مدينة زويل
ZEWAIL CITY
ESTABLISHED 2000
INAUGURATED 2011

مدينة زويل للعلوم والتكنولوجيا
Zewail City of Science and Technology

Parallel and Distributed Computing - Fall 2025

Parallel Edge Detection using the Sobel Operator

By

Habiba Emad 202202091 Wessam Mohamed 202201656 Maya Tarek 202201864
Rawan Wagih 202201168

GitHub Link: https://github.com/mayatarek/image-processing

1. Problem Statement:

Image processing has become one of the most important features of modern AI models, with edge detection playing a crucial role in interpreting visual data. The Sobel Operator is a common technique that finds the gradient, rate of change in direction and brightness of a pixel, and identifies the edges through that.

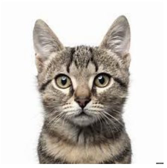
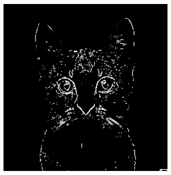
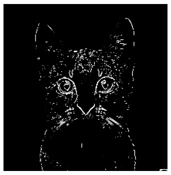
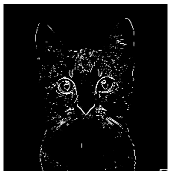
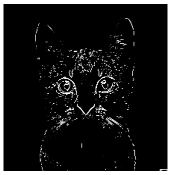
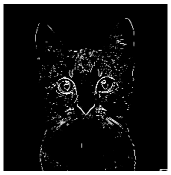
Processing large images can be computationally expensive, especially if the used algorithm is sequential. Sequential algorithms calculating the gradient for each pixel are slow and unreliable. In addition, insufficient memory access reduces the accuracy of edge detection.

With that being said, it is clear that insufficient algorithms negatively impact both the reliability and accuracy of the applications that depend on image processing.

2. Baseline Design:

- A grayscale image is loaded, then resized to 4000x4000 pixels, which is a large resolution
- Sobel Operator:
    - Sobel kernels: Gx for horizontal and Gy for vertical
    - Sobel Operator computes the gradient by looping on each pixel excluding edges, multiplies the 3x3 neighborhood by Gx and Gy, and computes the gradient magnitude
    - Only calculates gradient for pixels that have a 3x3 neighborhood)
    - If the magnitude of the pixel is greater than the threshold, that pixel is considered an edge
- For parallelization:
    - The outer for loops get parallelised using `#pragma omp parallel for collapse(2) schedule(guided)`
    - The inner loop uses `#pragma omp simd` for vectorized execution (performing multiple calculations at a time using a single instruction)
    - The algorithm runs for 1, 2, 4, and 8 threads, measuring execution time for each one
- A cache sensitivity test is done to determine how memory layout affects the performance

Example image:

Original image	Sequential	Parallel 1 thread	Parallel 2 threads	Parallel 4 threads	Parallel 8 threads
					

3. Parallelization approach and OpenMP directives:

Dividing the computational work of computing the gradient of each pixel across multiple threads using OpenMP

- Parallelizing Nested Loops
  - The code have two main loops, one iterates over the rows and one over the columns of an image
  - Using `#pragma omp parallel for collapse(2) schedule(guided),` is telling the system to collapse the two loops into a single one, the guided schedule ensures that work is assigned dynamically and by doing that, idle time is reduced and the load is being balanced.

- Victorization
  - The inner 3x3 multiplication is sped up using `#pragma omp simd`
  - SIMD (Single Instruction, Multiple Data): allow CPU to perform multiple operations at the same time using the single same instruction.

- Thread Control
  - Multiple threads (1, 2, 4, 8) are being experimented dynamically using `omp_set_num_threads(current_thread_count`

4. Performance tables

1. Execution Time, Speedup, and Efficiency for Different Thread Counts

Threads (p)	T_P (s)	Speedup (S= T_S / T_P)	Efficiency (E = S / p)
1	0.380624	1.01218	1.01218
2	0.18963	2.03164	1.01582
4	0.0970778	3.96858	0.992145
8	0.0732466	5.25978	0.657473

T_P → Parallel execution time, Speedup (S) → How many times faster the parallel approach compared to the sequential approach, Efficiency (E) → how well the threads are being used

From the performance table, we can see that:

- Speedup increases by increasing the number of threads (Not linearly).
- Efficiency dropped at using 8 threads, due to overhead (the program spends more time trying to manage the number of threads). Additionally, CPUs have limited cache lines. In “`img.at<uchar>(i+m,j+n)`” inside the loop multiple threads try to access adjacent memory at the same time, causing cache contention, where the threads are being slowed down so CPU can synchronize this access and making sure threads don’t interfere with each other.

