Informe - Trabajo Práctico

Programación Concurrente

Universidad Nacional de Quilmes

Grupo 09 - 2S2025

Autoría:

Confalonieri, Melisa
melisaconfalonieri@gmail.com

Raimondi, Gastón
gaston14015@gmail.com

1. Introducción

1.1. Dominio

El presente trabajo aborda el procesamiento de imágenes mediante la aplicación de filtros de convolución. El sistema permite trabajar con imágenes en formato RGB (color) o en escala de grises. Cada filtro se define mediante una matriz de valores, denominada *kernel*, que combina el valor de un píxel con el de los píxeles lindantes para generar un nuevo valor transformado en la misma posición.

El procesamiento se organiza en un pipeline de tres filtros sucesivos, donde la salida de cada filtro se utiliza como entrada del siguiente. Esta estructura permite realizar transformaciones progresivas sobre la imagen, aplicando diferentes efectos o realces según los *kernels* seleccionados.

1.2. Arquitectura del sistema

El sistema está organizado siguiendo el patrón Productor-Consumidor, donde un pool de threads trabajadores procesa tareas extraídas de un buffer compartido. La arquitectura puede describirse en cuatro niveles de abstracción:

I. Coordinación general
- **Main**: Dirige la ejecución completa del programa, creando el pipeline de imágenes y encolando las tareas correspondientes en el buffer.
- **Config**: Gestiona los parámetros de ejecución y delega a **FilterLoader** la carga de filtros desde archivos.
- **ThreadPool**: Administra la creación e inicialización de los threads trabajadores.

II. Procesamiento
- **FilterWorker**: Thread consumidor que extrae y ejecuta tareas del buffer.
- **ConvolutionTask**: Tarea que aplica un filtro sobre una región específica de la imagen, incluyendo los cálculos de píxeles basados en el *kernel* del filtro.
- **PoisonPill**: Señal para indicar a los workers que no quedan tareas y deben finalizar.

III. Datos
- **Filter**: Representa un *kernel* de convolución.
- **ImageData**: Encapsula una imagen completa y permite crear copias recortadas para cada etapa del pipeline de filtros.

- **ImagePartitioner**: Divide una imagen en regiones según la cantidad de threads disponibles.
- **ImageRegion**: Representa una región rectangular de una imagen.

IV. Sincronización
- **Buffer**: Cola bloqueante que almacena las tareas pendientes.
- **WorkerCounter**: Registra la cantidad de workers activos y permite esperar a que todos terminen.
- **FilterMode**: Interfaz que define la estrategia de sincronización entre filtros.
- **SequentialMode**: Impone que cada filtro complete toda la imagen antes de iniciar con el siguiente.
- **ConcurrentMode**: Permite que regiones de un filtro comiencen el siguiente apenas sus dependencias estén listas.

1.3. Particionamiento de la imagen

La imagen se divide en regiones horizontales para distribuir la carga de trabajo entre los threads. Si la imagen tiene altura H y se utilizan N workers, cada región contiene aproximadamente H/N filas; en caso de que H no sea divisible por N, las filas restantes se asignan a la última región para asegurar que todos los píxeles sean procesados.

La partición se realiza sobre la imagen de salida de cada filtro, ya que es esa la estructura donde los workers escriben los resultados. Esto garantiza que cada región tenga un rango de filas válido en el espacio de coordenadas de la imagen destino. Durante el procesamiento, cuando una tarea necesita leer píxeles de la imagen fuente, aplica un desplazamiento equivalente al radio del filtro para compensar el recorte.

1.4. Reducción por filtros

Cada filtro posee un radio r que determina cuánto se reduce la imagen al aplicarlo. Durante la convolución, los píxeles ubicados cerca de los bordes no pueden procesarse porque no cuentan con suficiente información alrededor, por lo que se descartan. El resultado es una reducción fija de $2r$ píxeles tanto en ancho como en alto. La misma es acumulativa y define el tamaño de salida utilizada por cada etapa del pipeline.

