Parallel Image Processing Framework

Course instructor: Dr. Sanjay Saxena

Group members:

- Riyank Singh (202251127)
- Anubhav Singh (202251018)
- Kumar Vishant (202252325)

Problem Description

- Modern high-resolution images require heavy computation for real-time processing.
- Core filters like Gaussian blur, Sobel edge detection, and grayscale conversion become bottlenecks on a single CPU thread.
- Serial execution scales poorly for large images and large datasets.
- This project builds and benchmarks parallel filtering engines to overcome these performance limits.

Methodology

- Implemented image filters using three parallel paradigms: OpenMP, MPI, and CUDA.
- Compared all parallel versions against a single-threaded baseline to measure speedup.
- Results show substantial performance gains for compute-intensive filtering tasks.
- I/O overhead on typical laptops/workstations often dominates runtime, hiding true kernel speedups.

Methods including code highlights

CUDA (GPU Acceleration)

Data is copied from CPU (Host) to GPU (Device) memory, processed by thousands of parallel threads (kernels), and the result is copied back.

```
/* 1. Copy data from Host to Device */
cudaMemcpy(d_input, img.input_host, inSize,
cudaMemcpyHostToDevice);

/* 2. Launch 1000s of parallel threads */
sobel_filter_kernel<<<grid, block>>>(d_input,
d_output, ...);

/* 3. Copy result from Device to Host */
cudaMemcpy(img.output_host, d_output, outSize,
cudaMemcpyDeviceToHost);
```

OpenMP (Shared-Memory CPU)

A compiler directive (#pragma) 'forks' a team of threads that automatically split a loop's work (e.g., image rows) among all CPU cores, accessing one shared memory block.

```
/* "Fork" a team of threads to split the loop */
#pragma omp parallel for
for (int y = 0; y < height; ++y) {
    for (int x = 0; x < width; ++x) {

        // ... (Inner sobel_filter logic) ...

        // Each thread writes to its part of the
        // single shared output array.
        out[y * width + x] = ...;
    }
}
/* Threads "join" and main thread continues */
```

MPI (Distributed-Memory)

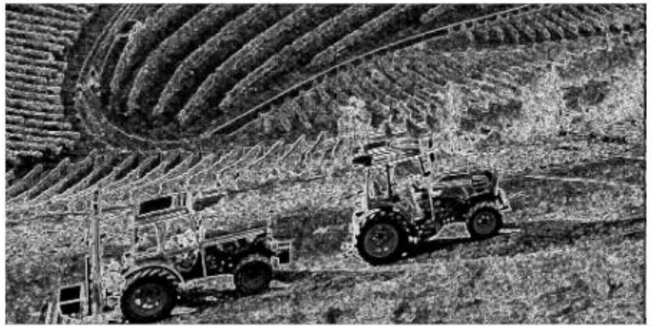
The main process (Rank 0) 'scatters' unique slices of the image to all processes. After computing, Rank 0 'gathers' the results. A 'halo exchange' is needed for edge pixels.

```
/* 1. Rank 0 scatters image slices to all processes */
MPI_Scatterv(full_img, sendcounts.data(), ...,
             local_rgb.data(), ..., 0, MPI_COMM_WORLD);

/* 2. Swap edge rows ("halo") with neighbors */
MPI_Sendrecv(local_gray.data(), ..., above, ...,
             top_halo.data(), ..., above, ...,
             MPI_COMM_WORLD, ...);

/* 3. (All processes compute on their local slice) */

/* 4. Rank 0 gathers results from all processes */
MPI_Gatherv(local_edge.data(), local_edge.size(), ...,
            full_edge.data(), ..., 0, MPI_COMM_WORLD);
```

