Real-time video processing

Visual Computing Assignment

Mads Pagh
Student ID: 202208375
Aarhus University
GitHub Repository

October 30, 2025

Abstract

This report investigates CPU versus GPU performance for real-time video filtering. The research question: *Does GPU-accelerated image processing consistently outperform CPU implementations for webcam video streams?* Five filters (grayscale, blur, edge detection, pixelation, comic art) were implemented using both CPU (OpenCV) and GPU (OpenGL shaders), tested across three resolutions. Results show GPU advantages only for complex multi-stage filters; simple operations show negligible speedup due to memory transfer overhead.

Contents

1 Methodology

1.1 System Architecture

The system implements a three-stage pipeline for real-time video processing and performance evaluation. Video frames are captured from the system webcam using OpenCV's VideoCapture interface, which provides raw RGB data at the camera's native resolution and frame rate. These frames then flow through one of two parallel processing pipelines: CPU-based processing using OpenCV's highly optimized image manipulation functions, or GPU-based processing through OpenGL fragment shaders written in GLSL 3.3. Both pipelines implement identical filter algorithms (Table 1) to ensure fair comparison.

Table 1: Implemented Filters

Filter	**Description**
None	Pass-through
Grayscale	Luminance conversion
Gaussian Blur	5×5 kernel
Edge Detection	Sobel operator
Pixelation	10×10 blocks
Comic Art	Edge + quantization

The rendering stage uses OpenGL for display on all tests. For GPU processing, frames remain in GPU memory throughout the pipeline (capture → process → render), eliminating redundant data transfers. For CPU processing, filtered frames must be uploaded from system RAM to GPU memory via `glTexImage2D()` before rendering. The difference is highlighted in the performance characteristics observed in the benchmarks, as the CPU pipeline incurs per-frame memory transfer overhead that the GPU pipeline avoids. The system supports dynamic runtime switching between processing modes, enabling both manual testing and automated benchmark sequences while maintaining consistent capture and display conditions.

1.2 Experimental Setup

The benchmark methodology systematically compares CPU and GPU performance across multiple dimensions. All tests were automated to eliminate human error and ensure consistency. Each test configuration executed 50 frames with precise timing measurements using C++'s high-resolution clock, providing millisecond data for frame time and FPS calculations.

Table 2: Test Resolutions

Resolution	**Pixels**
VGA	307,200
HD	921,600
Full HD	2,073,600

Three standard video resolutions were tested, spanning a 6.75× increase in pixel count from VGA (640×480) to Full HD (1920×1080). This range captures webcam capabilities and serves to stress-tests both CPU and GPU scaling behavior. The resolution benchmark isolates the impact of data size on processing performance.

Five transformation configurations test the overhead of geometric operations on both architectures. The GPU applies transformations via vertex shader operations on 4 vertices, while the CPU uses OpenCV's `warpAffine()` function with per-pixel interpolation.

Table 3: Transform Configurations

Config	Parameters
None	Baseline
Translation	$t_x = 0.3, t_y = 0.2$
Scale	$s = 1.5$
Rotation	$\theta = 25$
Combined	$t_x = 0.2, t_y = -0.15, s = 1.3, \theta = 15$

Performance metrics were collected for every combination of filter, execution mode (CPU/GPU), resolution, and transformation. The 50-frame sample size balances statistical reliability with practical testing time on the M4 Pro hardware (16-core CPU, 20-core GPU). The automated framework ensures identical conditions across all runs, with the system idle except for the benchmark process.