1.5. Pipeline de imagenes

Para evitar condiciones de carrera entre lectura y escritura, se utiliza un pipeline compuesto por cuatro imagenes:

imagen[0]: Imagen original (entrada del filtro 1)
imagen[1]: Salida del filtro 1 (entrada del filtro 2)
imagen[2]: Salida del filtro 2 (entrada del filtro 3)
imagen[3]: Salida del filtro 3 (resultado final)

Las tareas correspondientes a cada filtro reciben referencias fijas a su par entrada/salida al momento de ser creadas; por ejemplo todas las tareas del filtro 1 leen siempre de imagen[1] y escriben en imagen[2], independientemente del orden o momento en que se ejecuten.

1.6. Sincronización y disponibilidad de datos

Para que el procesamiento mantenga el orden lógico de los filtros y cada etapa trabaje siempre con datos válidos, el sistema ofrece dos estrategias de ejecución: secuencial y concurrente.

En el **modo secuencial**, la coordinación se regula por filtro. Un monitor implementa la barrera que bloquea el avance hasta que todas las tareas finalizan el filtro en curso. Solo entonces se habilitan las tareas del filtro siguiente, garantizando que nunca existan operaciones concurrentes sobre etapas distintas.

En el **modo concurrente**, la coordinación se regula por región. Para aplicar el filtro *F* sobre la región *R*, deben haberse completado previamente las regiones [*R-1, R, R+1]* del filtro *F-1*, dado que cada etapa depende de los valores producidos por esos mismos segmentos en la etapa anterior. Esta sincronización se implementa mediante un monitor que registra el estado de cada región y bloquea la ejecución cuando aún no es seguro avanzar. Las regiones sin vecinos –la primera y la última– solo esperan a su contigua, mientras que las tareas del filtro inicial pueden comenzar sin bloqueo.

2. Evaluación

2.1. Entorno de pruebas

Las pruebas se realizaron en un equipo con las siguientes características:

- **Procesador**: AMD Ryzen 7 5700G @ 3.80–4.40GHz, 8-Core / 16-Threads
- **Memoria RAM**: 32GB
- **Sistema operativo**: Windows 10 Pro, 64 bits
- **Java**: OpenJDK 21

2.2. Preparación de la evaluación

Se detallan a continuación las condiciones y parámetros bajo los cuales se realizaron las pruebas para evaluar el comportamiento del sistema en diferentes escalas.

I. Imagenes de prueba
   - Imagen pequeña: 512x512 píxeles
   - Imagen grande: 2048x2048 píxeles

II. Parámetros fijos
   - Tamaño del buffer: 32
   - Tamaño del filtro: 3x3

III. Variables evaluadas
   - Modos de ejecución: Secuencial y Concurrente
   - Cantidad de threads: 1, 4, 8, 16

IV. Metodología de medición

Para cada combinación de variables evaluadas se ejecutaron 13 iteraciones del programa. Las primeras 3 ejecuciones se descartaron para que el sistema alcanzara un estado estable. Los tiempos reportados corresponden al promedio de las 10 ejecuciones restantes.

