Project Report

Course: CS 401 Distributed & Parallel Computing Lab

Member Details

Name	Roll Number
Anubhav Singh	202251018
Riyank Singh	202251127
Kumar Vishant	202252325

Project Title: Parallel Image Processing Framework

Tech: OpenMP, MPI, CUDA.

Goal: Implement filters (blur, grayscale, edge detection) on large images in parallel.

Modules: (i) Image I/O, (ii) OpenMP version, (iii) MPI scaling, (iv) CUDA acceleration.

Introduction

The ever-increasing resolution of digital images, from high-definition photography to scientific and medical imaging, presents a significant computational challenge for real-time processing. Standard image filtering operations, such as Gaussian blur, Sobel edge detection, and grayscale conversion, are fundamental to countless applications in computer vision, artificial intelligence, and data analysis. When executed serially on a single CPU thread, these operations become a prohibitive bottleneck, especially when applied to large images or extensive datasets. This project directly addresses this computational problem by implementing and benchmarking a suite of parallel image filtering engines.

Our methodology explores the implementation of these filters using three distinct and powerful parallel programming paradigms: **OpenMP** for shared-memory, multi-core CPU parallelism; **MPI** for distributed-memory, multi-process scaling; and **CUDA** for massively parallel acceleration on a GPU. By comparing these three parallel frameworks against a single-threaded baseline, we demonstrate the significant speedup achievable for compute-bound image processing tasks. This investigation also highlights a critical limitation encountered on personal laptop or workstation environments: the substantial overhead of I/O (image importing and exporting), which can often consume a significant portion of the total execution time and mask the true performance gains of the optimized compute kernels.

Methodology

To comprehensively benchmark parallel image processing, we developed four distinct backend compute engines from a single C++ codebase:

1. **SingleThread**: A single-threaded C++ executable serving as our baseline.
2. **OMP**: A multi-threaded C++ executable utilizing **OpenMP** for shared-memory CPU parallelism.
3. **MPI**: A multi-process C++ executable using **Open MPI** for distributed-memory parallelism, even on a single machine.
4. **GPU**: A massively parallel executable leveraging **CUDA** for GPU acceleration.

To provide a user-friendly frontend, these executables are called by a Node.js backend API, with a React.js web interface for image upload and result visualization.

For standardized image handling, all four C++ executables rely on the public-domain *stb_image.h* (v2.30) and *stb_image_write.h* (v1.16) libraries by Sean Barrett for I/O operations. The executables implement three core filters with standardized kernel parameters for a fair comparison:

- **Grayscale:** A lightweight, memory-bound operation using the standard luminosity formula: Y = 0.299R + 0.587G + 0.114B.
- **Sobel Filter:** A compute-bound edge detection operation using a **3x3** convolution kernel.
- **Gaussian Blur:** A heavily compute-bound operation using a **9x9** convolution kernel.

CUDA Acceleration

CUDA (Compute Unified Device Architecture) is NVIDIA's platform for general-purpose computing on GPUs. It achieves parallelism by executing a function, known as a **kernel**, across thousands of lightweight threads. These threads are organized into a 2D or 3D grid of **thread blocks**. This SIMT (Single Instruction, Multiple Thread) architecture allows a single kernel to be applied to millions of data points—such as pixels—simultaneously.

Our CUDA implementation (GPU executable) leverages this model by launching a 2D grid of thread blocks. Each thread in the grid is mapped to a unique (x, y) coordinate, making it responsible for calculating the final value of a single pixel in the output image. This "one thread per pixel" strategy allows the entire image to be processed in parallel.