2. Sequential cache test

Access Pattern	Time (s)
Row-major (Good locality)	0.239019 s
Column-Major (Bad locality)	2.45575 s

3. Parallel cache test

Threads	Row-Major (s)	Column-Major (s)
1	0.242054 s	2.54429 s
2	0.120642 s	7.3519 s
4	0.0635616 s	8.00386 s
8	0.0555719 s	7.90912 s

From table 2 and 3 we can see that Row-major access is faster than column-major access as it improves cache efficiency by following the image's memory layout. Column-major causes more cache misses. This is due to the image being stored in a row by row manner so when accessed using row by row, we simply find adjacent data. However, with column by column we perform jumps over row to reach the next column which takes more time.

4. Cache misses percentages

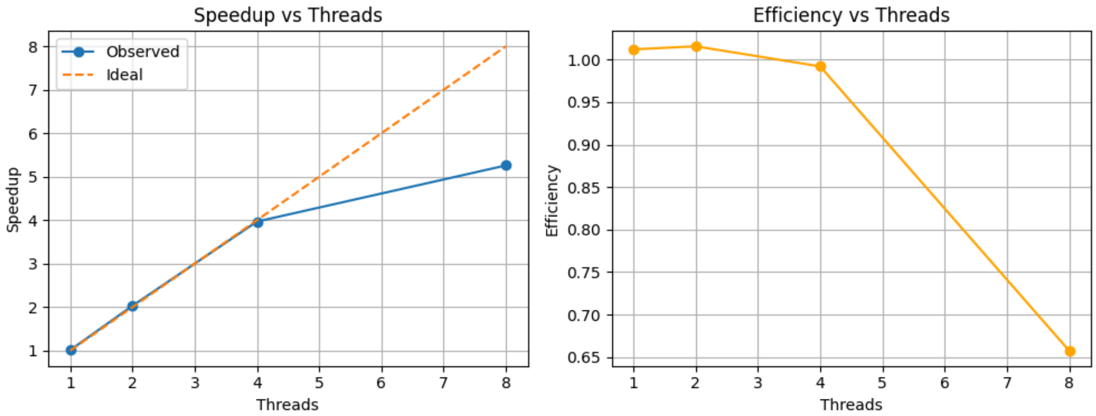
Approach	Sequential row-major	Sequential column-major	Parallel 1 thread row-major	Parallel 1 thread column-major	Parallel 2 threads row-major
Cache misses % of all cache references	8.68%	16.09%	7.40%	17.27%	7.84%

Approach	Parallel 2 threads column-major	Parallel 4 threads row-major	Parallel 4 threads column-major	Parallel 8 threads row-major	Parallel 8 threads column-major
Cache misses % of all cache references	19.83%	8.00%	18.62%	8.43%	25.00%

Table 4 proves that, as mentioned above Row-major access shows less cache misses as it follows the image's memory layout (having good locality), on the other hand the column-major access has higher cache misses (even when changing the number of threads) because of frequent jumping in memory. It was first observed that more threads result in very high cache misses percentages such as 47%, however on inspection it was discovered this was due to false sharing.

As the threads increased, more threads competed over the same cache line wrestling in high misses. This was fixed using “reduction” to remove this problem.

5. Speedup/Efficiency plots



6. Discussion of Amdahl’s and Gustafson’s analysis

As Amdahl’s law states, no matter how much we optimize or parallelize, we can never make a program infinitely faster because we are still limited by the sequential portions of the code. This was observed in our project when we used a small image (400,400), there wasn't much difference in the performance. In fact, many times there would be an overhead with the increase in threads so time ended up being worse. However, when we increased the image size(4000,4000), Gustafson's law became evident as we scaled the problem size, we were able to achieve faster overall performance. Amdahl’s scalability limits tell us no matter how many processors or threads are used, we have a serial fraction that can never be parallel. In this case, f can be approximated using the speed up at 8 threads to give us 7.4%.

7. Reflection: What bottlenecks remain?

- Memory: This problem is more memory bound as it does very little computations
- Thread Overhead: When using many threads on a smaller problem size, the switching between overheads is high as seen in the efficiency plot in the sections above.
- The serial fractions: No matter how many processors or threads we use, serial fractions will always remain impossible to parallelize therefore you can never reduce the time under a certain amount of time
- False Sharing: Multiple threads competing for the same cache line results in cache misses and evictions.