds to the total execution time.

This paragraph discusses other minor factors that impact performance. One of these factors is that there are parts of the program that are not parallelizable and take a constant amount of execution time. Since not the entire program can be parallelized, Amdahl's law explains that the speedup will be less than 2. Additionally, the lab machine used for testing only has 8 cores, which limits performance gain beyond 8 threads. This limitation is obvious in Figure 2; running the same task with 16 threads does not reduce execution time because CPU cores need to constantly switch contexts, which introduces overhead.

Another contributing factor is false sharing, where a part of the program allows multiple threads to access an integer array simultaneously, and if they access data at close addresses, each thread invalidates the other's cache line, resulting in increased memory latency. These various factors collectively worsen the overall performance of the program.

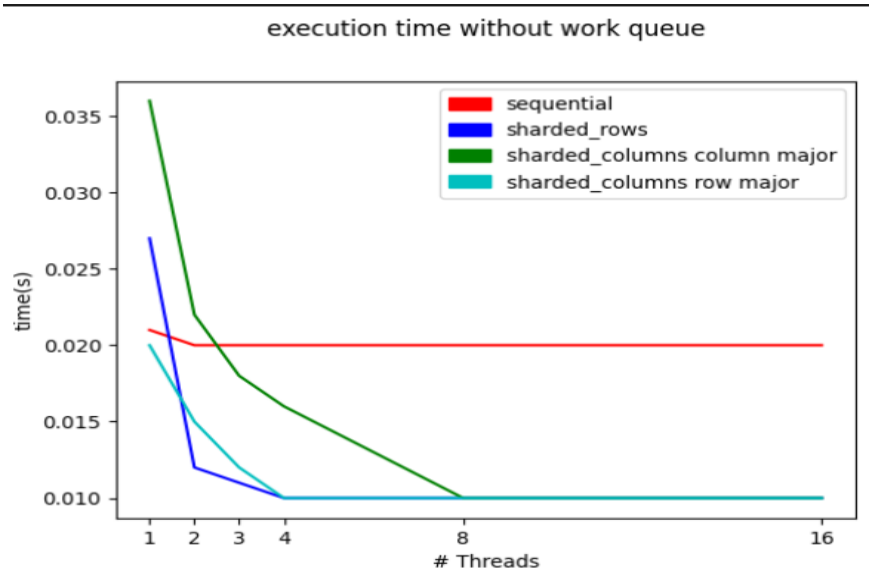

Figure 2: execution time

Part 3 - Work Pool implementation

This paragraph explains the work pool model and its implementation using a work queue. In the work pool model, a pool of tasks or works is initially created, and the program then spawns multiple threads, each responsible for picking and executing works from the pool as they become available. The distribution of work is dynamic, allowing for efficient task management.

Work pool implementation

A work queue contains two types of work: compute_local_target and normalize_local_target. The former is responsible for computing the local target and the latter for normalizing the local target. At the program's start, the work queue is initialized only with works of the compute_local_target type.

Each thread begins by being assigned a compute_local_target work. After computing the local target, the thread finds maximum and minimum values within the assigned region, and then inserts a new normalize_local_target work into the work queue. The thread proceeds to pick the next available work from the work queue if the work pool is not empty.

Note that, before any thread can execute a normalize_local_target work, all compute_local_target works must be finished. This is required because a normalize_local_target work requires global maximum and minimum values which depends on local maximum and local minimum. To enforce this, a condition variable is used to block a thread until all necessary regional maximum and