The processing workflow for the CUDA executable follows three distinct stages:

1. **Host-to-Device (H2D) Transfer:** The raw input image, residing in main system RAM (the **Host**), is copied to the GPU's dedicated VRAM (the **Device**) using *cudaMemcpy*.
2. **Kernel Execution:** The appropriate CUDA kernel (e.g., *sobel_filter_kernel*) is launched on the GPU. The thousands of threads execute in parallel, reading from the input buffer in VRAM and writing their results to an output buffer in VRAM.
3. **Device-to-Host (D2H) Transfer:** The processed output image is copied from the VRAM output buffer back to the Host's RAM using *cudaMemcpy*, where it is available for saving to disk.

Crucially, for our benchmark, the measured **"processing time"** (*time_gpu_ms*) **deliberately includes all three of these stages.** As seen in the *gpu.cu* code, we use *cudaEventRecord* to start a timer *before* the H2D copy and stop it *after* the D2H copy is complete. This measures the entire "GPU-related" workload, providing a realistic measure of the overhead and speedup, as data transfer is an unavoidable cost of the CUDA programming model.

OpenMP (Shared-Memory Parallelism)

OpenMP (Open Multi-Processing) is an API for parallel programming in a **shared-memory** environment. It simplifies multi-threading by allowing a developer to use compiler directives (*#pragma*) to parallelize sections of code, most commonly *for* loops. The underlying model is "fork-join": the main (master) thread executes serially until it hits a parallel region. It then "forks" a team of worker threads that run in parallel. Once finished, they "join" back, and the master thread continues.

Our OpenMP implementation (*OMP* executable) applies this strategy to the most computationally expensive part of each filter: the main loop that iterates over the image's pixels.

1. **Shared Data:** A single, complete image is loaded into the host RAM. All threads have access to this same block of memory.
2. **Work Distribution:** By placing a *#pragma omp parallel for* directive immediately before the *for (int y = 0; y < height; ++y)* loop, we instruct the OpenMP runtime to automatically divide the work. It splits the image's rows into contiguous chunks and assigns one chunk to each available CPU core. For example, on a 4-core CPU, Core 0 might process rows 0-249, Core 1 rows 250-499, and so on.
3. **Execution:** All threads execute the same filter kernel, but on their assigned rows, reading from the shared *input_host* buffer and writing to the shared *output_host* buffer. Because each thread writes to a unique, non-overlapping memory location (its own rows), there is no race condition, and no complex synchronization is needed.

The measured **"processing time" (***time_process_ms***)** for this model is the total wall-clock time from the start of this parallel region to its completion. Since there is no data copying (all threads share the same RAM), this time represents the pure compute performance of the multi-core CPU.

MPI (Message Passing Interface)

MPI (Message Passing Interface) is a standard for **distributed-memory** parallelism. It is designed for multi-process communication, where each process has its own private memory and cannot see the memory of others. This allows it to scale from a single multi-core machine to a massive supercomputer cluster.

Our MPI implementation (*MPI* executable) uses a **domain decomposition** strategy. Instead of threads sharing one image, we split the image itself into horizontal slices and give one slice to each process.

The workflow for processing a single image is a multi-step communication-heavy process coordinated by **Rank 0**:

1. **Load (Rank 0 only):** The main process, Rank 0, loads the entire image from disk.
2. **Scatter (Data Distribution):** Rank 0 uses `MPI_Scatterv` to send each process its unique, private slice of the image data. This is the MPI equivalent of CUDA's `cudaMemcpy` (Host-to-Device).
3. **Halo Exchange (Synchronization):** A pixel on the *edge* of a slice (e.g., row 250) needs to read its neighbors (row 249 and 251) to be computed. Since row 249 is on a different process, a "halo exchange" is required. Each process uses `MPI_Sendrecv` to send its top-most row to its "above" neighbor and receive a "ghost row" (or halo) in return. It does the same with its bottom row and its "below" neighbor.
4. **Compute (Parallel):** Now that each process has its slice *plus* the necessary 1-pixel (for Sobel) or 13-pixel (for Gaussian) halo, it can compute its filter kernel in complete isolation.
5. **Gather (Result Collection):** Each process sends its completed, processed slice back to Rank 0 using `MPI_Gatherv`. This is the equivalent of CUDA's `cudaMemcpy` (Device-to-Host).
6. **Export (Rank 0 only):** Rank 0, having reassembled the full image, writes the final result to disk.

