Programación Concurrente

Trabajo Práctico

Autoría

Bolaños, Rodrigo

Email: rodrimanuelbw@gmail.com

Legajo: 45573612

Perin, Alejo

Email: alejoperin11@gmail.com

Legajo: 45065719

Moreira Toja, Ignacio

Email: ignaciomoreiratoja@outlook.com.ar

Legajo: 35155148

Introducción

El trabajo práctico se basa en la implementación de un programa Java con el fin de tomar un directorio de imágenes en formato .jpg y aplicarle un filtro convolucional configurable de 3x3, para así obtener a cada imagen transformada resaltando los bordes de la misma.
El filtro elegido para este trabajo práctico es el sobel horizontal, el cual aísla los bordes horizontales de la imagen usando el siguiente kernel:

$$k_{\text{sobel horizontal}} = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}$$

La aplicación de los filtros se hará de forma concurrente para lo que se usarán distintos threads (que llamaremos FilterWorkers) y que cada uno recibirá una imagen de un buffer creado por el ThreadPool, el manager de threads.
Dicho ThreadPool se encarga de crear a los FilterWorkers, de inicializar el buffer y de cargar las tareas en el buffer. Una vez cargadas todas las tareas, debe asegurarse que los procesos terminen, para esto utiliza PoisonPills, que al ejecutarlas lanzan una excepción y detienen al FilterWorker que tomó dicha tarea.
Para saber cuándo terminar la ejecución se implementa WorkersCounter que recibe por parámetro la cantidad de FilterWorkers a ser creado por el ThreadPool, y luego recibe la advertencia de cada FilterWorker que se va a detener por haber agarrado una PoisonPill para así contabilizar los threads activos y poder terminar el programa.
Para esto, se utiliza un notify al llegar a 0 FilterWorkers activos y un wait en el método de finalizar para esperar a que los threads ejecutándose sean 0.
El FilterWorker por su parte, toma las tareas del buffer y las ejecuta. Como fue mencionado anteriormente, al tomar una tarea PoisonPill, éste lanza un mensaje para el WorkersCounter y luego detiene su ejecución.
Las tareas son de la clase FilterTask que al ser ejecutadas aplican el filtro a la imagen. Para tal consigna, contiene el kernel elegido por el grupo, y luego mediante el uso de las clases ImageIO, BufferedImage y WritableRaster manipula las imágenes pixel a pixel para lograr aplicar el filtro.
Al ejecutar el método run(), lee una imagen desde un archivo, aplica el filtro que realza los bordes calculando una combinación de los valores de los píxeles vecinos, y luego guarda la imagen resultante en una carpeta de salida. El filtro se implementa usando la matriz (kernel) de 3x3 y el resultado ajusta los valores de los píxeles para asegurarse de que estén dentro de un rango específico, lo que permite resaltar los bordes de la imagen original.

Evaluación

Para la evaluación del programa creado para este trabajo práctico, se realizaron una serie de ejecuciones con diferentes combinaciones de imágenes, threads y tamaño de buffer.
Se realizaron diez ejecuciones de cada combinación de los ítems antes descriptos, para así conseguir un dato más fiel, al sacar el promedio, que si sólo se corriera una vez.

La computadora que se usó para dicha tarea utiliza un procesador Ryzen 5 5660g de 6 núcleos y 3,89 GHz de velocidad, 16 Gb de memoria RAM, corriendo Windows 10 x64.

A continuación se detallan las tablas resultado de los promedios de las ejecuciones:

