OpenMP Report: Parallel Image Brightness and Smoothing

With a Comparative Analysis on Small vs. Large Images

Mehul Goyal

November 2, 2025

1 Introduction

The goal of this assignment was to implement a two-stage image processing pipeline using OpenMP to accelerate computation. The two stages are:

1. **Brightness Adjustment:** A fixed offset (`BRIGHTNESS_OFFSET = 50`) is added to every pixel. This is a task with very high ILP.
2. **3x3 Mean Filter:** Each pixel is replaced by the average of its 3x3 neighborhood. This "smoothing" operation is a neighborhood-dependent parallel task.

The performance of a single-threaded serial version was compared against a multi-threaded OpenMP Pragma version. This analysis was performed on two different images to observe the effect of problem size on parallel efficiency:

- **Small Image:** 736x552 (84 KB)
- **Large Image:** 1024x1402 (1.98 MB)

2 Parallelization Strategy

- **Serial Version:** Both stages were implemented using standard C loops. The brightness stage iterates over the 1D pixel array, and the filter stage uses nested loops to iterate over the 2D pixel grid.
- **OpenMP Parallel Version:**
  - **Brightness:** This stage was parallelized using `#pragma omp parallel for schedule(static)`. This is highly efficient as each pixel's new value is independent of all others.
  - **Filter:** This stage was parallelized using `#pragma omp parallel for schedule(static) collapse(2)` on the outer `y` and `x` loops. This flattens the 2D iteration space into a single large parallel region, which is efficient for static scheduling as each pixel's 3x3 kernel calculation takes roughly the same amount of time.

3 Sample Data Output

The following terminal output shows a 5x5 sample from the top-left corner of each image, demonstrating the effect of each processing stage.

```
(base) PS C:\linux restore\open_mp\assignment-4> gcc image_process.c -fopenmp -o image_process
(base) PS C:\linux restore\open_mp\assignment-4> ./image_process
== Image Brightness + 3x3 Smoothing (OpenMP) ==
Loaded image: 1024x1402, Channels: 3

Running Serial baseline... Done in 0.131000 sec
Saved 'output_bright_serial.png' and 'output_filtered_serial.png'

--- Performance Test ---
Threads Parallel_Time(s)        Speedup         Efficiency
------------------------------------------------------------
 1      0.128000                1.02x           1.0234
 2      0.065000                2.02x           1.0077
 4      0.046000                2.85x           0.7120
 8      0.035000                3.74x           0.4679
16      0.027000                4.85x           0.3032
Saved performance data to 'image_performance.csv'
Saved final parallel results to 'output_bright_parallel.png' and 'output_filtered_parallel.png'

--- Original Image (5x5 Sample, Top-Left) ---
113  90  39  84 145
 91  94  95 129 103
 83  97  65  50   8
 94  51  18  33  41
135  19  27  44  37

--- Brightened Image (5x5 Sample, Top-Left) ---
163 140  89 134 195
141 144 145 179 153
133 147 115 100  58
144 101  68  83  91
185  69  77  94  87

--- Filtered (Smoothed) Image (5x5 Sample, Top-Left) ---
163 140  89 134 195
141 135 132 129 134
133 126 120 110 112
144 115  94  85  89
185 106  83  87  93
```