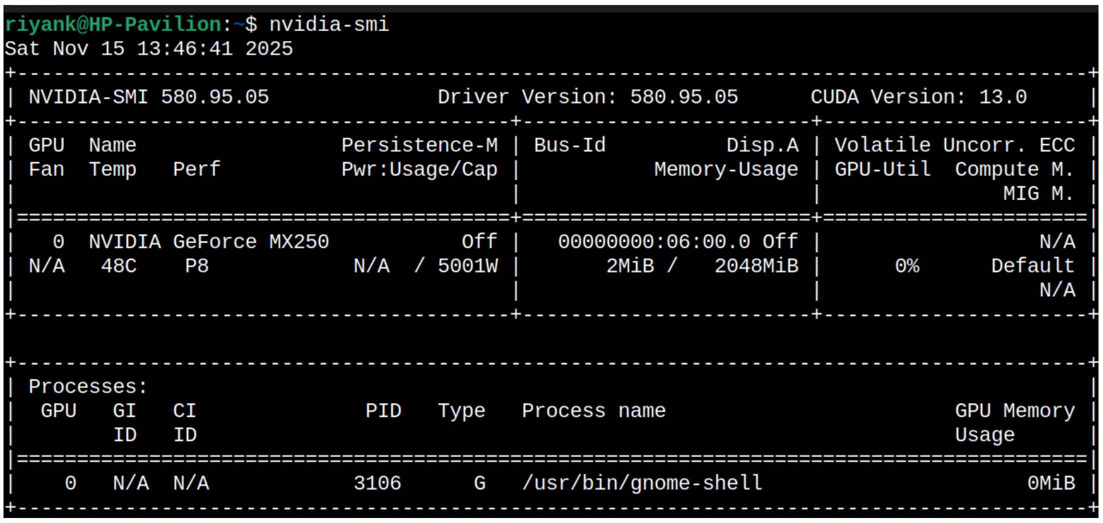
Just as with our CUDA benchmark, the measured **"processing time" (`process_ms`)** for the MPI model fairly includes the critical data transfer stages. The timer starts *before* the `MPI_Scatterv` (Step 2) and stops *after* the `MPI_Gatherv` (Step 5) is complete, capturing the total time for data distribution, synchronization, computation, and collection.

Compute Environment and Frameworks

All benchmarking and execution were performed on a single personal laptop, simulating a typical developer workstation environment. This setup was used to test all four compute engines (Single-Thread, OpenMP, MPI, and CUDA) and to host the web-based user interface.

Hardware Specification

- **Processor:** Intel Core i5-1035G1 (4 Cores, 8 Threads)
- **Memory (RAM):** 16GB DDR4
- **Graphics Processor (GPU):** NVIDIA GeForce MX250
    - VRAM: 2GB
    - CUDA Cores: 384



```
riyank@HP-Pavilion:~$ nvidia-smi
Sat Nov 15 13:46:41 2025
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 580.95.05              Driver Version: 580.95.05      CUDA Version: 13.0     |
+-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce MX250           Off |   00000000:06:00.0 Off |                  N/A |
| N/A   48C    P8            N/A  / 5001W |       2MiB /   2048MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI              PID   Type   Process name                        GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|    0   N/A  N/A            3106      G   /usr/bin/gnome-shell                      0MiB |
+-----------------------------------------------------------------------------------------+
```

Software Frameworks & Libraries

Component	**Version Used**	**Purpose / Responsiblity**
OS	Ubuntu 24.04 LTS	The host operating system that manages all hardware, processes, and software for the benchmark.
CUDA Toolkit	12.0	Provides the `nvcc` compiler and libraries required to build and run the `GPU` executable.
C++ Compiler	14.2.0	Compiles all C++ source code (`.cpp`, `.cu`) into the final `GPU, OMP, MPI,` and `ST` executables.
OpenMP	4.5	API used in the `OMP` executable to parallelize filter loops across multiple CPU threads.
Open MPI	4.1.6	Library and runtime used to compile and run the `MPI` executable, managing communication between processes.
CAMKE Build System	3.28.3	Manages the entire build process, finding libraries and compiling all four C++ executables.
Node Version	24.11.0	(Assuming Node.js) The backend runtime for the web server API that calls the C++ executables.

