High-Performance Image Filtering: MPI, OpenMP, CUDA, and Hybrid Approaches

Adam Benlemlih, Gaby Le Bideau

October 26, 2025

Contents

1 Introduction

This project focuses on the parallelization of an image filtering application for animated GIFs. The starting point was a sequential implementation that applies three successive filters to each frame of a GIF: grayscale conversion, blur, and Sobel edge detection. While this sequential approach is adequate for small images, it becomes prohibitively slow when processing large or numerous frames. Our objective was to develop several parallel implementations to significantly improve performance across different computing architectures. These implementations explore various levels of parallelism, from distributed memory systems using MPI to shared memory parallelism with OpenMP and GPU acceleration using CUDA. We also explored hybrid approaches to maximize resource utilization.

The GIF processing application is a good exercice for parallelization. First, the frame-level is naturally suited for distributed computing, as each can be processed independently. Second, the pixel-level operations within the filters lend themselves to data parallelism, by applying the same operation to different pixels simultaneously. Third, the stencil operations in the blur and Sobel filters can benefit from optimized memory access patterns and specialized hardware acceleration. In our implementations, we begin with a pure MPI approach for distributed computing environments, followed by an OpenMP implementation for multi-core systems. We then look at GPU acceleration with CUDA, which seems promising for the filtering operations. Finally, we develop hybrid implementations that combine these paradigms to create comprehensive parallelization strategies that can adapt to diverse computing environments. During the project, we tried to look not only at raw performance but also scalability, load balancing, and smart resource utilization. Throughout versions, we try to add optimizations targeted at the underlying architecture, while maintaining the correctness and quality of the image processing results.

We note an issue in the original sequential implementation where the blur filter was only applied to approximately 20% of the image (just the top and bottom portions). Our implementations apply the filters uniformly across the entire image. Additionally, we adjusted the threshold values in the Sobel edge detection filter to improve the quality of the output images.

2 Overview of Implementations

2.1 MPI Implementation

The MPI implementation was built in two main stages to make filtering animated GIFs faster. The first stage takes advantage of the fact that many GIFs have multiple frames, and these frames can be processed independently. The root process (rank 0) loads the entire GIF using the provided library, then splits the frames evenly among all available MPI ranks. If the number of frames doesn't divide evenly, the extra frames are spread out one per process to keep the workload balanced. A custom structure tracks which frames go to which process, as well as the number of bytes and their positions in memory. This helps with efficient data transfer using `MPI_Scatterv` and `MPI_Gatherv`, both of which use the `MPI_BYTE` datatype to avoid the overhead of creating custom MPI types. Precomputing all byte counts and displacements also reduces communication time.

To speed up the distribution of metadata, all relevant information—like frame counts, displacements, and indices—is packed into a single buffer and sent using one `MPI_Bcast` call. This avoids the slowdown that can happen with multiple smaller broadcasts. Each process then applies three filters—grayscale, blur, and Sobel edge detection—to its assigned frames independently. These steps are compute-intensive but do not require communication between processes, which keeps the communication overhead low. After all filtering is done, the processed frames are gathered back at rank 0 using `MPI_Gatherv` to rebuild the complete GIF.

When working with a GIF that has only one large frame or when more detailed parallelism is needed, a second method is used. In this version, each frame is split into horizontal slices, with each process handling a portion of the rows. To correctly apply filters like blur and Sobel, each process needs access to the rows directly above and below its slice. This is handled using `MPI_Sendrecv` to exchange so-called "ghost rows" between neighboring processes. Once each process finishes filtering its slice, the results are

sent back to rank 0, which reassembles the full image.

Although it's possible to use MPI I/O to let each process read and write its own part of the image directly, this was not implemented. Because the provided GIF library handles I/O in a specific way, adding MPI I/O would have made the implementation more complicated without clear performance gains.

2.2 OpenMP Implementation

The OpenMP implementation is designed to take advantage of shared-memory parallelism on multi-core systems. Unlike the MPI version, which distributes entire frames to different processes across nodes, this approach focuses on parallelizing operations within a single frame. This allows for fine-grained control and efficient use of processor cores within one machine.

The main strategy involves applying filters—grayscale, blur, and Sobel—by parallelizing loops over image rows and pixels. Within each frame, we used `#pragma omp parallel for` to parallelize row-level processing with static scheduling, which worked well given the regular and predictable nature of the filtering workload. Although we experimented with parallelizing across frames using dynamic scheduling, this part of the code was commented out due to limited performance gains in typical workloads.