2.3. Resultados

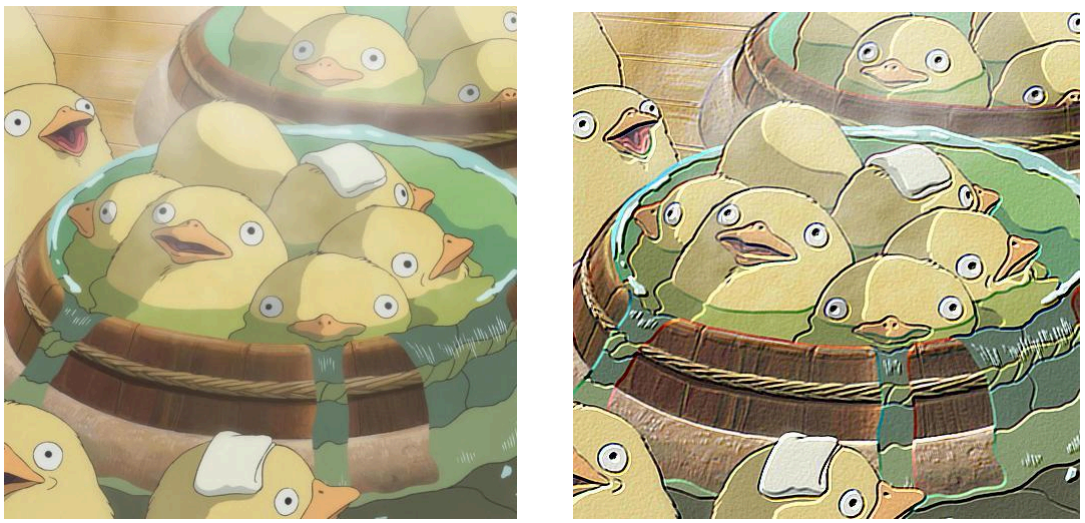
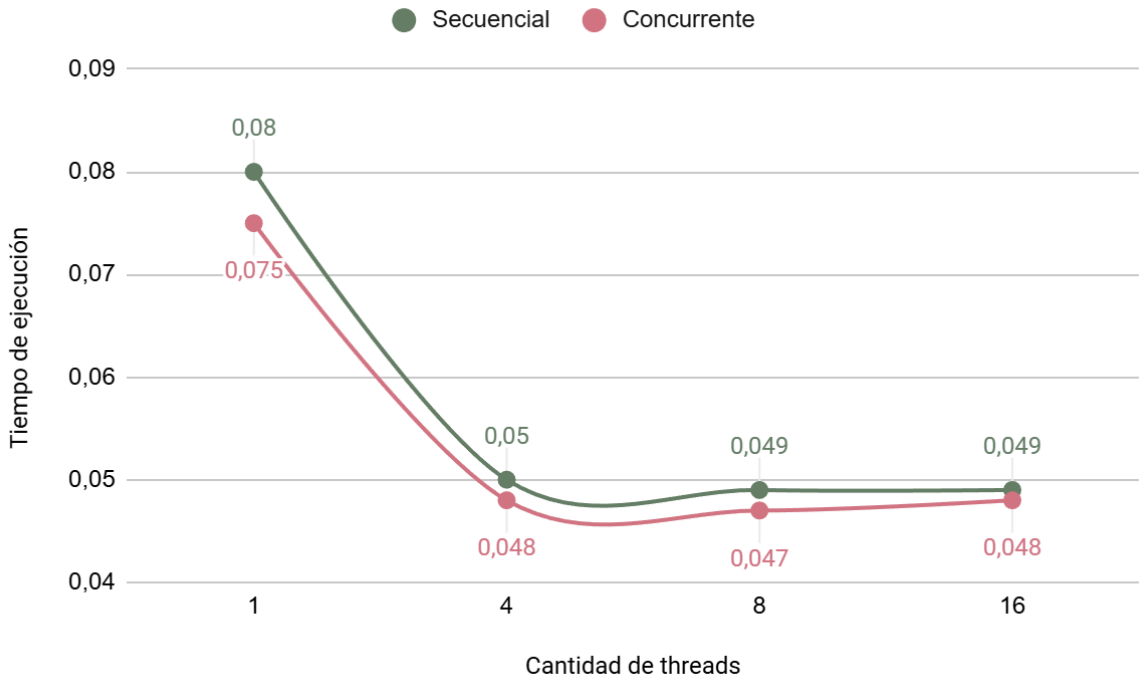