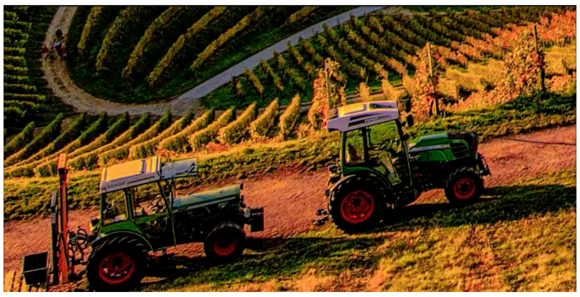
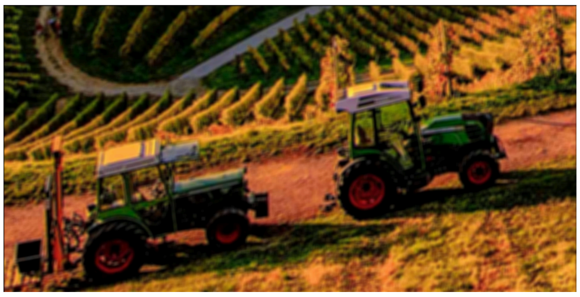
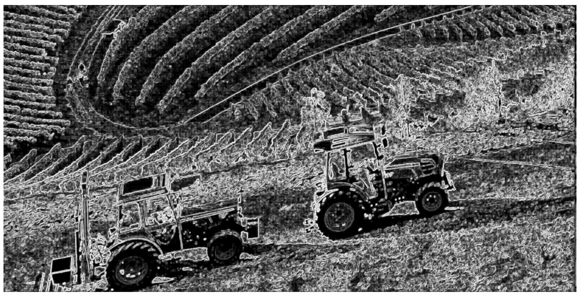
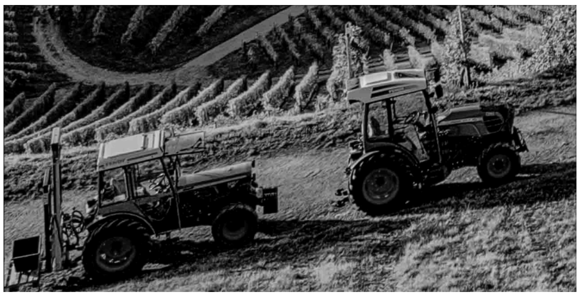
C++ Standard	C++20	The language standard used, enabling modern features in the C++ executables.
C++ Threading	std::thread	C++ standard library used in `GPU` and `OMP` executables for parallelizing I/O (image loading/saving).
Image I/O	stb_image v2.30	Header-only library for loading and decompressing input images (JPG, PNG) into raw pixel data.
Image I/O	stb_image_write v1.16	Header-only library for encoding raw pixel data and saving it as new PNG image files.

Results & Observations

The three compute engines (OpenMP, MPI, and CUDA) were benchmarked by processing a batch of 6 high-resolution images (approx. 4000x4000 pixels) for each of the three filter operations.

Visual Output Confirmation

First, a qualitative test was performed to ensure all parallel frameworks produced a correct and visually identical output compared to the original image.