To improve memory access patterns, especially for the blur filter, we introduced tiling. The image is divided into smaller blocks (tiles) defined by a `TILE_SIZE` constant. Each tile is processed fully before moving on, which helps reduce cache misses and improves speed by keeping data in fast-access memory.

For the Sobel filter, we reduced repeated calculations by pre-computing row offsets. Instead of recalculating pixel positions for each filter access, we computed base positions once per row:

```
const int row_prev = CONV(j-1, 0, width);
const int row_curr = CONV(j, 0, width);
const int row_next = CONV(j+1, 0, width);
```

This saves time in the innermost loops by avoiding redundant arithmetic.

We also used vectorization to speed up pixel operations. With `#pragma omp simd`, the compiler generates instructions that can process multiple pixels at once using CPU vector units:

```
#pragma omp simd
for (int k = 0; k < width; k++) {
    // Process pixels in a vectorized way
}
```

This improves performance, especially on systems with wide vector registers.

In parts of the blur filter that required checking for convergence, we used parallel reduction to combine results across threads. Using `reduction(|:continue_flag)` allowed each thread to update a shared flag efficiently without the need for explicit locks:

```
#pragma omp parallel for ... reduction(|:continue_flag)
```

Finally, to avoid slow memory access on NUMA (Non-Uniform Memory Access) systems, we ensured that memory buffers were initialized in parallel. Each thread initializes the part of memory it will later use, helping ensure that memory is placed in the local memory region of the thread's CPU:

```
// Initialize buffer with NUMA-aware initialization
#pragma omp parallel for default(none) shared(p, buffer, width, height)
```

2.3 CUDA Implementation

In this implementation we begin with loading the input GIF into host memory, after which each frame is transferred to GPU global memory. To manage large image sizes and optimize memory locality, each

frame is divided into tiles. This tiling enables processing in smaller segments and allows better control over GPU resource usage.

Filtering is performed using three separate CUDA kernels for grayscale conversion, blur, and Sobel edge detection. The grayscale kernel assigns one thread per pixel in a 2D grid. Each thread computes the average of the RGB components and writes the grayscale value back to memory. Since this operation is pixel-local, there are no inter-thread dependencies, allowing for efficient execution.

The blur and Sobel kernels involve neighborhood-based operations and therefore require access to surrounding pixels. A naive implementation would have each thread read its neighbors directly from global memory, resulting in redundant accesses and poor performance. To address this, both kernels employ a shared memory tiling strategy. Each thread block cooperatively loads a tile of the image, including a halo region that captures the required neighboring pixels. This tile is stored in fast shared memory. Threads then synchronize using `__syncthreads()` to ensure all data is available before computation begins. The blur kernel performs a convolution over this shared memory region, reducing global memory traffic and improving throughput.

The Sobel kernel uses the same shared memory approach but focuses on gradient computation. Using only the blue channel for simplicity, each thread calculates horizontal and vertical gradients over a 3×3 window. The magnitude of the gradient is compared against a threshold to generate a binary edge map. The use of shared memory significantly reduces memory latency and improves performance for this compute-bound operation.
To support large images that exceed device memory, we use a two-level tiling strategy. At the host level, images are divided into 1024×1024 tiles, which are processed sequentially. At the device level, each tile is processed using a grid of 16×16 thread blocks, where each block loads and processes a subregion using shared memory.

3 Hybrid Methods

3.1 Hybrid MPI+OpenMP Implementation

To make better use of modern HPC clusters—where each node has multiple CPU cores—we combined our existing MPI and OpenMP implementations into a hybrid model. This design uses MPI to coordinate work across nodes and OpenMP to run computations in parallel within each node, taking advantage of both distributed and shared memory parallelism.

3.1.1 Multi-level Parallelism

The hybrid model applies a two-level parallelization strategy:

- **MPI Across Nodes**: Frames are distributed between nodes using the same method as in the standalone MPI version.
- **OpenMP Within Nodes**: Inside each node, multiple threads handle separate frames or split up a single frame to work in parallel.

This layered approach helps us make full use of all the available CPU cores across the cluster.

3.1.2 Load Balancing

We used a combination of strategies to balance the workload:

- **Frame Assignment**: The master MPI process (rank 0) calculates how to evenly divide frames among MPI ranks, even when the number of frames doesn't divide cleanly.
- **Dynamic Thread Scheduling**: Inside each MPI process, OpenMP threads use dynamic scheduling (`schedule(dynamic)`) to pick up work as it becomes available. This helps balance any variation in how long different frames take to process.