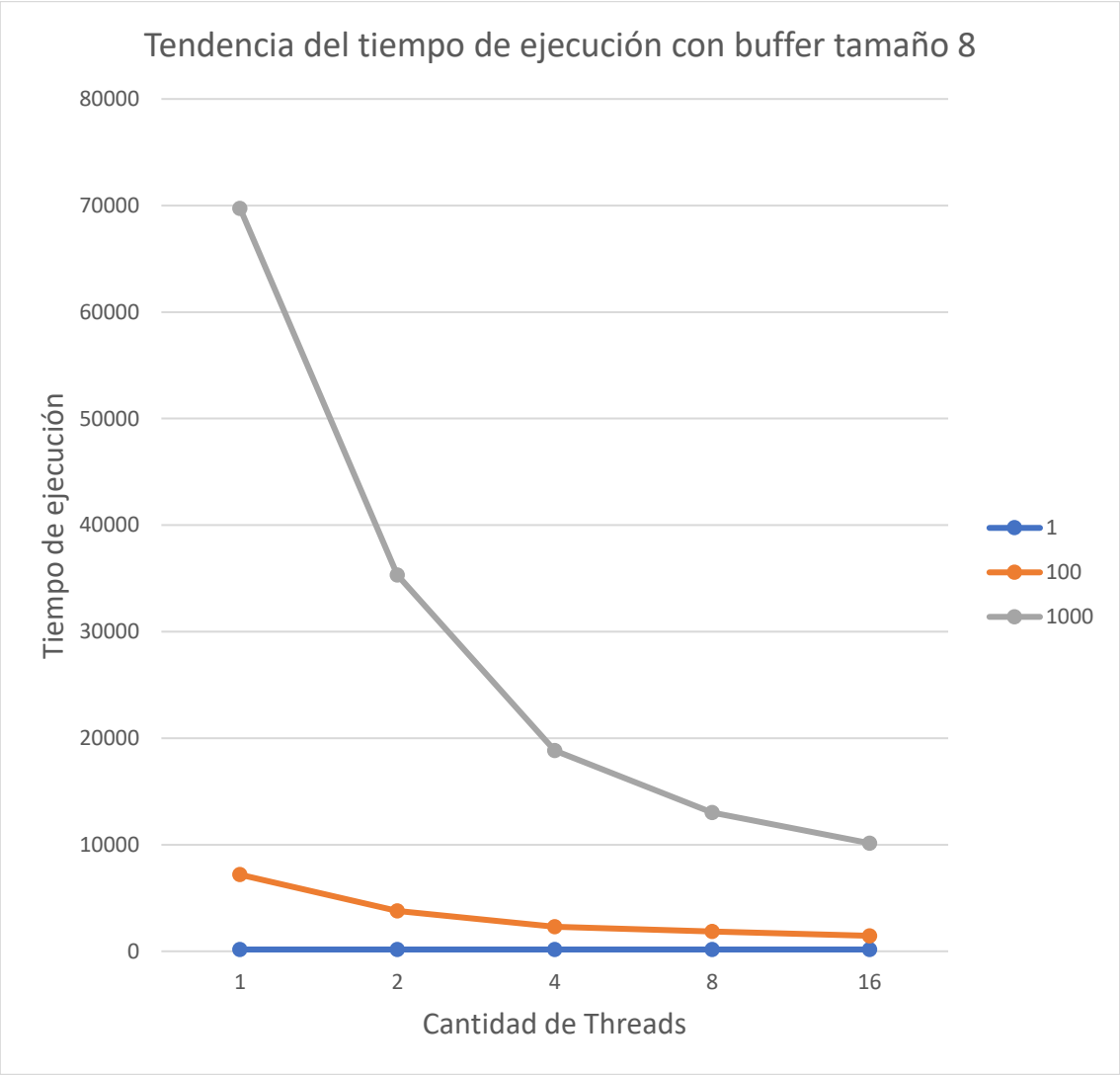
Imágenes:	1				
	Threads				
Buffer	1	2	4	8	16
2	171	166	170	168	168
8	168	167	167	168	167
32	170	168	166	169	169

Imágenes:	100				
	Threads				
Buffer	1	2	4	8	16
2	7292	3851	2308	1774	1460
8	7190	3793	2292	1859	1455
32	7264	3796	2315	1851	1431

Imágenes:	1000				
	Threads				
Buffer	1	2	4	8	16
2	69767	35677	18845	13187	10200
8	69739	35330	18829	13035	10149
32	69927	35534	18579	13007	10203