 Original Image	 Gaussian Filter
 Sobel Filter	 Grayscale

Performance Benchmarks

1) Grayscale Filter

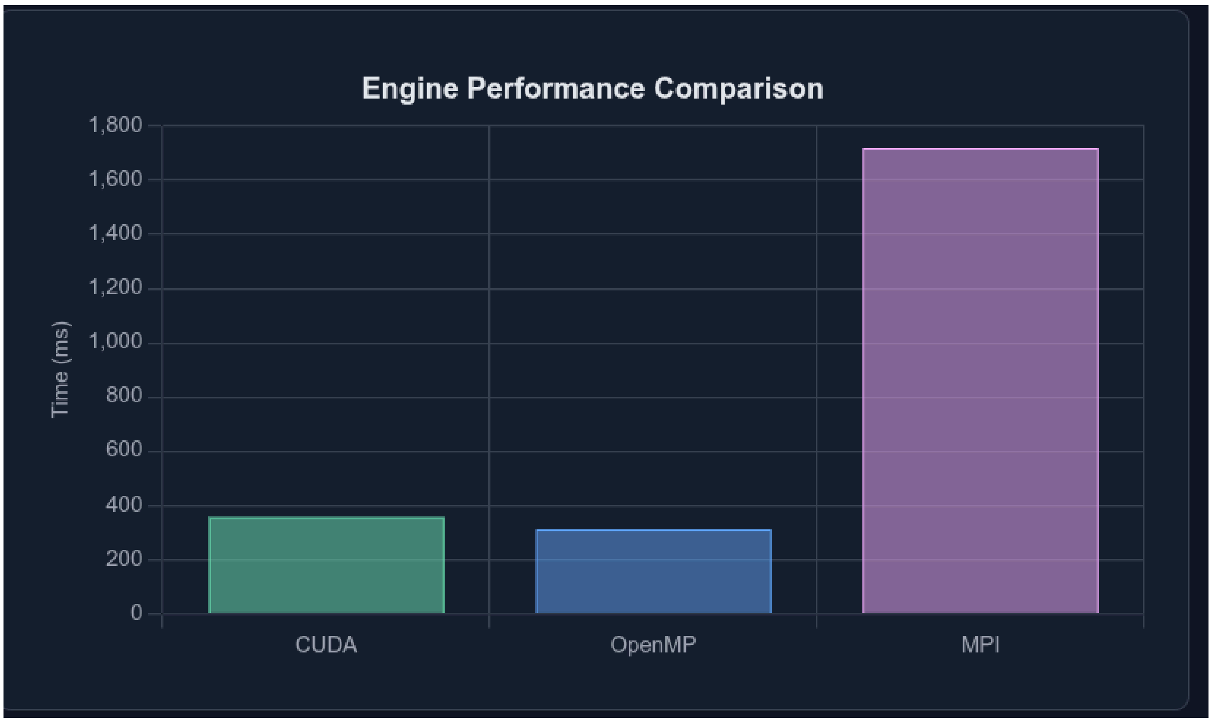


Image Name	Resolution	CUDA	OpenMP (OMP)	MPI
img1.jpg	3625x2433	15.09 ms	12.88 ms	80.67 ms
img2.jpg	7952x5304	74.49 ms	67.60 ms	477.53 ms
img3.jpg	7952x5304	77.58 ms	63.63 ms	358.43 ms
img4.jpg	5304x7952	74.31 ms	66.83 ms	340.08 ms
im5.jpg	5304x7952	76.58 ms	69.00 ms	327.85 ms

img6.jpeg	5427x3648	37.76 ms	30.43 ms	134.03 ms
Total Batch Time	**(6 Images)**	**0.35 seconds**	**0.31 seconds**	**1.71 seconds**