3.1.3 Communication Optimization

To reduce communication overhead, we grouped all metadata needed for scatter operations, like frame counts, displacements, and indices—into a single buffer. This buffer is sent using one `MPI_Bcast` call instead of multiple broadcasts, significantly lowering latency. For the frame data itself, we use `MPI_BYTE` along with pre-calculated displacements to keep the scatter and gather operations efficient.

3.1.4 Thread-safe MPI Setup

Because MPI and OpenMP are used together, we initialize MPI with thread support to prevent conflicts:

```
MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
```

This setup ensures that only the main thread handles MPI communication, while OpenMP threads can still run computations in parallel.

3.1.5 Memory Management

Efficient memory use is important in a hybrid setup, so we applied a few important practices:

- Buffers are pre-allocated before entering any parallel regions.
- Memory is initialized in a NUMA-aware way so that threads access memory close to their CPU cores.
- All resources are cleaned up properly, with error handling in place to avoid leaks or crashes.

3.1.6 Color Quantization

Since GIFs are limited to 256 colors, we apply a simple color quantization step to reduce color depth:

```
const int COLOR_MASK = 0xE0; // Keeps only the top 3 bits
p[i][j].r = p[i][j].r & COLOR_MASK;
p[i][j].g = p[i][j].g & COLOR_MASK;
p[i][j].b = p[i][j].b & COLOR_MASK;
```

This reduces the 24-bit RGB color space to 512 colors, keeping us safely within the format's limitations.

3.2 Hybrid MPI+CUDA Implementation

This version is designed for clusters where each node has one or more GPUs, allowing the program to scale across nodes and make full use of the GPU hardware on each node.

3.2.1 Multi-level Parallelization Strategy

The implementation uses three levels of parallelism:

- **MPI Across Nodes**: Frames are split and distributed to different nodes using MPI.
- **CUDA on Each Node**: Each MPI process runs its filters on an assigned GPU.
- **GPU Threads**: The GPU further splits the work across thousands of threads to process pixel data in parallel.

This setup allows both horizontal scaling (across nodes) and vertical scaling (within each node via GPU parallelism).

3.2.2 GPU Device Management

Each MPI process is assigned a GPU based on its rank to make efficient use of available hardware:

```
/* Set GPU device for this MPI process */
if (gpu_count > 0) {
    int gpu_id = rank % gpu_count;
    cuda_err = cudaSetDevice(gpu_id);
}
```

This ensures that MPI processes are distributed evenly across all available GPUs, which is especially important in multi-GPU systems. We also added error checking to detect and handle any GPU initialization problems gracefully.

3.2.3 Broadcast Optimization

To reduce the cost of metadata communication, we pack all necessary scatter arrays into one buffer and broadcast it in a single call:

```
// Pack all scatter arrays into a single buffer for efficient transfer
memcpy(&all_scatter_info[0], scatter_data->image_counts, size * sizeof(int));
memcpy(&all_scatter_info[size], scatter_data->image_displs, size * sizeof(int));
// ... more copying ...
MPI_Bcast(all_scatter_info, total_scatter_info_size, MPI_INT, 0, MPI_COMM_WORLD);
```

This strategy significantly reduces communication overhead, which becomes more noticeable when many GPUs are involved.

3.2.4 Asynchronous Processing with CUDA Streams

To keep the GPU busy and avoid loss of time, we use CUDA streams for asynchronous kernel launches:

```
// Apply blur kernel with shared memory
blur_kernel<<<gridDim, blockDim, sharedMemSizeBlur, stream>>>(
    d_pixels, d_temp, width, height, blurSize);
```

This lets the GPU overlap computation with memory transfers.

3.2.5 Memory Management Optimizations

We applied the following to keep GPU operations efficient:

- Buffers are allocated once with enough space to cover all expected data.
- All memory operations are wrapped in error checks.
- Reused buffers avoid unnecessary reallocations between steps.
- Synchronization points ensure that operations finish in the correct order.

3.2.6 Adaptive Block Sizing

Since different GPUs have different shared memory limits, we adapt the kernel's block size if needed:

```
if (sharedMemSizeBlur > deviceProp.sharedMemPerBlock) {
    // Fall back to a smaller tile size
    blockSize.x /= 2;
    blockSize.y /= 2;
}
```

This ensures the kernels run correctly and efficiently across a range of GPU architectures.