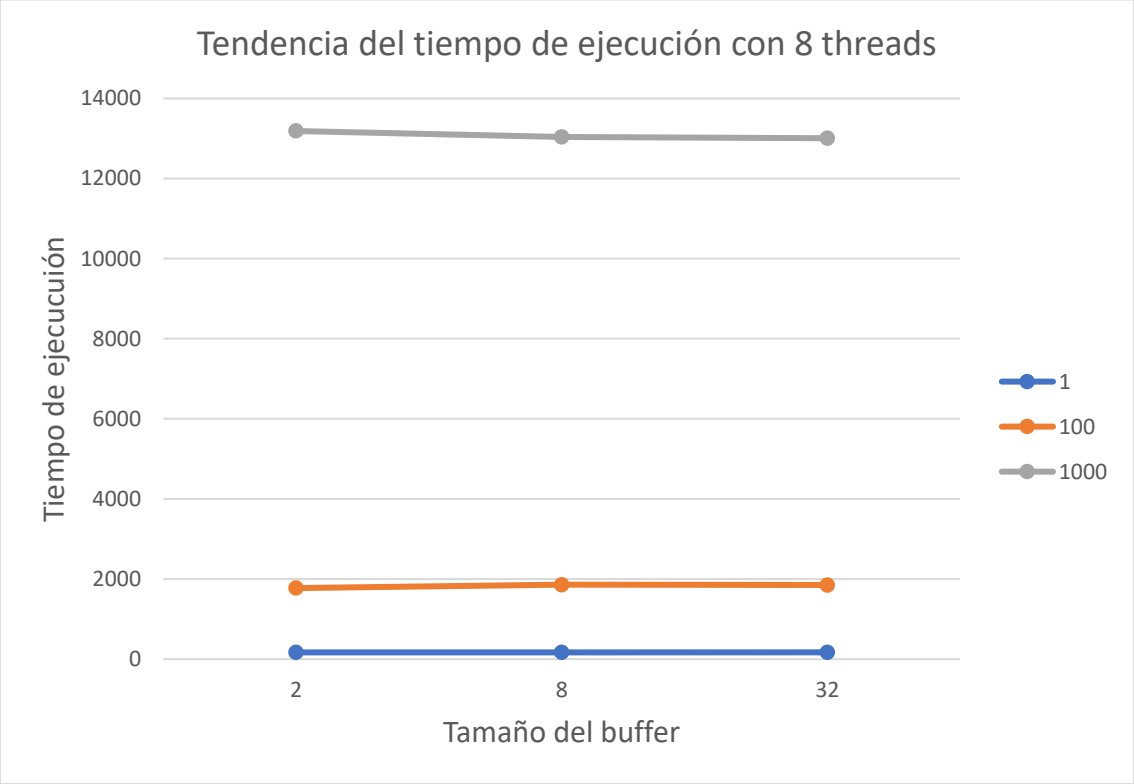
Para un buffer de tamaño 8 se consigue la siguiente tabla y su respectivo gráfico:

Buffer	8				
	Threads				
Imágenes	1	2	4	8	16
1	168	167	167	168	167
100	7190	3793	2292	1859	1455
1000	69739	35330	18829	13035	10149



Y para 8 threads la consiguiente información:

Threads	8		
	Buffer		
Imágenes	2	8	32
1	168	168	169
100	1774	1859	1851
1000	13187	13035	13007



Análisis

Según los datos obtenidos, podemos destacar la importancia del rol que toma la cantidad de threads como causal del aumento o disminución del tiempo de ejecución. Así mismo, se observa que la cantidad de threads es importante siempre y cuando no se aumente la misma luego de sobrepasar la cantidad de tareas a disponer. Como ejemplo, en las ejecuciones del programa utilizando una sola imagen, las diferencias en los tiempos de ejecución son despreciables, quizás atribuibles a ruido generado por otros procesos de la computadora. Esto se debe a que el análisis de cada imagen la realiza un FilterWorker, por lo que todos los FilterWorkers (o threads) que se instanciaron extra, no realizaron ninguna tarea y rápidamente terminaron su ejecución mediante la PoisonPill.

En los casos en los que se aprecia una mejoría sustancial en los tiempos de ejecución es en las ejecuciones de varias imágenes, pudiendo ser evidenciado en las tablas de las ejecuciones para 100 y 1000 imágenes.

Por su parte, el tamaño del buffer no genera un cambio drástico en los tiempos de ejecución, pudiendo teorizarse que el constante consumo de dicho buffer es repuesto rápidamente por el mismo con más tareas o las respectivas PoisonPill.

Por último, si evaluamos la última de las variables, la cantidad de imágenes, tiene claramente inferencia en el tiempo de ejecución, dilatando los ciclos de tareas que cada FilterWorker realiza. Por ello, mientras más imágenes se tengan y menos threads haya disponibles tomará mayor tiempo de ejecución, ya que cada thread deberá ejecutar más ciclos para poder procesar todas las tareas.

Como expresado con anterioridad, la cantidad óptima de threads y tamaño de buffer sólo son relevantes mientras los threads no superen a la cantidad de imágenes a ser procesada. En cualquier otro caso, siempre será mejor contar con la mayor cantidad de threads y un tamaño de buffer del doble de los threads.

Con los valores relevados no se puede establecer una relación entre los valores óptimos de cantidad de threads y tamaño de buffer con el hardware utilizado; ya que, con excepción de las ejecuciones con una imagen que son un caso particular por lo explicado anteriormente, los valores óptimos siempre tendieron a los mismos parámetros.

Para concluir, se debe destacar que como se viene comentando las ejecuciones menos óptimas se obtienen con un solo thread y con diferentes tamaños de buffer; mientras que las ejecuciones más óptimas se obtienen con 16 threads y tamaño de buffer entre 32 y 8.