Imagen original y resultado tras aplicar los tres filtros

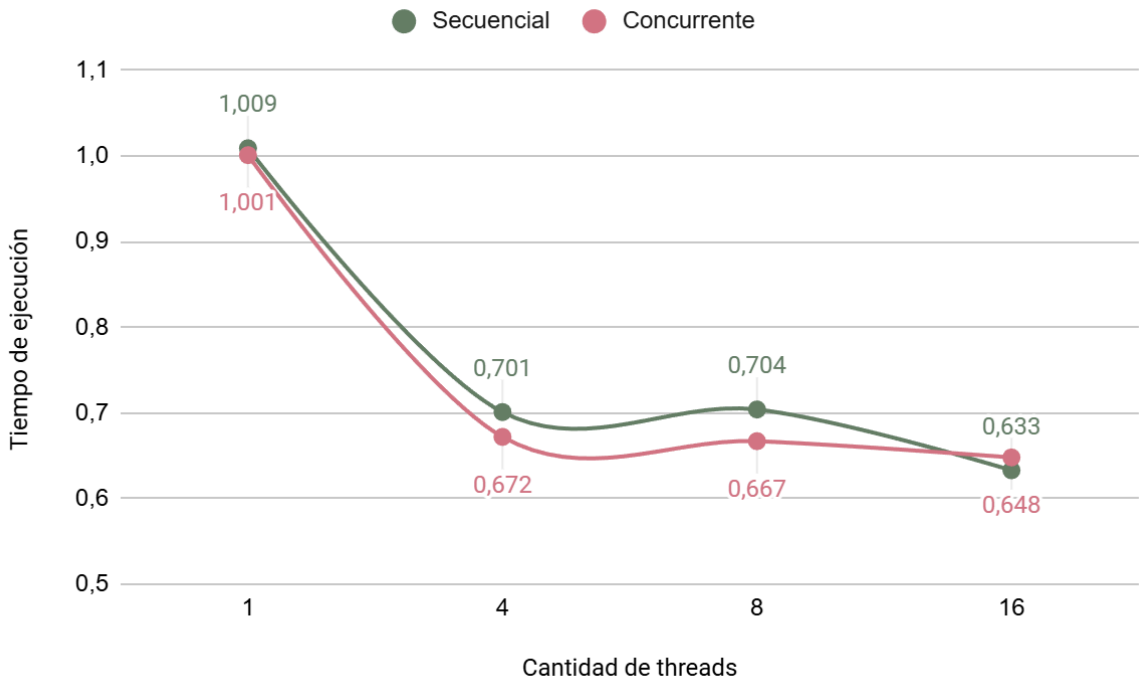
I. Imagen 512x512 – Tiempos de ejecución (ms)

Threads	Secuencial	Concurrente
1	0,08	0,075
4	0,05	0,048
8	0,049	0,047
16	0,049	0,048



II. Imagen 2048x2048 – Tiempos de ejecución (ms)

Threads	Secuencial	Concurrente
1	1,009	1,001
4	0,701	0,672
8	0,704	0,667
16	0,633	0,648



3. Análisis

3.1. Tiempos extremos observados

- **Imagen 512x512**
  - Tiempo máximo: 0,080ms (modo secuencial, 1 thread)
  - Tiempo mínimo: 0,047ms (modo concurrente, 8 threads)

- **Imagen 2048x2048**
  - Tiempo máximo: 1,009ms (modo secuencial, 1 thread)
  - Tiempo mínimo: 0,633ms (modo secuencial, 16 threads)

3.2. Rendimiento y escalabilidad

Los resultados muestran un patrón consistente en ambos tamaños de imagen: la mejora más significativa ocurre al incrementar la cantidad de threads de 1 a 4, con reducciones marcadas en los tiempos de ejecución. Esto evidencia que el paralelismo inicial aprovecha de manera efectiva la capacidad de los threads para procesar regiones de la imagen simultáneamente.

A partir de ese punto, las mejoras se amortiguan y el rendimiento alcanza un punto de saturación; en algunos casos se registra una ligera degradación, como ocurre al pasar de 8 a 16 threads en la imagen pequeña en modo concurrente, o de 4 a 8 threads en la imagen grande en modo secuencial. Esto indica que la gestión de hilos y la coordinación necesaria para el procesamiento simultáneo pueden superar los beneficios de agregar más threads cuando cada uno tiene poca carga de trabajo.

Con 8–16 regiones y 8–16 threads, cada hilo procesa apenas una o dos regiones por filtro, lo que limita las oportunidades de solapamiento de tareas para el modo concurrente y genera implícitamente una barrera similar a la ejecución secuencial. Una granularidad más fina permitiría que los hilos avancen a filtros siguientes mientras otros completan el actual, aunque también incrementaría el overhead de sincronización.

En conjunto, la ejecución concurrente se muestra más eficiente que la secuencial, especialmente para imagenes grandes; sin embargo, el rendimiento no escala de manera lineal con el número de threads. La similitud en algunos casos entre ambos modos sugiere que, para filtros livianos (3x3) y pipelines cortos (3 filtros), el costo de sincronización requerido por el modo concurrente puede compensar o incluso superar los beneficios del solapamiento de tareas.