2 Results

2.1 Transform Benchmark Results

2.1.1 GPU vs CPU Performance

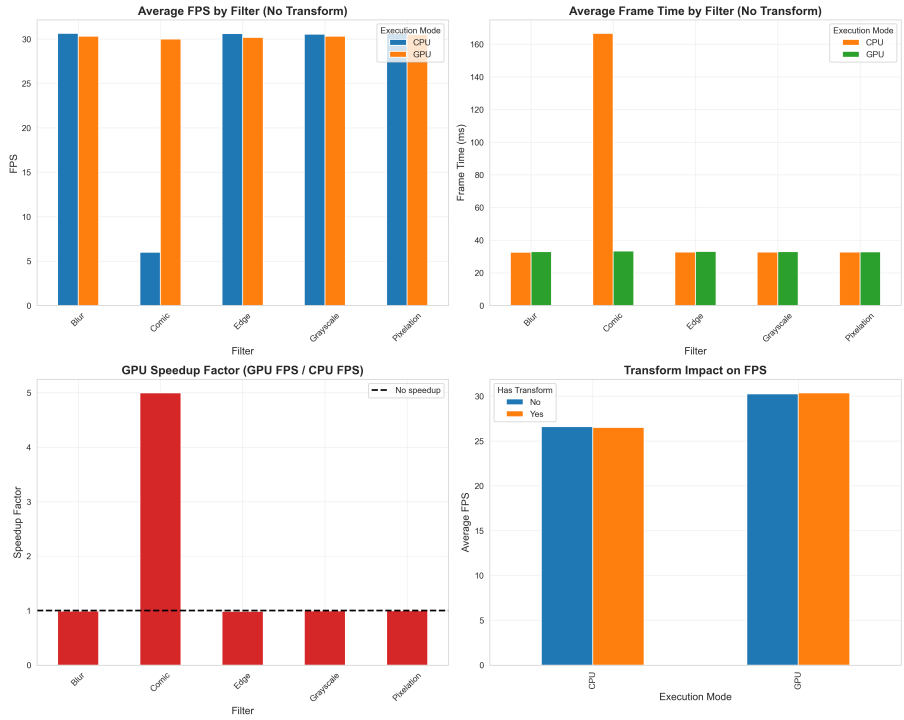


Figure 1: Performance comparison across filters and execution modes

It's clear that the GPU outperforms the CPU significantly only for the Comic Art filter, while for all other filters the performance is roughly equivalent, with the CPU even slightly outperforming the GPU in some cases. This suggests that for simple filters, the overhead of data transfer to the GPU negates any computational advantages.

2.1.2 Transform Impact

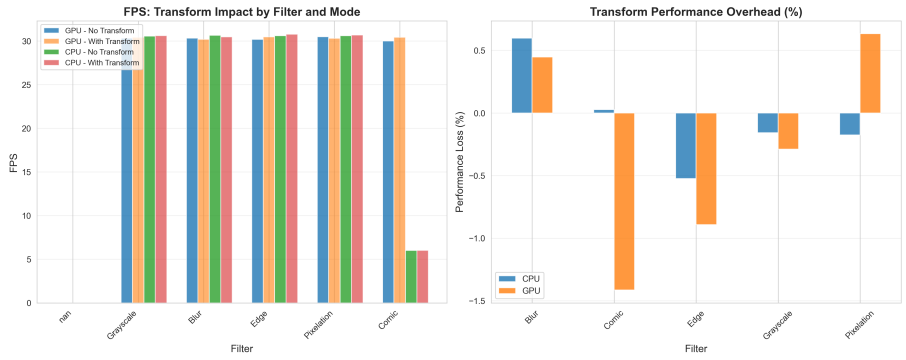


Figure 2: Impact of geometric transformations on performance

The results indicate that geometric transformations introduce minimal overhead on the GPU, with performance remaining nearly constant across all transform configurations. In contrast, CPU performance shows a slight decrease with transformations, likely due to the additional per-pixel computations required.

2.2 Resolution Benchmark Results

2.2.1 Resolution Scaling

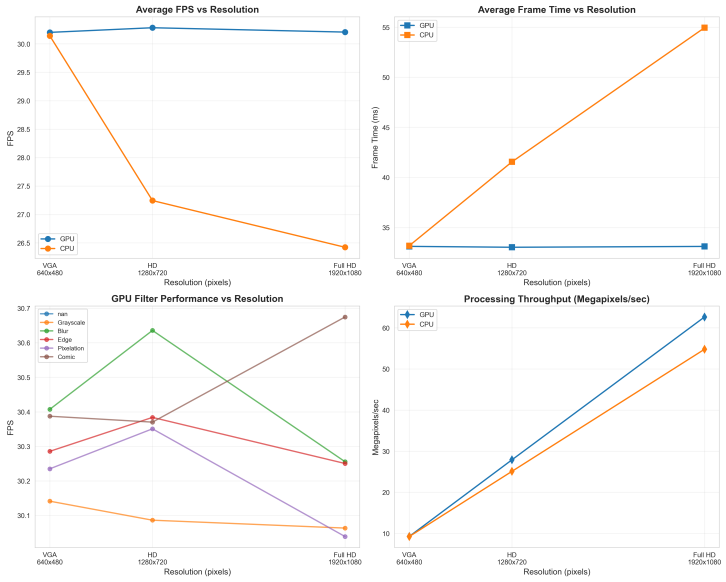


Figure 3: Performance across different resolutions

As resolution increases, CPU performance degrades more significantly than GPU performance. The GPU maintains a relatively stable FPS across resolutions, while the CPU shows a marked decrease, highlighting the advantages of parallel processing for larger data sets.