3.2.7 Collective Communication Optimization

To send and receive pixel data efficiently, we use optimized MPI collective operations with pre-calculated displacements:

```
MPI_Scatterv(all_pixels, scatter_data->scatter_byte_counts,
            scatter_data->scatter_byte_displs, MPI_BYTE,
            scattered_pixels, sendcount * sizeof(pixel), MPI_BYTE,
            0, MPI_COMM_WORLD);
```

Using byte-level transfers and scatter/gather patterns helps reduce communication overhead.

Overall, the MPI+CUDA hybrid implementation delivers good performance on GPU-equipped clusters. It effectively splits work between nodes and uses GPU parallelism to handle filters. This makes it well-suited for processing high-resolution or multi-frame GIFs (see part 4).

3.3 Triple Hybrid MPI+OpenMP+CUDA Implementation

3.3.1 Four-Tier Parallelization Architecture

The implementation uses a hierarchical parallel design:

- **MPI (Across Nodes)**: Distributes frames among different nodes in the cluster.
- **OpenMP (Across CPU Cores)**: Enables thread-level parallelism within each node.
- **CUDA (Across GPUs)**: Offloads compute-heavy tasks to one or more GPUs on each node.
- **GPU Threads**: Launches thousands of threads per GPU using 2D thread blocks for pixel-level operations.

This layered approach scales across all levels of hardware resulting in both horizontal and vertical parallelism.

3.3.2 Adaptive CPU/GPU Processing

A key feature of this implementation is its ability to choose between CPU and GPU processing dynamically, based on image size and hardware availability:

```
// Decide whether to use CPU or GPU
bool use_gpu = false;
if (pixel_count >= MIN_SIZE_FOR_GPU && my_gpu_count > 0) {
    // Try GPU processing with fallback to CPU
    try {
        process_image_cuda(...);
        use_gpu = true;
    } catch (...) {
        fprintf(stderr, "Warning: GPU processing failed, falling back to CPU\n");
    }
}
if (!use_gpu) {
    process_image_cpu(...);
}
```

This ensures small images stay on the CPU to avoid unnecessary memory transfer, while large or compute-intensive frames are accelerated using the GPU. If GPU execution fails, the system falls back to CPU processing.

3.3.3 Dynamic Resource Management

To handle environments with varying GPU availability, we dynamically assign GPUs to MPI processes:

```
// Distribute available GPUs among MPI processes
int gpus_per_process = (gpu_count + size - 1) / size;
int start_gpu = min_int(rank * gpus_per_process, gpu_count);
int end_gpu = min_int(start_gpu + gpus_per_process, gpu_count);
my_gpu_count = end_gpu - start_gpu;
```

This allocation ensures fair use of all GPUs, even on clusters with uneven GPU distribution. For CPU-side parallelism, OpenMP threads use dynamic scheduling:

```
#pragma omp parallel for schedule(dynamic)
for (int i = 0; i < my_n_images; i++) {
    // Process frames
}
```

This helps to equalize the workload when frame sizes or processing times vary.

3.3.4 Broadcast Optimization

To reduce communication overhead, we bundle all scatter metadata into a single buffer and broadcast it once:

```
// Pack all scatter arrays into a single buffer
memcpy(&all_scatter_info[0], scatter_data->image_counts, size * sizeof(int));
memcpy(&all_scatter_info[size], scatter_data->image_displs, size * sizeof(int));
// ...
MPI_Bcast(all_scatter_info, total_scatter_info_size, MPI_INT, 0, MPI_COMM_WORLD);
```

This cuts down on latency and simplifies coordination between CPU and GPU.

3.3.5 Memory Optimization

- **GPU Memory**: Shared memory is used to reduce global memory traffic during stencil operations.
- **Host Memory**: Buffers grow dynamically to avoid repeated reallocations.
- **Cache Usage**: Tiling strategies improve CPU cache locality during filtering.
- **NUMA Awareness**: Memory is initialized close to the threads that will use it.

3.3.6 Error Handling and Resilience

Robust error handling ensures stable execution even in failure scenarios:

- CUDA errors are checked and reported with detailed messages.
- GPU failures trigger automatic fallback to CPU processing.
- MPI uses abort calls to cleanly exit on critical failures.
- All memory and GPU resources are cleaned up properly to avoid leaks.

3.3.7 OpenMP-CUDA Integration

A interesting aspect is how we integrate OpenMP threads with CUDA devices and streams:

```
int gpu_idx = omp_get_thread_num() % my_gpu_count;
int stream_idx = omp_get_thread_num() / my_gpu_count;
```

Each OpenMP thread is mapped to a specific GPU and stream, in order to have concurrent GPU execution across multiple streams and devices.

