A1 Report

Overview:

Implementing 2D filters efficiently is a critical aspect of image and signal processing, as they enhance the quality of digital images and refine signal data. This report focuses on the development and optimization of an OpenMP-based program designed to accelerate 2D filtering operations, aiming to boost performance and scalability in these essential processing applications.

Improvement in the code

The earlier code which was submitted in A1Q1 had a race condition. In the new code the issue was fixed by introducing memoization, and an output buffer. All the threads are now working on different lines to avoid the race condition.

What is the Function doing?

The filter2d function is designed to apply a 2D convolution operation to a matrix A using a kernel K. The function employs OpenMP for parallel processing, dividing the computation among threads to optimize performance.

To handle edge cases and ensure correctness, the function initializes two boundary buffers, top_memo and bottom_memo. These buffers store the top and bottom rows for each block of rows, which are necessary for processing edge cells during the convolution operation. This setup allows the function to access neighboring rows without conflicts, even when operating in parallel.

The main computation is performed by dividing matrix A into blocks, with each block assigned to a thread. Within each block, rows are processed sequentially to maintain order. For each cell (excluding the boundary rows and columns), a 3x3 submatrix centered on the cell is extracted. This submatrix is then convolved with the kernel K, and the results are summed to compute the new value for the cell.

After the convolution operation, the function ensures that the first row of the matrix remains unchanged. The values of the first row are restored from the top_memo buffer. This preservation step maintains the integrity of the original input matrix's first row.

Overall, the function achieves an efficient parallel implementation by balancing the workload across threads, managing edge cases with boundary buffers, and ensuring sequential processing of rows within blocks.

Code Complexity:

The computational complexity of the code primarily derives from the nested loops that iterate over every element of the n-by-m input matrix. For each element, a fixed-size 3x3 kernel is applied, resulting in a constant amount of work per element. Thus, in a serial execution model, the time complexity is O(n * m). Since the code is parallelized with p threads using OpenMP, the overall workload can be distributed, ideally reducing the effective computation time to O((n * m) / p).

Parallel Code Complexity: O((n*m)/p)

Sequential Code Complexity: O(n*m)

Experimental Setup:

Hardware and Software Requirements

To replicate the readings please use the below mentioned specifications:

- **CPU:** Intel(R) Xeon(R) Gold 6448Y
- **RAM:** 256GB
- **Compiler:** g++ with OpenMP (version 6)
- **Compiler Command:** g++ -g -O2 -fopenmp -std=c++17 a1.cpp -o a1

Input Parameters

The code was tested using a range of matrix sizes (N rows by M columns) and thread counts (OMP_NUM_THREADS). For strong scaling tests, N and M were fixed (e.g., N=10,000, M=10,000) while the number of threads was increased from 1 to 32. For weak scaling tests, N was scaled up proportionally with the number of threads to maintain a constant workload per thread. Each configuration was run multiple times to ensure stable and reliable performance measurements.

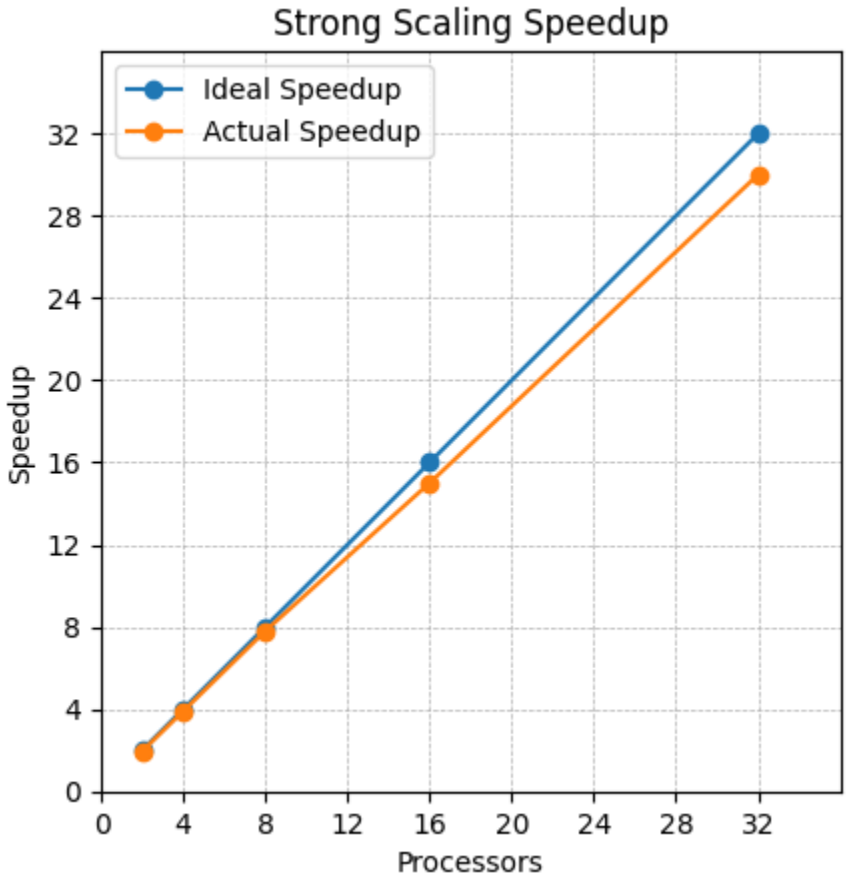
Experiment Results

The table below shows the result of the experiment conducted. These values can now be assessed to understand the scalability and performance of the code with same problem sizes(n=m=10,000) and number of processors. The execution times, reported in seconds, represent the average of three independent runs, ensuring statistical reliability.

Table showing Processors and Runtime n when n=m=10,000			
Processors	Runtime (s)	Speedup	Efficiency
1	3.9	-	-
2	2	1.95	0.98
4	1	3.90	0.98
8	0.5	7.80	0.98
16	0.26	15.00	0.94
32	0.13	30.00	0.94

Strong Scaling Analysis:

The strong scaling results demonstrate that increasing the number of processors significantly reduces runtime while maintaining high efficiency. Starting from 3.9 seconds on a single processor, the runtime nearly halves with two processors and continues to decrease as more processors are added. By the time we reach 32 processors, the execution time is only 0.13 seconds, resulting in a speedup of about 30 times. Throughout these tests, efficiency remains consistently high—mostly above 90%—indicating that adding more computing resources leads to nearly proportional performance gains for the given problem size.



Conclusion

The implementation of a parallelized 2D filtering function using OpenMP demonstrates substantial performance improvements, both in terms of runtime reduction and scalability. By addressing earlier issues such as race conditions and introducing memoization and output buffers, the revised code ensures correctness and efficiency. The use of boundary buffers and thread-specific row processing allows the function to handle edge cases effectively and maximize thread utilization.

Strong scaling tests show that the runtime decreases significantly as the number of processors increases, achieving near-linear speedups with high efficiency (above 90%). With 32 processors, the code achieves a speedup of 30x compared to a single processor, demonstrating its scalability and optimized parallel execution. These results highlight the effectiveness of the implementation in handling large matrix sizes while leveraging modern multicore architectures.