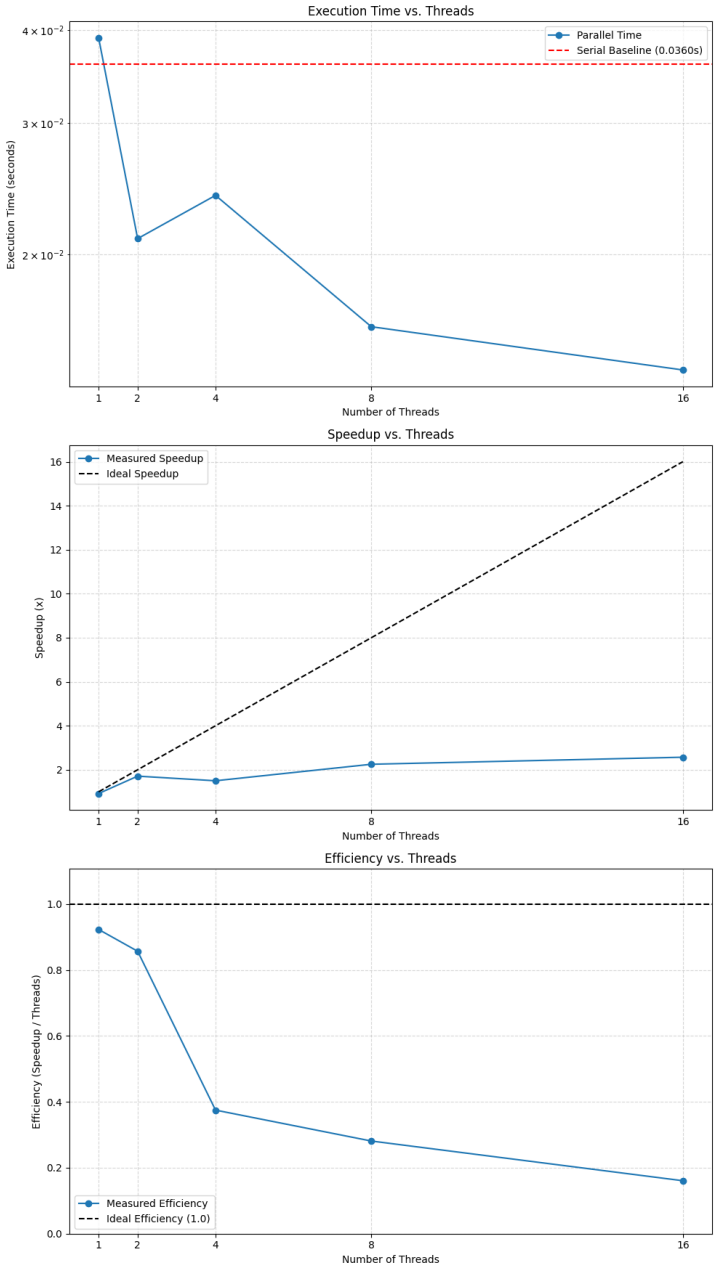
```
(base) PS C:\linux restore\open_mp\assignment-4> gcc image_process.c -fopenmp -o image_process
(base) PS C:\linux restore\open_mp\assignment-4> ./image_process
== Image Brightness + 3x3 Smoothing (OpenMP) ==
Loaded image: 736x552, Channels: 3

Running Serial baseline... Done in 0.036000 sec
Saved 'output_bright_serial.png' and 'output_filtered_serial.png'

--- Performance Test ---
Threads Parallel_Time(s)        Speedup         Efficiency
------------------------------------------------------------
 1      0.039000                0.92x           0.9231
 2      0.021000                1.71x           0.8571
 4      0.024000                1.50x           0.3750
 8      0.016000                2.25x           0.2812
16      0.014000                2.57x           0.1607
Saved performance data to 'image_performance.csv'
Saved final parallel results to 'output_bright_parallel.png' and 'output_filtered_parallel.png'

--- Original Image (5x5 Sample, Top-Left) ---
 63  63  62  61  60
 66  65  64  63  62
 68  67  66  65  63
 68  68  66  65  63
 68  67  66  65  63

--- Brightened Image (5x5 Sample, Top-Left) ---
113 113 112 111 110
116 115 114 113 112
118 117 116 115 113
118 118 116 115 113
118 117 116 115 113

--- Filtered (Smoothed) Image (5x5 Sample, Top-Left) ---
113 113 112 111 110
116 114 114 112 111
118 116 115 114 112
118 117 116 114 113
118 116 115 114 112
```

4 Performance Analysis

The serial and parallel versions were run with 1, 2, 4, 8, and 16 threads.

