Informe del Trabajo Práctico de Programación Concurrente

Segundo Cuatrimestre 2022

Carro, Nahuel Agustín
Martinez, Lucas Javier
Nicolini, Lautaro Nehuén

14/11/2022
UNQ

AUTORÍA

Carro, Nahuel Agustín - nahuelagustincarro@gmail.com - Legajo 59072
Martinez, Lucas Javier - lucasluquitas477@gmail.com - Legajo 59776
Nicolini, Lautaro Nehuén - lautaronnicolini@gmail.com - Legajo 33508

INTRODUCCIÓN

La consigna del trabajo pedía que desarrollaramos un programa que pudiera aplicar filtros de convolución a una imagen, para ello, se debía dividir la misma (a la que representamos como una matriz de n*m píxeles) y crear tareas que varios procesos concurrentes debían resolver. Esta tarea sería aplicar una transformación en el valor de la matriz de entrada (consiste en tomar los vértices vecinos a cada celda, multiplicarlos uno a uno por el valor en la celda correspondiente a la matriz de 3 x 3 que se usa de filtro, y sumarlos) y asignarlo como valor en las mismas coordenadas de una matriz salida, posteriormente, esa matriz se convierte en una imagen de salida.

CLASES QUE COMPONEN EL PROGRAMA

La solución que desarrollamos cuenta con la siguiente estructura de clases:

Buffer: clase encargada de recibir y dispensar tareas (objetos que implementan la interfaz Runnable de Java), su constructor recibe por parámetro un número entero que sirve para indicarle la capacidad de almacenamiento de tareas.

Config: clase encargada de recibir configuraciones de entrada para la ejecución del programa, configuraciones disponibles: nombre del archivo de entrada, nombre del archivo de salida (en ambos casos, nombre + extensión, ejemplo entrada.jpg, salida.jpg), tamaño del buffer (rango posible entre 1 y 64), la cantidad de threads que ejecuten tareas (rango posible entre 1 y 16) y nombre del filtro a aplicar (valores posibles: “blur”, “sharpen”, “sobelvertical”, “sobelhorizontal” y “bordes”).

FilterWorker: clase encargada de ejecutar las tareas, su constructor recibe por parámetro un objeto de clase Buffer del que obtendrá las tareas.

PoisonPill: clase que implementa la interfaz Runnable de Java, su constructor recibe por parámetro un objeto de clase WorkerCounter. Cuando un thread le envía el mensaje run a este objeto, le pide al contador de trabajadores que tomó por parámetro que reste un trabajador y lanza una excepción en tiempo de ejecución, se usa como mecanismo de corte para los threads que buscan tareas del buffer.

Task: clase encargada de realizar el cálculo del nuevo valor de cada pixel de la imagen y

su asignación en la matriz de salida, su constructor toma por parámetro un ArrayList<double[][]> (una colección de N matrices de números de 3 de alto por 3 ancho, donde N es la cantidad de canales de la imagen de entrada, cada matriz contiene los valores de los píxeles necesarios para realizar la transformación), unas coordenadas “i” y “j” (la posición del píxel que se está transformando), dos objetos de tipo WritableRaster (la matriz de entrada y la de salida) y el nombre del filtro a aplicar. Contiene también las matrices que representan a cada uno de los filtros. Por cada matriz, la multiplica por su matriz correspondiente en el filtro, acumulando el valor de la suma de cada pixel y luego, lo asigna en la matriz de salida en la coordenada i,j.

ThreadPool: clase encargada de instanciar N FilterWorkers, donde N es el número que solicite el usuario, también recibe las tareas de filtrado y las agrega al buffer junto con N objetos de clase PoisonPill.

Timer: clase que se encarga de tomar el tiempo de ejecución del programa.

WorkerCounter: clase encargada de llevar registro de la cantidad de threads trabajadores que aún están corriendo y de restar de dicho contador cuando algún thread ya no ejecuta más tareas. Sirve de monitor para frenar la ejecución de main hasta que termine el filtrado completo de la imagen.