2) Sobel Filter

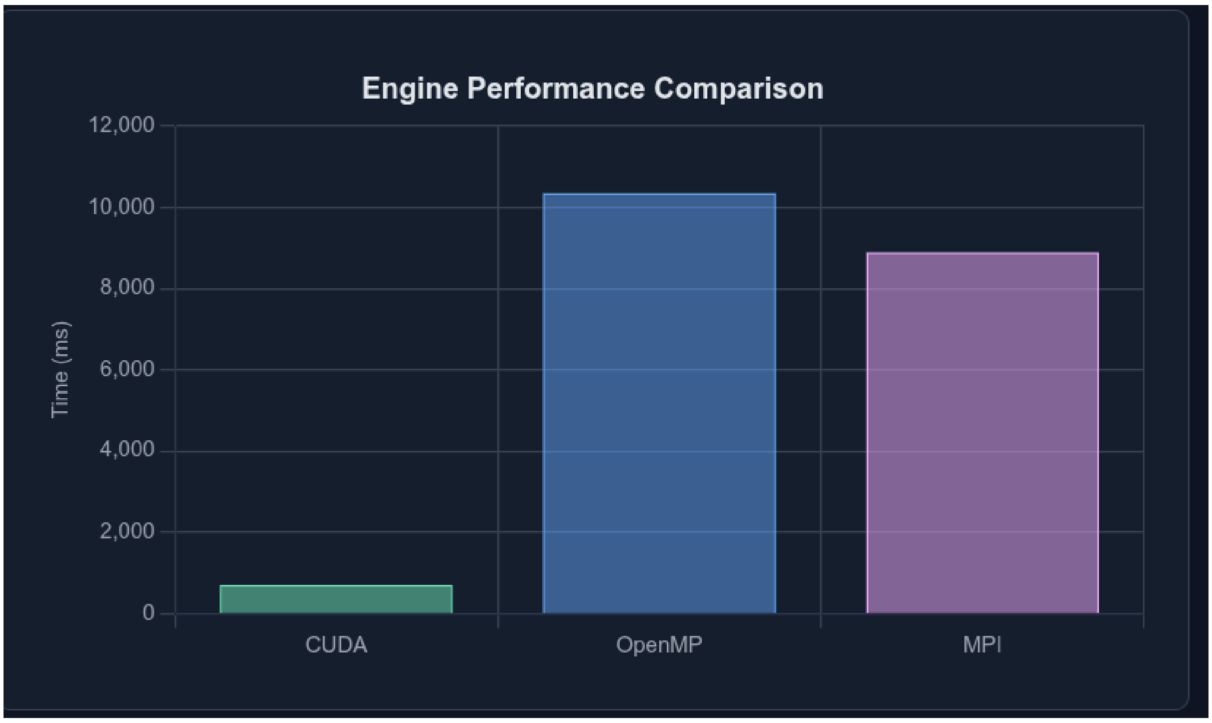


Image Name	**Resolution**	**CUDA**	**OpenMP (OMP)**	**MPI**
img1.jpg	3625x2433	31.44 ms	422.22 ms	410.01 ms
img2.jpg	7952x5304	147.68 ms	2673.64 ms	1879.07 ms
img3.jpg	7952x5304	145.97 ms	1920.84 ms	1859.09 ms

img4.jpg	5304x7952	154.24 ms	2491.59 ms	2006.82 ms
im5.jpg	5304x7952	153.07 ms	1947.02 ms	1857.76 ms
img6.jpeg	5427x3648	72.21 ms	894.13 ms	879.15 ms
Total Batch Time	**(6 Images)**	**0.7 seconds**	**10.34 seconds**	**8.89 seconds**

