Procesamiento de video en algoritmos paralelos, transformando un video a escala de grises

Autores

Kevin Nicolai García Rodríguez
Juan José Márquez Villarreal

Fecha

Noviembre 2025

1. Objetivo

El objetivo de este laboratorio es implementar y comparar la eficiencia de un algoritmo secuencial y uno paralelo multihilos al procesar un video, transformando el video original a uno en escala de grises. El resultado esperado es un informe que demuestre el *speedup* (aceleración) logrado mediante el paralelismo.

2. Marco Teórico

El procesamiento de video consiste en aplicar diferentes operaciones sobre una secuencia de imágenes (frames) para transformarlas, analizarlas o mejorar su calidad. A diferencia de trabajar con una sola imagen, procesar video implica manejar muchos fotogramas por segundo, lo cual aumenta bastante la carga de trabajo.

Aquí es donde la paralelización se vuelve muy útil, ya que cada frame puede procesarse por separado sin depender de los demás. Esto permite repartir el trabajo entre varios núcleos del procesador y acelerar el tiempo total. Algunas de las aplicaciones más comunes son la conversión de color, la detección de bordes, el análisis de movimiento y la extracción de características.

3. Metodología

3.1. Configuración del Hardware

Las pruebas se realizaron en un computador portátil con las siguientes características:

- **Equipo**: Dell Latitude 5510
- **CPU**: Intel Core i5-10210U (4 núcleos, 8 hilos, hasta 4.20 GHz)
- **GPU**: Intel UHD Graphics integrada (1.10 GHz)
- **Memoria RAM**: 16 GB
- **Sistema Operativo**: Ubuntu 24.04.3 LTS (64 bits)
- **Kernel**: Linux 6.14.0-36-generic

3.2. Configuración del Software

Para el desarrollo y las pruebas se usaron las siguientes herramientas:

- **Python 3.12.11**
- **NumPy 2.2.6**
- **OpenCV 4.12.0.88**
- Librerías estándar: `multiprocessing`, `time`, `os`

3.3. Descripción del Algoritmo

El proceso para convertir un video a escala de grises se dividió en varias etapas consecutivas, desde la preparación del entorno hasta la creación del video final. A continuación, se describe cada una de ellas de forma general:

1. **Limpieza y preparación del ambiente.** Se eliminan las carpetas utilizadas en ejecuciones anteriores (`video`, `frames_video_original`, `frames_video_result`) y luego se vuelven a crear para tener un entorno limpio antes de procesar el video.

2. **Carga del video de entrada.** El video original (`cat_video.mp4`) se copia a la carpeta `video/`. Con OpenCV se obtiene:

   - el total de frames,
   - los FPS,
   - y la duración aproximada.

3. **Extracción de frames.** El video se recorre frame por frame. Para cada uno:

   - Se lee el frame en formato BGR.
   - Se convierte a RGB.
   - Se guarda como imagen JPEG en `frames_video_original/`.

   Esto genera una secuencia ordenada de imágenes que representan el video original.

4. **Procesamiento secuencial (OpenCV).** Cada imagen se procesa en orden:

   - Se carga desde disco.
   - Se convierte a escala de grises con `cv2.cvtColor`.
   - Se reacomoda en formato BGR.
   - Se guarda en `frames_video_result/`.

   Todo esto ocurre en un único hilo de ejecución.

5. **Procesamiento paralelo (multiprocessing).** Aquí se usa `multiprocessing.Pool` para repartir el trabajo entre todos los núcleos del procesador. Cada proceso ejecuta la función `process_single_frame(i)`, que:

   - lee el frame i,
   - lo convierte a escala de grises,
   - y lo guarda en la carpeta de salida.

   Como cada frame es independiente, esta técnica permite reducir bastante el tiempo total.

6. **Reconstrucción del video final.** Una vez procesados todos los frames, se crea un nuevo video:
    - se determina la resolución usando el primer frame procesado,
    - se configura `cv2.VideoWriter`,
    - y se agregan los frames finales en orden.

Este flujo completo permite comparar con precisión el tiempo que tardan ambas implementaciones y generar un video final que muestra el resultado.

4. Resultados

Durante las pruebas se procesaron **880 frames** utilizando las dos estrategias: el método secuencial y el método paralelo con múltiples procesos. Los tiempos obtenidos fueron:

- **Tiempo secuencial:** 4.5840 s
- **Tiempo paralelo (8 núcleos):** 1.2080 s
- **Speedup obtenido:** $\approx 3{,}7\times$

Los resultados muestran una mejora importante al usar paralelismo. Las gráficas que comparan tiempos y speedup se incluyen a continuación:

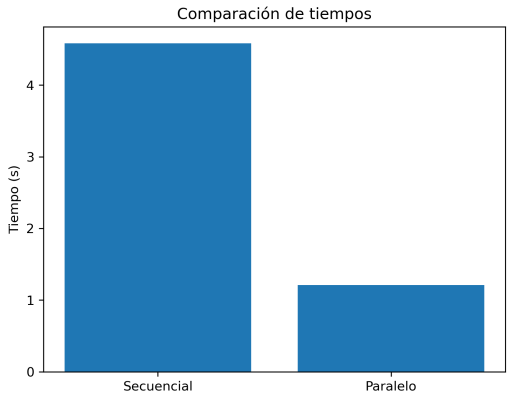

Figura 1: Comparación de los tiempos de ejecución entre los métodos secuencial y paralelo.

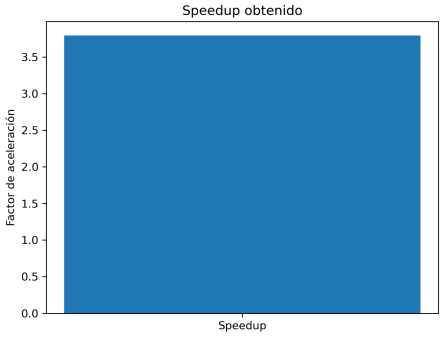

Figura 2: Speedup obtenido mediante procesamiento paralelo.

5. Análisis de Rendimiento

El procesamiento paralelo tomó solo 1.2080 segundos, mientras que el método secuencial tardó 4.5840 segundos. Esto representa un **speedup cercano a 3.7x**, lo cual deja claro que los **8 núcleos** del equipo se aprovecharon bastante bien.

Este buen rendimiento se explica porque:

- Cada frame puede procesarse por separado.
- OpenCV ya trae funciones optimizadas en C/C++, que reducen gran parte del trabajo pesado.
- El paralelismo permite repartir la carga y disminuir tiempos de espera.

De hecho, si todo este proceso se hubiera hecho con **Python puro**, sin OpenCV, el tiempo secuencial sería de aproximadamente **40–50 minutos**, mientras que el paralelo estaría cerca de **15 minutos**. Esto muestra lo mucho que ayuda usar librerías optimizadas antes incluso de aplicar paralelismo.

En general, los resultados dejan claro que el procesamiento paralelo no solo acelera el trabajo, sino que escala bien para tareas intensivas por píxel.

6. Conclusiones

Con este experimento fue posible comprobar el impacto positivo que tiene el paralelismo en el procesamiento de video. Repartir la carga entre varios procesos reduce notablemente el tiempo total sin afectar el resultado final.

El *speedup* obtenido demuestra que tareas como convertir frames a escala de grises son fáciles de paralelizar porque no dependen unas de otras. Además, el uso de OpenCV aporta una mejora adicional muy importante en comparación con lo que sería hacerlo solo con Python.