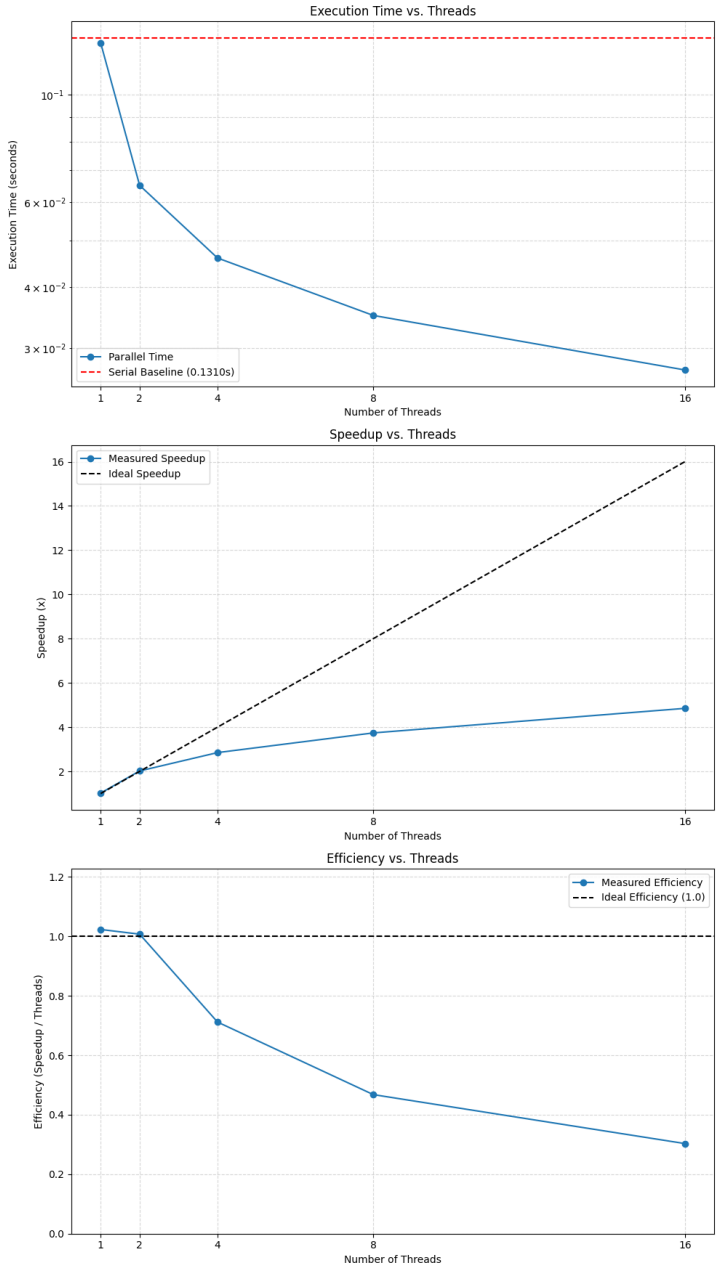
4.1 Small Image (84 KB) Performance

The serial baseline for the small image was **0.036 seconds**. The performance plot shows that while speedup is achieved, it is limited. The efficiency drops off very quickly, falling below 50% at 8 threads. This indicates that for a small problem size, the overhead of creating and managing threads is significant compared to the small amount of parallel work.



