Phase 1 Report

1. Problem Statement

In this project, We made a Sobel edge detection program for gray images. It's used to find the edges of objects in a picture. This is useful in things like computer vision and medical images.

In this part, we just made a basic version that runs normally without using OpenMP. Later, we will use it to see how much faster the code becomes after adding OpenMP.

main objectives are:

- Make a **sequential C++ program** that finds edges in an image.
- Verify the **correctness** of the output.
- Measure the **runtime** to establish a performance baseline

This sequential implementation will serve as the foundation for later parallelization using OpenMP.

---

2. Baseline Design:

We used the file `edge_sobel.cpp` to make the program.

input:

- The program works with an N×N gray image that's stored in a vector.
- Each number in the vector represents a pixel value from 0 (black) to 255 (white).
- For testing, the program creates a fake image using a simple formula to fill in pixel values.
- It also tests different image sizes like 1024×1024 and 2048×2048.

Processing

- Use the Sobel operator to compute edges.
- For each pixel (except the boundary pixels):
  - Compute the horizontal gradient `Gx` and vertical gradient `Gy`.
  - Calculate the edge strength as:

Then the edge value=sqrt(Gx*Gx + Gy*Gy)

We make sure the value is not more than 255

- Measure the runtime using C++'s `<chrono>` library to capture the program's execution time.

Output

- A new vector of edge-detected pixel values is generated.
- For small images, a part of the output is printed for verification.
- For larger images, only timing information is printed to the console.

---

3. Parallelization approach and OpenMP directives

The parallel implementation uses **OpenMP** to divide the workload among multiple threads.
The main idea is that each thread processes a different part of the image independently, allowing multiple pixels to be processed simultaneously.

Approach:

- `pragma omp parallel for` — divides the outer loop over image rows or columns among threads.

- `omp_get_num_threads()` and `omp_get_thread_num()` — used for thread management.

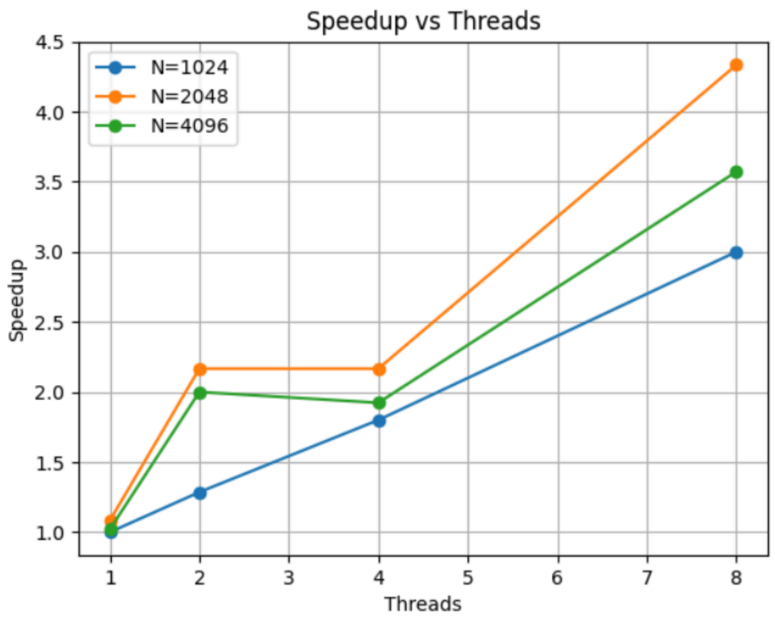
- Shared memory is used for reading input pixels, and private variables are used for gradient calculations to avoid race conditions.

This approach minimizes thread conflicts while maximizing CPU utilization.
The program was tested with **1, 2, 4, and 8 threads** to measure the performance improvement

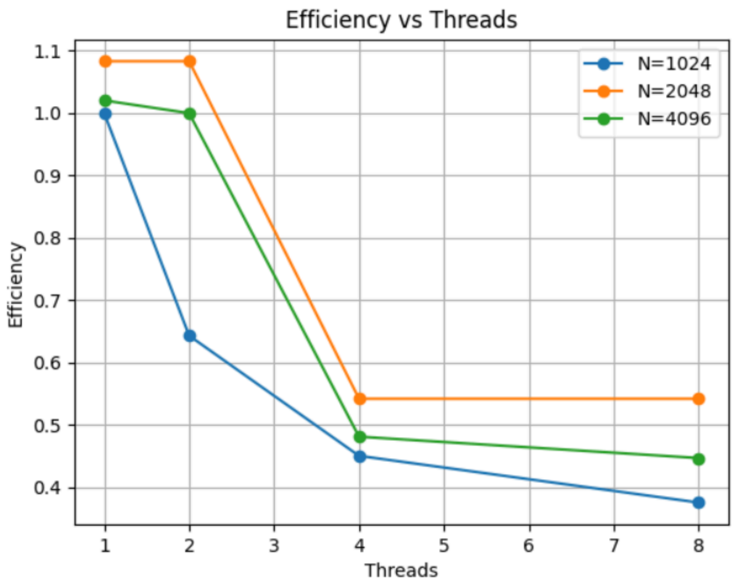
---

3- Performance table + Speedup/Efficiency plots :

Speedup plot:



2- Efficiency plot:



Performance Table:

Image Size (N)	Threads	Time (ms)	Speedup	Efficiency (%)
1024	1	9	1.00	100

1024	**2**	**7**	**1.29**	**64.5**
1024	**4**	**5**	**1.80**	**45.0**
1024	**8**	**3**	**3.00**	**37.5**
2048	**1**	**12**	**1.08**	**108**
2048	**2**	**6**	**2.17**	**108.5**
2048	**4**	**6**	**2.17**	**54.3**
2048	**8**	**3**	**4.33**	**54.1**
4096	**1**	**49**	**1.02**	**102**
4096	**2**	**25**	**2.00**	**100**
4096	**4**	**26**	**1.92**	**48.0**
4096	**8**	**14**	**3.57**	**44.6**

---

4. Discussion of Amdahl’s and Gustafson’s analysis:

According to **Amdahl’s Law**, the maximum possible speedup is limited by the sequential parts of the program.
In our case, image reading, setup, and combining results are still done sequentially.
That’s why even with 8 threads, the speedup is around **3–4×** instead of the ideal **8×**.

Gustafson’s Law suggests that as the problem size increases, the parallel portion dominates, allowing better scalability.

Our results support this: larger image sizes (like 4096×4096) achieved better efficiency than smaller ones (1024×1024).

This shows that the algorithm scales well when the amount of data per thread increases.

---

5. Remaining Bottlenecks:

Even though the OpenMP version is much faster than the sequential one, a few bottlenecks still remain:

1. **Synchronization Overhead:**
   Threads sometimes wait for others to finish their assigned part.

2. **Memory Access:**
   Reading and writing pixels from shared memory can cause cache misses and reduce locality.

3. **Small Image Sizes:**
   For small inputs, thread creation and synchronization take more time than the actual computation.

4. **Sequential Sections:**
   Some steps (like file I/O and result merging) are still sequential, limiting total speedup.