Main: clase principal del programa, en primer lugar instancia un objeto de la clase Config para la toma de parámetros de la ejecución, luego toma el archivo de entrada y lo transforma, de acuerdo a lo sugerido en el enunciado de este trabajo, en un objeto de clase WritableRaster, del cual obtendrá la información de la imagen. De acuerdo al alto, ancho y cantidad de canales de la imagen de entrada, genera un WritableRaster que se usará como salida del programa. En este punto se instancia un objeto de clase Timer, al cual le envía el mensaje “startRunning()” para empezar a contar el tiempo de ejecución del programa; luego Main se encargará de inicializar el buffer, un objeto de clase WorkerCounter y un objeto de clase ThreadPool (descritos previamente). Main se encarga también de generar los objetos de clase Task y los envía al ThreadPool. Una vez que generó todas las Task, consulta con el ThreadCounter si todos los trabajadores terminaron, de no ser así, se va a dormir hasta que terminen, en ese momento genera la imágen de salida convirtiendo la matriz resultante en una imagen y termina el programa imprimiendo por pantalla el tiempo tardado en la ejecución en milisegundos.

EVALUACIÓN

Las ejecuciones del programa que usamos para obtener datos fueron hechas en una computadora con sistema operativo Windows 10 Enterprise, con un procesador Intel i7 10700 de 2.90GHz y 16 GB de RAM.