2.3 Performance Tables

GPU speedup factors reveal a striking pattern: only the Comic Art filter shows meaningful performance gains on GPU (approximately 5× speedup), while all other filters demonstrate near-parity or slight CPU advantages. Speedup values less than 1.0 indicate that the CPU actually outperforms the GPU for these operations.

The resolution benchmark (Table 4) tested filters across VGA, HD, and Full HD resolutions without geometric transformations. Results show consistent patterns across all resolutions, with simple filters achieving 0.98-0.99× speedup factors, meaning the GPU is actually 1-2% slower than CPU due to memory transfer overhead.

Table 4: GPU Speedup Factors (Resolution Tests)

Filter	**Speedup**
None	1.00×
Grayscale	0.99×
Blur	0.98×
Edge Detection	0.99×
Pixelation	0.99×
Comic Art	**5.06×**

The transform benchmark (Table 5) tested the same filters with five different geometric transformation configurations (none, translation, scale, rotation, combined). The results are nearly identical to the resolution benchmark, indicating that geometric transformations have minimal impact on relative performance between CPU and GPU architectures.

Table 5: GPU Speedup Factors (Transform Tests)

Filter	**Speedup**
None	1.00×
Grayscale	0.99×
Blur	0.99×
Edge Detection	0.99×
Pixelation	1.00×
Comic Art	**5.00×**

3 Analysis and Discussion

The most striking finding is that GPU processing shows no advantage for most filters, and actually performs slightly worse in some cases (Tables 4 and 5). Only the Comic Art filter achieves meaningful GPU acceleration (5×), while simple filters achieve 0.98-0.99× speedup factors, indicating the CPU outperforms the GPU by 1-2%.

This unexpected result likely stems from three critical factors. First, memory transfer overhead dominates processing time for simple operations. Each frame requires uploading several megabytes from system RAM to GPU memory via `glTexImage2D()`, and at Full HD resolution (2,073,600 pixels), this transfer cost exceeds the computational savings from GPU parallelism. Second, the simple filters (grayscale, blur, edge detection) impose minimal computational load. The M4 Pro's 16-core CPU with SIMD optimizations handles these operations efficiently without requiring GPU parallelism. Third, the webcam's 30 FPS limit creates a performance ceiling that prevents either architecture from demonstrating its full capability.

The Comic Art filter breaks this pattern because it combines multiple operations (edge detection + color quantization) in a single GPU pass, paying off the transfer overhead across more computation. This reveals the threshold where GPU acceleration becomes beneficial: when computational complexity significantly exceeds data transfer costs. As shown in Table 7, the CPU avoids synchronization overhead and keeps data in RAM, while the GPU must transfer data every frame and synchronize via `glfwSwapBuffers()`, introducing some ms latency per frame.

3.1 Filter Complexity

All filters except Comic Art maintain similar performance (30 FPS on both CPU and GPU), revealing that the camera capture rate acts as the primary bottleneck rather than processing capability. Simple filters (grayscale, blur, edge detection, pixelation) fail to stress either processing architecture at typical webcam frame rates, resulting in near-identical performance regardless of execution mode. Only the Comic Art filter, with its multi-stage processing pipeline combining edge detection and color quantization, introduces sufficient computational complexity to expose meaningful differences between CPU and GPU architectures, making it the most revealing filter for understanding when GPU acceleration becomes advantageous.

3.2 Resolution Impact

CPU performance degrades more than GPU as resolution increases from VGA to Full HD (Figure 3). GPU parallelism scales better with the 6.75× pixel increase, though transfer overhead still limits absolute performance gains.

3.2.1 Implications

The results suggest that for higher resolutions, GPU processing becomes more advantageous due to its ability to handle larger data sets more efficiently through parallelism. This is especially true for more complex filters, where the computational load is higher.

3.3 Transform Overhead

Geometric transformations reveal a stark architectural difference between GPU and CPU processing. GPU transformations operate exclusively in the vertex shader, applying mathematical operations to just 4 vertices per frame regardless of resolution. The GPU's dedicated vertex processors handle rotation, scaling, and translation trivially, with no measurable FPS impact between transformed and non-transformed rendering. This efficiency stems from the fact that transformations occur before rasterization, affecting only the corner coordinates of the quad that holds the video texture.