4 Performance Evaluation

4.1 Experimental Setup

Hardware and Software Details: The experiments were conducted on a system with the following configuration:

- **CPU:** 13th Gen Intel® Core™ i7-13700K with 24 logical cores.
- **Memory:** 64 GB DDR4 RAM.
- **GPU:** NVIDIA RTX A4000 featuring 6144 CUDA cores and 16 GB GDDR6 memory.
- **Operating System:** Ubuntu 20.04 LTS.
- **Software:**
  - CUDA Toolkit 12.6
  - OpenMPI 4.0.2
  - GCC 9.3.0 with OpenMP support.
- **Libraries:** the provided `gif_lib` is used for GIF manipulation.

Benchmark Dataset:

- **Images:** We used the provied set of GIF images (both single-frame and animated). Resolutions range from 320×200 to high-resolution images. Animated GIFs vary in the number of frames (from 1 up to 30).
- **Synthetic Datasets:** Two additional datasets were generated:
  - **Increasing Resolution:** Images in the `generated/increasing_size` folder with resolutions such as 100×100, 200×200, 400×400, and 800×800.
  - **Increasing Frame Count:** Animated GIFs in the `generated/increasing_frames` folder with progressively more frames.

Metrics:

- **Execution Time:** Total runtime (in seconds) for each implementation (sequential, MPI, OpenMP, CUDA, and hybrid).
- **Speedup:** The ratio of sequential time to parallel execution time.
- **Efficiency:** Speedup normalized by the number of processes or threads.

4.2 Results and Graphs

Figure 1 presents an overview of the measured speedups for each implementation (sequential, OpenMP, MPI, MPI+OpenMP, CUDA, CUDA+MPI, and Hybrid MPI+OpenMP+CUDA) across a test datasets. The height of each bar represents the speedup factor relative to the sequential baseline, so higher values indicate better performance. We see that GPU-based or hybrid GPU approaches often yield the largest speedups, particularly for large images, though CPU-based approaches (MPI or MPI+OpenMP) remain competitive in some cases.

To provide more detail, Figure 2 breaks down the speedup results by each dataset individually. From these plots, one can see that certain implementations are favored depending on whether the image is large and single-frame (where CUDA or CUDA+MPI may shine) versus smaller or multi-frame (where MPI+OpenMP or the triple-hybrid approach do very well).