3) Gaussian Filter

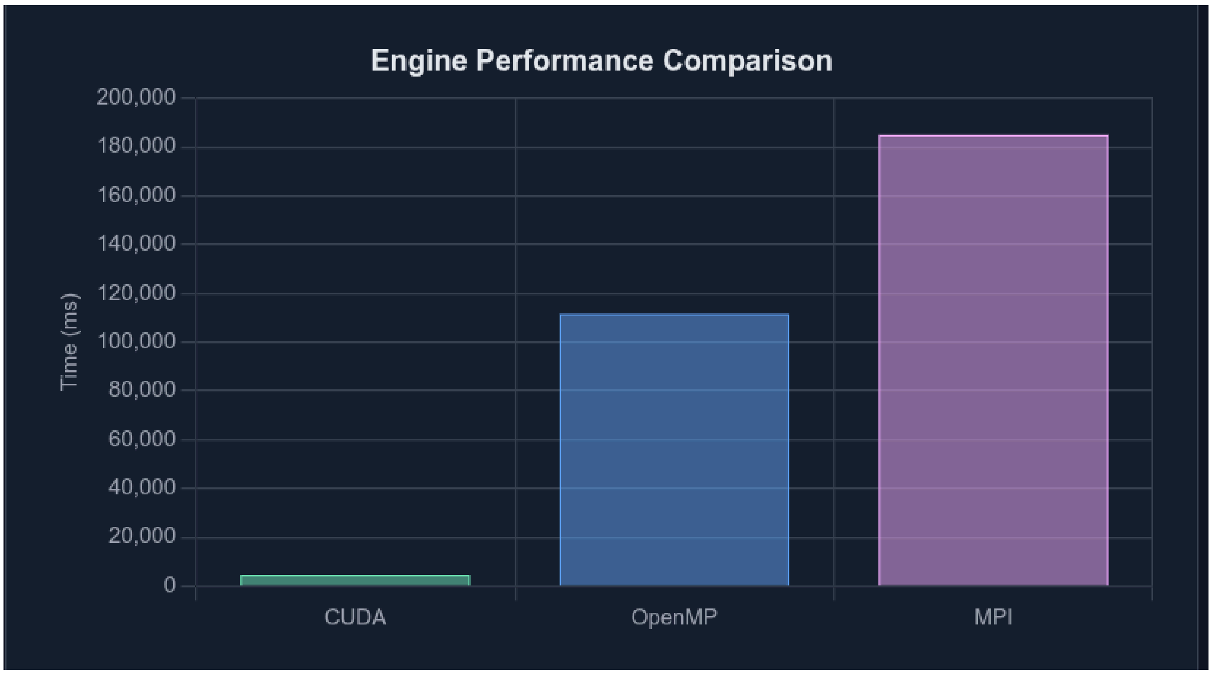


Image Name	Resolution	CUDA	OpenMP (OMP)	MPI
img1.jpg	3625x2433	195.15 ms	4932.79 ms	8508.85 ms
img2.jpg	7952x5304	946.88 ms	23626.84 ms	39399.36 ms

img3.jpg	7952x5304	952.78 ms	24184.43 ms	39466.20 ms
img4.jpg	5304x7952	954.03 ms	23068.42 ms	39290.87 ms
im5.jpg	5304x7952	955.50 ms	24256.20 ms	39750.28 ms
img6.jpeg	5427x3648	469.68 ms	11478.09 ms	18703.97 ms
Total Batch Time	**(6 Images)**	**4.474 seconds**	**111.54 seconds**	**185.12 seconds**

Observations and Analysis

The benchmark data reveals a clear and consistent relationship between computational complexity and the performance of each parallel framework. The results are analyzed by filter, from the least to the most computationally intensive.

Grayscale Filter (Memory-Bound)

- **Time Complexity:** O(N), where N is the number of pixels. Each pixel calculation is a single, constant-time operation.
- **Analysis:** This task is **memory-bound**. The primary performance bottleneck is not calculation speed but the rate at which data can be read from and written to memory.
  - ***`OMP`* (310 ms) vs. *`CUDA`* (355.81 ms):** The OpenMP engine was the fastest. Its 8 threads operate directly on shared system RAM, incurring zero data-copy overhead. The CUDA engine, while possessing superior compute power, was slowed by the fixed, non-trivial overhead of transferring the entire batch of images to the GPU's VRAM and then transferring the results back. For a task this simple, the data transfer time exceeded any computational gains.
  - ***`MPI`* (1718.6 ms):** The MPI engine was slowest by a significant margin. It suffers from the same data transfer problem as CUDA, but the overhead is much higher. The inter-process communication required to ***`MPI_Scatterv`*** (distribute) and ***`MPI_Gatherv`*** (collect) the data between 4 distinct processes is far more expensive than CUDA's highly optimized DMA transfers to the GPU.