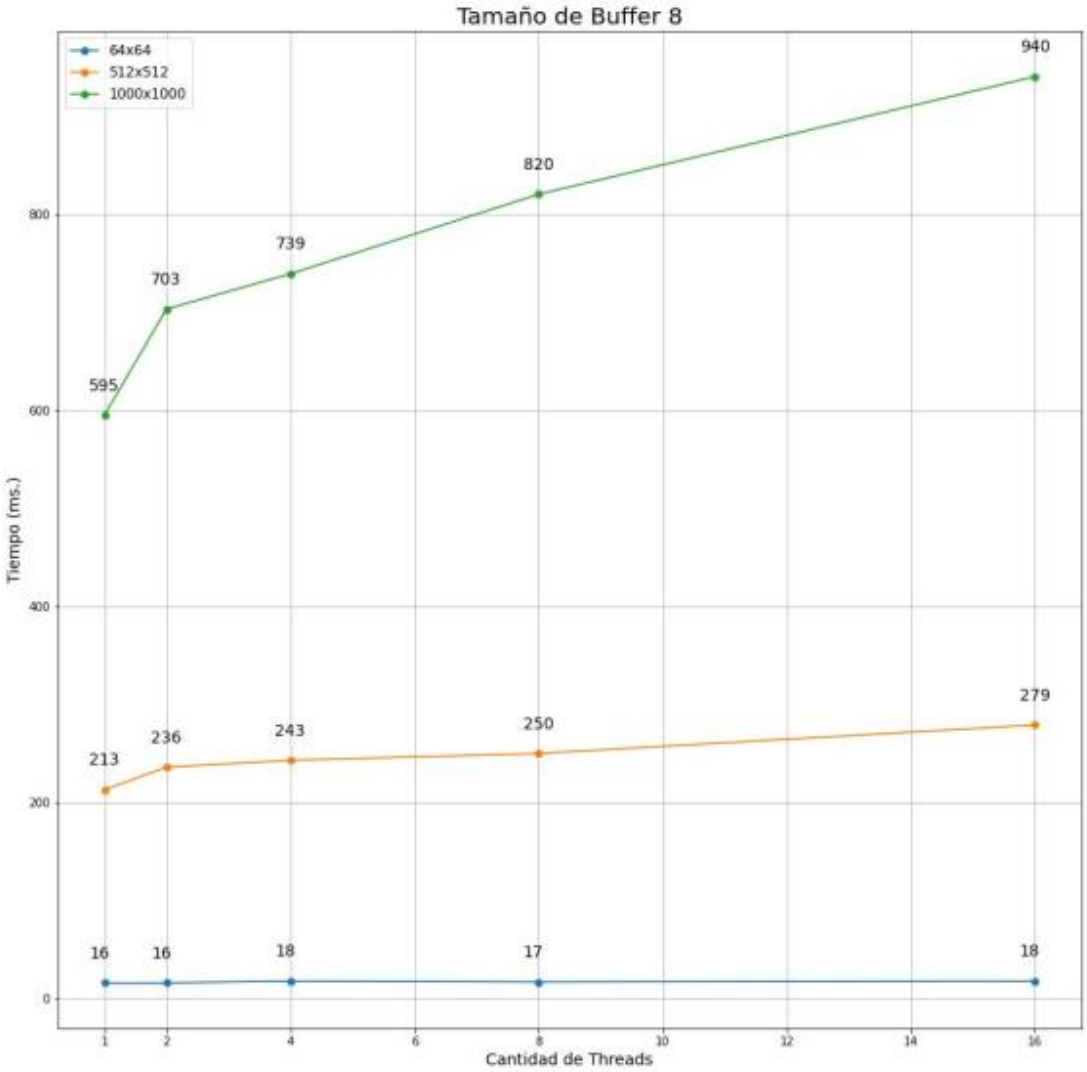
DATOS

Imagen pequeña 64*64		Threads				
		1	2	4	8	16
Tamaño del buffer	1	25 ms	24 ms	25 ms	38 ms	56 ms
	2	17 ms	18 ms	17 ms	17 ms	21 ms
	8	16 ms	16 ms	18 ms	17 ms	18 ms
	32	15 ms	15 ms	17 ms	18 ms	18 ms
	64	15 ms	15 ms	16 ms	17 ms	37 ms

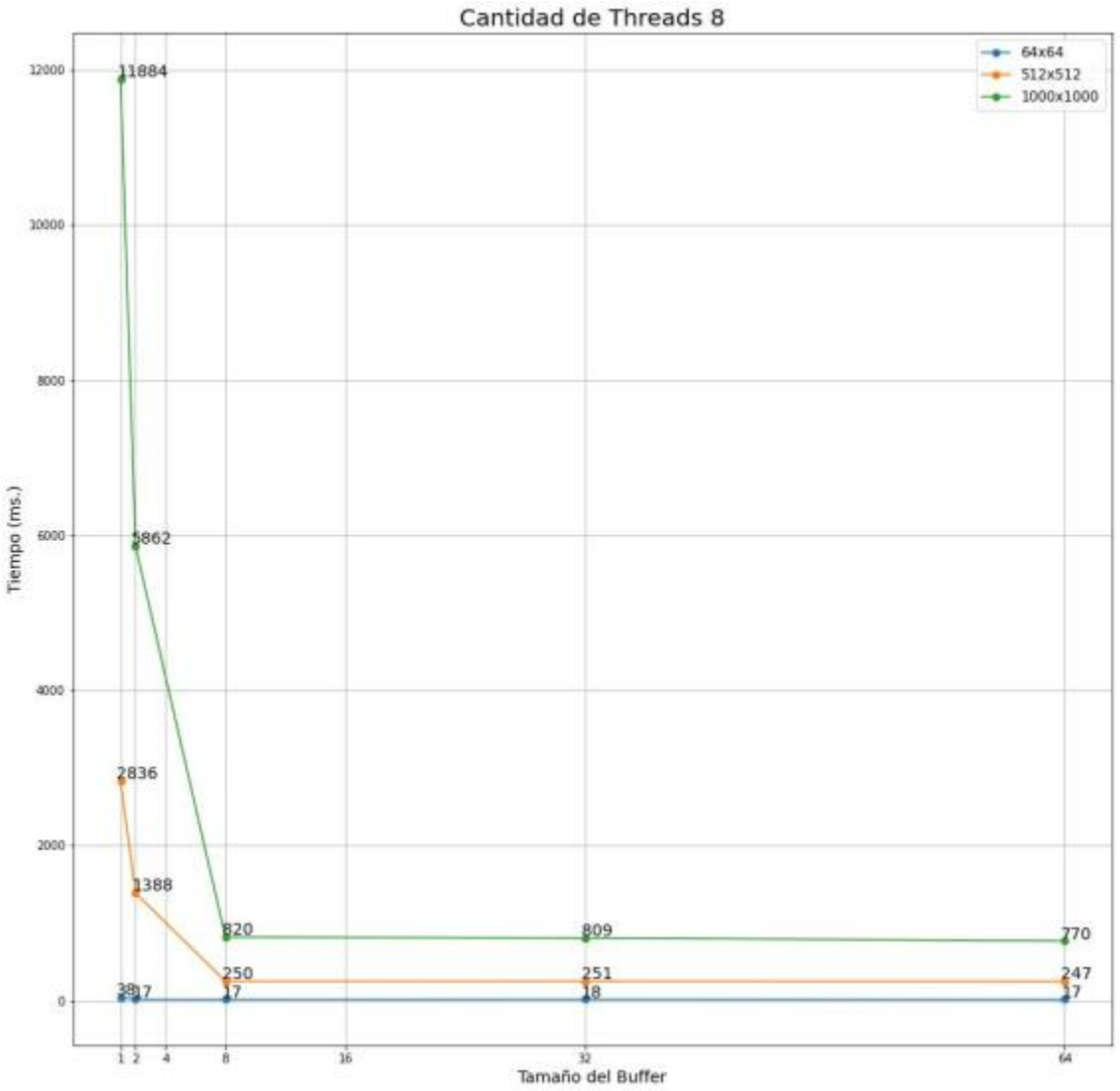
Imagen mediana 512*512		Threads				
		1	2	4	8	16
Tamaño del buffer	1	1130 ms	1333 ms	1657 ms	2836 ms	5769 ms
	2	589 ms	665 ms	820 ms	1388 ms	2510 ms
	8	213 ms	236 ms	243 ms	250 ms	279 ms
	32	208 ms	235 ms	240 ms	251 ms	267 ms
	64	212 ms	232 ms	240 ms	247 ms	260 ms