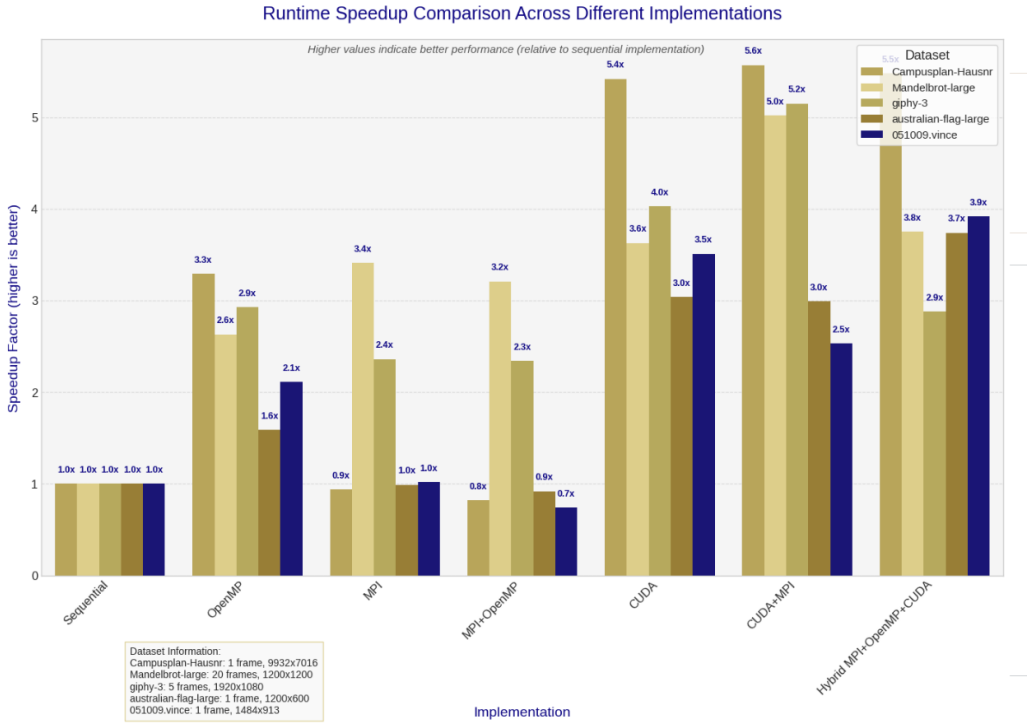

Figure 1: **Runtime Speedup Comparison Across Different Implementations.**

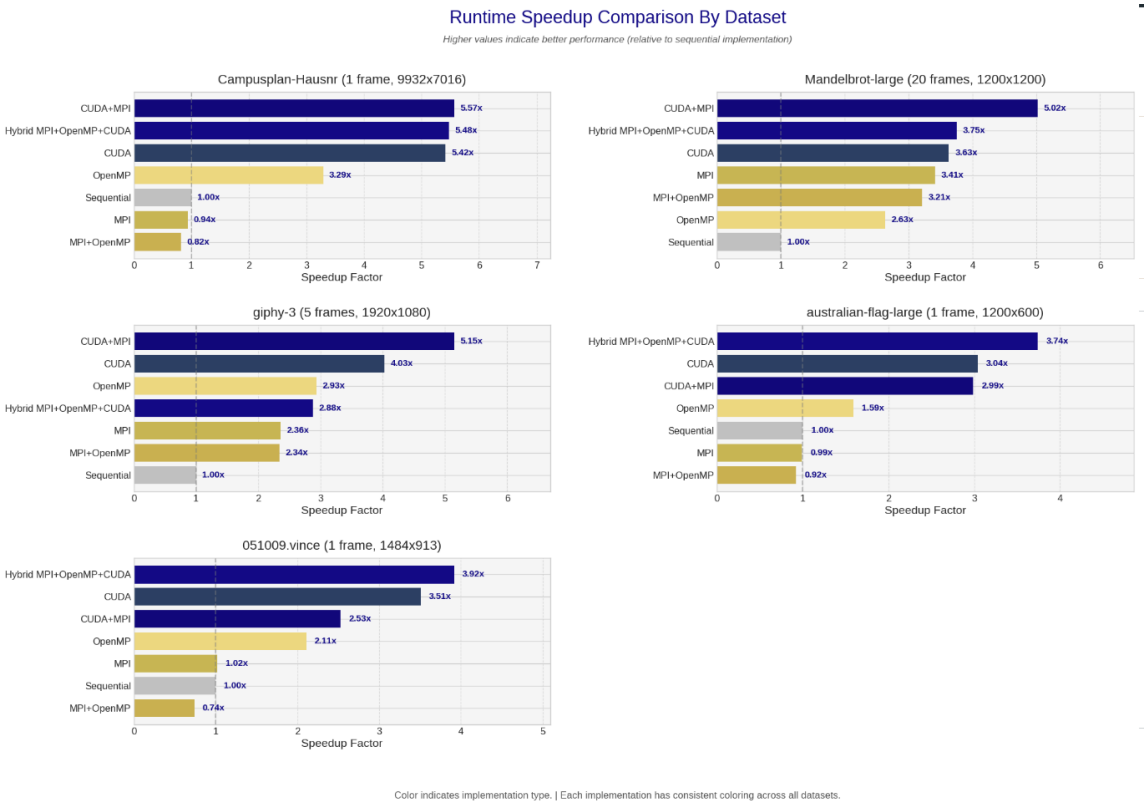

Figure 2: **Runtime Speedup Comparison by Individual Dataset.**

Performance on the Generated Noise Dataset: This plot shows how the different implementations scale with increasing image resolution. The speedup is relative to the sequential baseline (i.e., higher is better):