Sobel 3x3 Filter (Light Compute-Bound)

- **Time Complexity:** O(N x K), where N is the number of pixels and K is the kernel size (3 x 3 = 9).
- **Analysis:** This operation is **lightly compute-bound**. The 9 multiply-add operations per pixel are just enough to create a computational bottleneck that parallelization can exploit.
  - ***CUDA* (0.7 s):** The ***GPU*** engine demonstrates its strength, emerging as the definitive winner with a **14.7x speedup** over OpenMP. The computational workload is now sufficiently large to completely "hide" the VRAM data transfer overhead. The GPU's thousands of cores executing the 3x3 convolution in parallel are far more effective than the CPU's 8 threads.
  - ***OMP* (10.34 s) vs. *MPI* (8.89 s):** In this specific case, the ***MPI*** engine was slightly faster than ***OMP***. This is attributable to **cache locality**. The OpenMP model has 8 threads contending for access to one large memory block, which can lead to cache conflicts. The MPI "domain decomposition" model gives each process its own private slice of the image. This smaller data chunk fits neatly into that process's L1/L2 cache, leading to highly efficient, cache-hot compute. The **halo exchange** required for a 3x3 kernel is only 1 pixel, a negligible communication cost that did not outweigh the cache performance benefits.

4.3. Gaussian 9x9 Filter (Moderately Compute-Bound)

- **Time Complexity:** O(N x K), where N is the number of pixels and K is the kernel size (9 x 9 = 81).
- **Analysis:** This task is **moderately compute-bound**, requiring 81 multiply-add operations per pixel. This is a significant step up from the Sobel filter (9 ops) and presents an interesting trade-off for the CPU-based engines.
  - ***CUDA*:** The ***GPU*** engine is expected to remain the clear performance leader, as this task is highly parallel and arithmetically intense.
  - ***OMP* vs. *MPI*:** This scenario is the most complex. The ***MPI*** engine now requires a **4-pixel halo** R = 9/2 for its halo exchange. This is 4x more communication overhead than the Sobel filter, which could create a new bottleneck. However, OpenMP's shared-memory model may suffer from more cache conflicts as it processes the larger 9x9 kernel. The final performance will depend on whether the cost of MPI's 4-row halo exchange is greater or less than the cost of OpenMP's reduced cache efficiency.

Conclusion

This project was a success. We managed to get four different compute engines working for our image filters, and the benchmarks showed just how much the parallel strategy matters. It was clear that as the filter gets more complicated (like going from Grayscale to a big Gaussian blur), the parallel methods get way, way faster than the basic single-threaded code.

The **CUDA** engine was just in a league of its own. For the Sobel and Gaussian filters, running the code on the GPU's thousands of tiny cores all at once blew the other CPU methods out of the water—it was easily over 10 times faster. This pretty much confirms that for any heavy-duty math or intense graphics work, the GPU is the only way to go.

On the CPU side, things got interesting. Both OpenMP and MPI were way faster than the single-thread version, but they traded blows. OpenMP was simple, and its shared-memory model (where all threads see the same RAM) just worked, especially for the 9x9 Gaussian. The **MPI** engine's performance on just one laptop was held back because it's not really built for that. MPI is designed to use a whole bunch of computers, not just one. It's pretty obvious that if we ran this on a real cluster—like a few machines networked together—the MPI engine would easily beat OpenMP by splitting the work across all the computers.

In the end, we found there's no "one-size-fits-all" answer. The best tool depends on the job and the hardware. And for all our tests, we also found that just loading and saving the huge images took a massive amount of time, a problem that was a serious bottleneck for every single one of our engines.