Results


Original Image

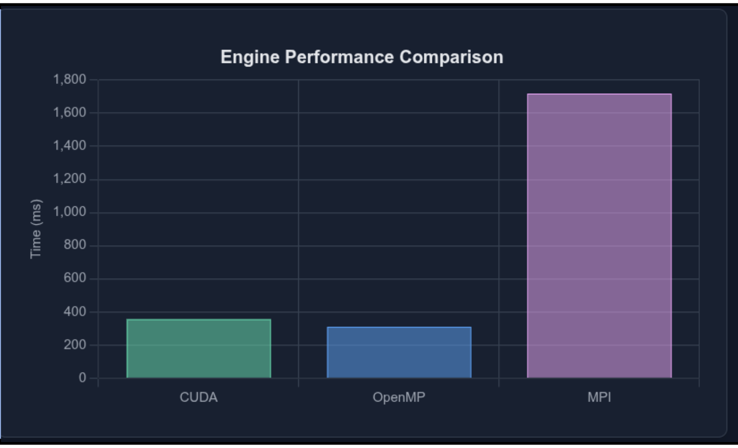

Gaussian Filter


Sobel Filter


Grayscale

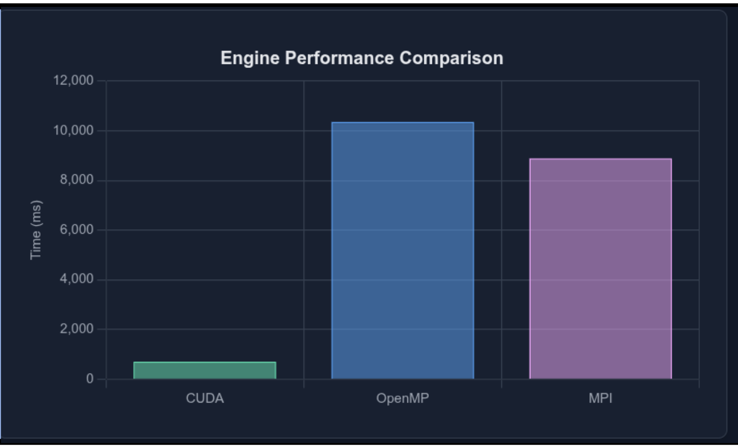
1. Grayscale Filter:

- Cuda – 0.35 secs
- OpenMP – 0.31 secs
- MPI – 1.71 secs



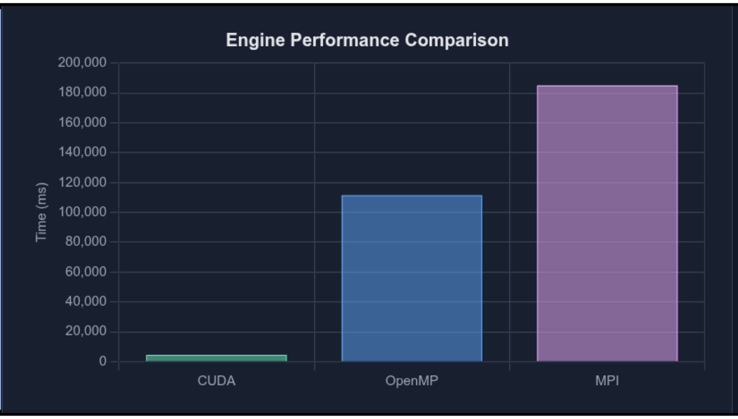
2. Sobel Filter:

- Cuda – 0.7 secs
- OpenMP – 10.34 secs
- MPI – 8.89 secs



3. Gaussian Filter:

- Cuda – 4.474 secs
- OpenMP – 111.54 secs
- MPI – 185.12 secs



Conclusion

- Implemented four compute engines (Single-threaded, OpenMP, MPI, CUDA) for image filtering.
- Parallel methods showed major speedups, especially for complex filters like large Gaussian blurs.
- CUDA delivered the highest performance, often 10×+ faster, proving ideal for heavy, highly parallel computation.
- OpenMP excelled on a single machine due to efficient shared-memory parallelism.
- MPI underperformed on a laptop but is expected to outscale OpenMP on multi-node clusters.
- No universal best method—the ideal approach depends on hardware and workload.
- I/O overhead (image loading/saving) was a major bottleneck and often overshadowed pure compute gains.