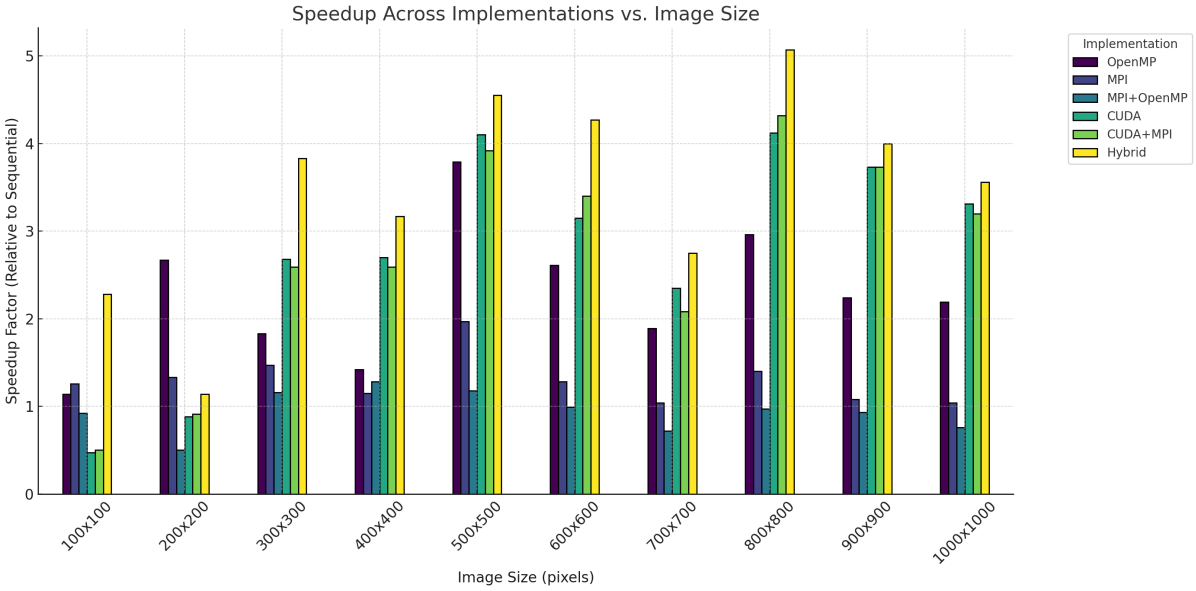

Figure 3: **Speedup vs. Image Size (Generated Noise Dataset).**
This benchmark measures the speedup factor across increasing image resolutions for each implementation using synthetic single-frame noise images.

Small Images (100x100 to 300x300):

- Hybrid implementation already achieves good speedup (2.28x to 3.83x).
- CUDA-only and CUDA+MPI underperform for very small sizes due to kernel launch and data transfer overheads.
- OpenMP and MPI show modest gains, but MPI+OpenMP struggles due to overhead outweighing parallel benefit.

Medium Images (400x400 to 700x700):

- CUDA-based implementations start to show strong performance (CUDA hits 4.10x at 500x500).
- Hybrid consistently outperforms all others in this range, peaking at 4.55x at 500x500.
- OpenMP remains efficient (around 2–3x), while MPI+OpenMP stays below baseline for some cases due to coordination cost.

Larger Images (800x800 to 1000x1000):

- Hybrid dominates again, reaching 5.07x at 800x800.
- CUDA+MPI performs comparably to CUDA alone.
- MPI-only shows limited gains, implying communication costs dominate at larger data sizes unless complemented by GPU acceleration.

Analysis of Frame Size Scaling Performance: Figure 4 illustrates how each implementation scales with increasing frame size for a single-frame, noise-filled image. Several trends can be observed:

- **Small Images (100x100 to 300x300):** The hybrid implementation already demonstrates a noticeable speedup, while CUDA and CUDA+MPI lag due to GPU launch overheads and data transfer latency. MPI+OpenMP underperforms due to synchronization costs dominating over actual computation time.