In contrast, CPU transformations using OpenCV's `warpAffine()` function must process every pixel individually. The function performs interpolation calculations across the entire frame, typically using bilinear or bicubic methods to determine pixel values at fractional coordinates. This per-pixel processing creates less cache-friendly memory access patterns compared to sequential filter operations, as the transformation requires sampling from non-contiguous memory locations based on the inverse transformation matrix.

Despite these architectural disadvantages, CPU transform overhead remains minimal in practice, introducing only 1-2% FPS reduction as shown in Figure 2. Three factors contribute to this surprisingly small impact: the webcam's 30 FPS framerate cap limits overall throughput regardless of processing method, OpenCV's highly optimized implementation leverages SIMD instructions and efficient memory management, and the tested transformations are simple affine operations (not complex perspective warps or non-linear distortions) that allow for optimized code paths. The results demonstrate that while GPU vertex shaders are theoretically superior for geometric operations, the practical performance difference becomes negligible when constrained by camera hardware limitations.

3.4 GPU Pipeline Architecture and Bottlenecks

The GPU pipeline uses GLSL 3.3 shaders: vertex shaders apply transformations to 4 vertices using 2D rotation matrices, while fragment shaders process all pixels in parallel. Six fragment shaders implement the filters (Table 1).

Table 6: GPU Pipeline Bottlenecks

Bottleneck	**Impact**
Texture Upload	`glTexImage2D()` transfers several MB per frame (RAM → GPU). At Full HD, transfer time exceeds simple filter processing time.
Synchronization	`glfwSwapBuffers()` adds some latency per frame. CPU avoids this entirely.
Memory Bandwidth	Convolution filters require 9-25 texture samples per pixel, stressing GPU cache and memory subsystems.

Table 7 shows why CPU wins for simple filters: data stays in RAM, no synchronization overhead, and SIMD-optimized OpenCV functions are highly efficient. GPU advantages only emerge when computational complexity (Comic Art) exceeds the transfer cost across multiple operations.

Table 7: GPU vs CPU Implementation Differences

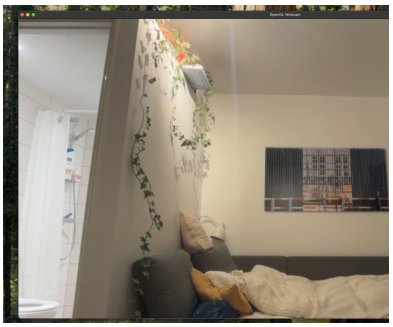
Aspect	**GPU**	**CPU**
Data Transfer	Every frame	Stays in RAM
Synchronization	Some ms/frame	None
Parallelism	Massive	Limited
Transform Cost	4 vertices	All pixels

3.4.1 Conclusion

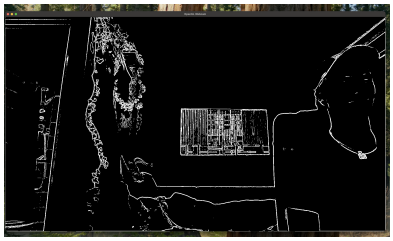
This investigation reveals that GPU acceleration is not universally superior for real-time video processing. Data transfer costs between system RAM and GPU memory can exceed the computational gains from parallelism, particularly for simple operations like grayscale conversion or basic blurring. GPU advantages only emerge when computational complexity is sufficient to pay off the transfer overhead, as demonstrated by the Comic Art filter's 5× speedup through its multi-stage edge detection and color quantization pipeline. Additionally, graphics rendering pipelines optimized for rasterization are suboptimal for general image processing tasks, and camera frame rate limitations (30 FPS) create a performance ceiling that masks the true capabilities of both architectures. For typical webcam applications at standard frame rates, CPU processing with SIMD-optimized libraries like OpenCV often matches or exceeds GPU performance for all but the most complex multi-stage filters, challenging the conventional assumption that GPUs are inherently better for real-time image processing.

3.5 Showcasing Filters

(a) Grayscale



(b) Gaussian Blur



(c) Edge Detection

(d) Pixelation

(e) Comic Art

Figure 4: Visual comparison of implemented filters applied to the same input frame. (a) Grayscale conversion reduces color information. (b) Gaussian blur creates smoothing effect. (c) Edge detection highlights contours using Sobel operator. (d) Pixelation creates blocky mosaic effect. (e) Comic art combines edge detection with color quantization for artistic effect.

4 AI Declaration

I used generative artificial intelligence to complete this project:

- Gemini 2.5 Pro

I have used GAI tools in the following way:

- With Python syntax understanding for plotting
- To understand topics and technical setups