Imagen grande 1000*1000		Threads				
		1	2	4	8	16
Tamaño del buffer	1	4662 ms	5097 ms	6725 ms	11884 ms	23058 ms
	2	2839 ms	2664 ms	3110 ms	5862 ms	11113 ms
	8	595 ms	703 ms	739 ms	820 ms	940 ms
	32	622 ms	687 ms	738 ms	809 ms	876 ms
	64	609 ms	687 ms	728 ms	770 ms	869 ms

Tamaño de buffer 8



Cantidad de threads 8



CONCLUSION

Nos pareció particular, el hecho de que una mayor cantidad de threads no lleva necesariamente a un menor tiempo de resolución de los problemas, tanto el tiempo de inicialización de los threads, como también los chequeos extra que hace por cada thread el monitor (al validar la condición del while para saber si puede ejecutar o irse a dormir) hacen que el proceso demore más que si lo resolviera un sólo thread.
Es notable también que el tamaño del buffer influye mucho en el tiempo que demora el programa, sobre todo al pasar de un thread a 2 y de 2 a 8, donde se nota que a partir de este punto, agregar más espacio en el buffer no influye demasiado en el tiempo que

demora. Esto se puede deber a que los threads llegan a tomar las tareas del buffer antes que este se llene.
En los datos de la imagen más pequeña (64x64) podemos apreciar que con el tamaño de buffer 2 no hay mucha variación respecto a la cantidad de threads y agregando más tamaño al buffer, en pocas palabras se “estabiliza” el tiempo que tarda el proceso en esta configuración. El único caso donde se nota un cambio de tiempo notable, a partir de este punto, es con la configuración de 16 threads y tamaño de buffer 64, donde el tiempo de resolución es considerablemente mayor.
En los datos de la imagen mediana (512x512) podemos ver que el mejor tiempo de resolución es con la configuración de 1 thread y tamaño de buffer 32 y la configuración que más tarda es con 16 threads y tamaño de buffer 1, en este caso, creemos que se debe a que al haber un sólo espacio en el buffer, hay 15 threads que sólo se “despiertan” para chequear si pueden ejecutar una tarea y se vuelven a “dormir” y sólo 1 levanta la tarea.
Por último, en la imagen grande (1000x1000), la combinación óptima de configuración es 1 thread y tamaño de buffer 8, y la peor es similar al caso anterior con 16 threads y tamaño de buffer 1, dándose una vez más este “cuello de botella” que genera el tamaño de buffer en 1.
Este análisis nos lleva a afirmar que el tamaño de la imagen influye en la configuración óptima, como se puede notar, principalmente entre la imagen pequeña y la mediana, y entre la mediana y la grande, la configuración óptima es similar (a partir de tamaño de buffer 8 y 1 thread).
Como conclusión final podemos decir que es preferible disponer de mayor espacio en el buffer y pocos threads ejecutando, pero que también, en cierto punto, seguir incrementando el tamaño del buffer no mejora el rendimiento.