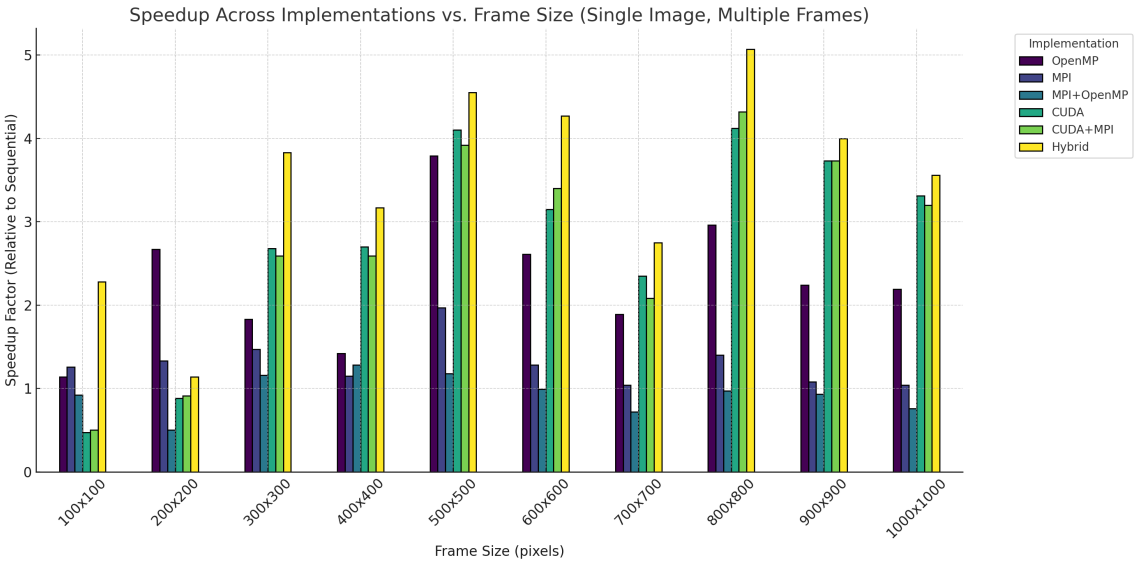

Figure 4: **Speedup Across Implementations vs. Frame Size.**
This plot shows the relative speedup (compared to sequential) for each implementation as the image resolution increases. Higher bars indicate better performance.

- **Mid-sized Images (400x400 to 700x700):** This is where CUDA-based implementations start to shine, with CUDA reaching over 4x speedup at 500x500 resolution. The hybrid approach consistently delivers the highest performance, taking advantage of both CPU and GPU parallelism. OpenMP and MPI show moderate gains.
- **Large Images (800x800 to 1000x1000):** Hybrid MPI+OpenMP+CUDA continues to outperform all other methods, peaking at 5.07x speedup. CUDA and CUDA+MPI also achieve strong results, while MPI-only implementations plateau due to communication bottlenecks.

5 Adaptivity Strategy

The adaptive filtering logic implemented in our project is based on the performance trends observed during testing. As seen before, no single parallelization strategy is optimal for all scenarios. Instead, the best choice depends on the specific input characteristics and available hardware.

The data and hardware-aware decision-making process to select the most suitable backend is set-up in the `adaptive_filter.c` file, where execution mode is determined either from user input or via an automatic system if the `auto` mode is selected or no flag is set.

5.1 Hardware Resource Detection

When the program starts, it first checks the computer's resources:

- It finds out how many processing units are available (such as CPU cores and, if running in a distributed setting, how many MPI processes are active).
- It also looks for any available GPUs by asking the system if any CUDA-enabled devices are present.

This step helps decide whether to use GPU acceleration, spread the work across multiple processes, or rely on multithreading.

5.2 Input Set Analysis

Next, the program examines the input GIF:

- It counts the number of frames in the GIF.

- It measures the size of the images (their width and height).

For instance, a single very large image may benefit more from GPU processing, while a GIF with many smaller frames might be best handled by dividing the work among several processes.

5.3 Adaptive Parallelization Approach

- **Large, Single-Frame Inputs:** Benchmark results show that GPU-based approaches (CUDA or Hybrid CUDA+MPI+OpenMP) provide superior speedup on large images. The automatic mode detects high-resolution inputs (via pixel count) and selects CUDA if a GPU is present, or MPI/OpenMP hybrids otherwise.
- **Multi-Frame Inputs:** For animated GIFs, splitting work across MPI ranks improves throughput. Our adaptive logic detects the number of frames and prefers MPI or hybrid strategies accordingly. If GPUs are available, it selects Hybrid CUDA+MPI+OpenMP for best performance.
- **Small Inputs:** For small images (e.g., 100×100), benchmarking revealed that the CUDA and MPI overhead can outweigh the benefits. In these cases, our logic defaults to lightweight OpenMP or even sequential mode to avoid unnecessary resource activation.

Each strategy is encapsulated in a separate function (e.g., `run_cuda_filter()`, `run_mpi_domain_filter()`), which makes our code adaptable.

6 Conclusion

In this project, we explored a wide range of parallelization strategies to accelerate image filtering on animated GIFs, starting from a sequential baseline and progressing through MPI, OpenMP, CUDA, and several hybrid combinations. Each implementation targeted a different aspect of parallel computing—from distributed memory systems to shared memory multithreading and GPU acceleration.

Our results demonstrate that no single strategy is optimal across all scenarios. MPI and OpenMP provide solid performance on CPU-bound systems, while CUDA offers substantial speedups for large images. Hybrid models, particularly the triple hybrid MPI+OpenMP+CUDA implementation, deliver the best performance by leveraging the strengths of each platform simultaneously.

In the future we could think about including more advanced auto-tuning for hardware-aware kernel configuration, support for asynchronous I/O, and further extension of the adaptive logic to real-time scenarios.