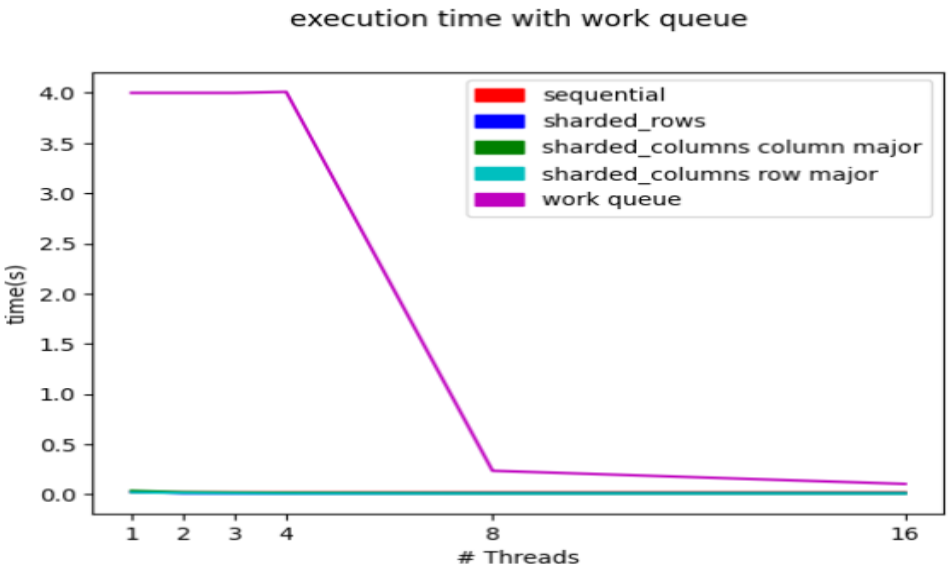

Figure 3: execution time with work queue

minimum values are computed and stored inside a shared array.

Once all required values are computed and available, the last compute_local_target work broadcasts to all idling threads, waking them up to start working. Upon the completion of all works, the program checks if the work pool is empty, then a thread either exits by calling pthread_exit method or moves on to the next work. This implementation ensures the efficient coordination and execution of tasks within the workpool model.

Analysis

This paragraph explains the results presented in Figure 4 and Figure 3. Note that the data for 1, 2, 3 threads are not included in these graphs due to the huge amount of execution time.

In Figure 4, it's observed that the number of L1 cache misses drops significantly from 4 threads to 8 threads. This is due to the fact that chunk size is proportional to the number of threads running. For example, with 4 threads, the image is divided into 4 by 4 blocks, while with 8 threads, it's divided into 8 by 8 blocks. This change in chunk size directly contributes to the reduction in the number of cache misses. When the chunk size is small, there is a higher likelihood of threads sharing the same cache line, increasing the potential for false sharing and leading to many cache misses. Conversely, a larger chunk size ensures that each thread's cache line is filled with data only from its designated image block, reducing false sharing and subsequently reducing cache misses.

This paragraph explains the result shown in Figure 3. The program split

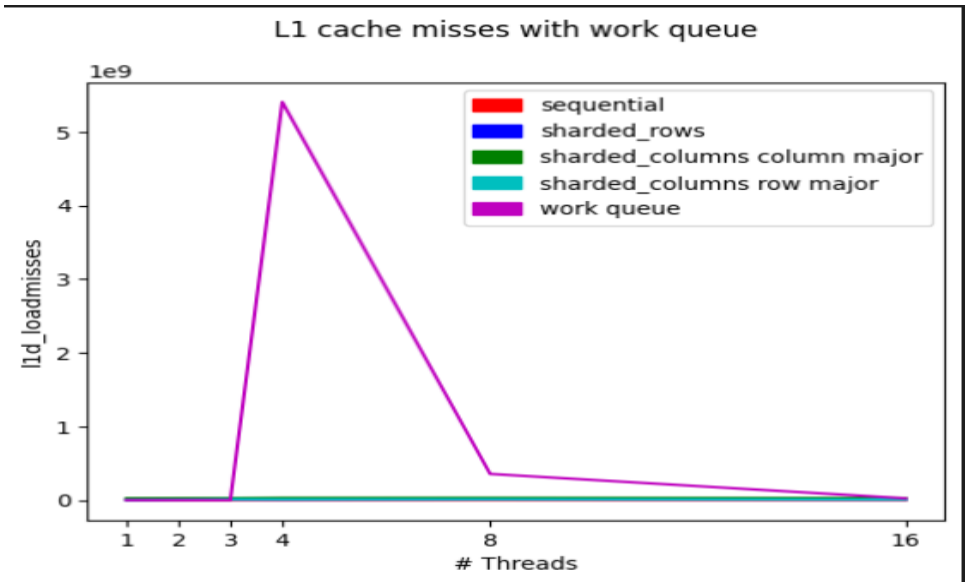

Figure 4: L1 cache misses with work queue

up a 1024 x 1024 image which has more than 1 million pixels. If the chunk size is 1, this program split this image into more than 1 million small tasks, in other words, this splitting method is more fined-grained in terms of task granularity. Running only a single thread would result in a significant amount of time required for execution, even though there may not be much idling time and synchronization overhead. If chunk size remains the same while the number of threads increases, this approach might introduce much more synchronization overhead and context switch as well as more cache contention. On the other hand, using 8 threads with chunk size 8 is more efficient because the program divides the image into larger chunks, which significantly reduces overhead introduced by cache contention and work switching. However, this method still takes longer execution times than other methods due to several reasons. First, the sequential method and the other three splitting method are easier for the computer to identify memory access pattern, hence, prefetching reduce execution time to a great extent. Second, storing temporary results in an array before switching works and the computation of start and end positions for each block contributes to the longer execution time.