4.2 Large Image (1.98 MB) Performance

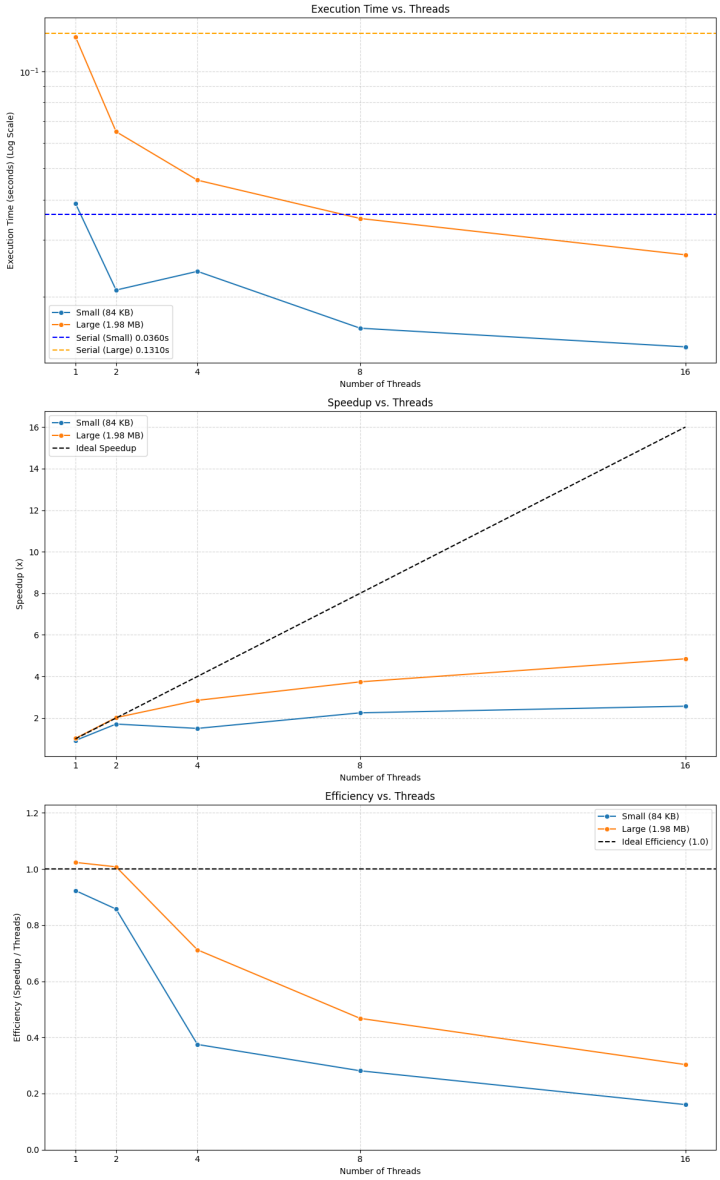
The serial baseline for the large image was **0.131 seconds** (approx. 3.6x longer than the small image). The performance plot shows much better scaling. The speedup continues to increase up to 16 threads, achieving a speedup of **4.85x**. The efficiency remains much higher than in the small image, staying above 70% at 4 threads.



4.3 Comparative Analysis

Plotting both results on the same axes clearly illustrates the impact of problem size on parallel performance.

- **Execution Time:** The execution time taken by the the **Large** image is higher than the **smaller** image but when we compare the difference ine excution time with the serial execution for both the images, The **large** images shows a much higher speedup.
- **Speedup:** The **large** image achieves a dramatically better speedup. At 16 threads, the large image gets a **4.85x** speedup, while the **small** image only gets a **2.57x** speedup.
- **Efficiency:** The efficiency for the large image is much higher at every thread count. This may be because the parallelization overhead was fixed for both the images so Efficiency percentage got higher for the **Larger** image.



5 Conclusion

1. Open MP is highly effective to speedup highly parallel tasks like image processing.
2. This performance benefit is even more useful for larger image tasks, which hints at the success of GPU based image processing in the Deep Learning World.
3. It is also observed that smoothing **degrades** the image quality in **low quality** images while for high quality images, smoothing has little effect on the clarity. This says that we need more efficient methods for smoothing and compressing low quality images rather than simple 3 by 3 kernel smoothing.

(a) High Quality Image: Quality degrades, but features are saved

(b) Low Quality: Features Fade away

Figure 1: Comparison of two images side by side.

6 Image Comparision

The code files and output screenshots for this assignment can be accessed through this repository:
https://github.com/Mehul1729/Parallel-Computing-with-OpenMP