Computación paralela y distribuída Práctica 1

Nicolas Ricardo Enciso, Oscar Fabian Mendez
Universidad Nacional Bogotá, Colombia
Email:nricardoe@unal.edu.co, ofmendez@unal.edu.co

***Resumen*—El presente trabajo expone los resultados obtenidos, luego de hacer una paralelización en CPU, usando POSIX de Linux, como método de aceleración del tiempo de procesamiento, en la aplicación de un efecto borroso en imágenes (blur), la cual es una convolución con una matriz kernel gaussiana. Los resultados muestran la Ley de Amdahl como limitante del factor de aceleración, así como las limitaciones de CPU en búsqueda de paralelización.**

I. Introducción.

En el procesamiento de imágenes, es bien sabido que, el poder computacional requerido para poder hacer transformaciones sobre gráficos es elevado, debido a la gran cantidad de operaciones matemáticas que se deben hacer de forma constante, para poder lograr un resultado visible en imágenes, tiempo el cual es directamente proporcional a la cantidad de núcleos físicos con los que cuenta la CPU de la máquina host del analizador de imágenes.

Sin embargo, disparar múltiples hilos, no es garantía de un factor de aceleración acorde a la cantidad de hilos disparados. Ésto es debido a lo que se explica en la Ley de Amdahl, la cual nos ilustra que, teniendo una parte de nuestro código con posibilidad de ser paralelizado, aún así no se tendrán mejores exponenciales a mayor cantidad de hilos, eso es debido a que, se llega a un punto donde el factor de mejora no es nada valioso, inclusive, se puede llegar a un punto donde el tener múltiples instancias de hilos lanzados, creen aún más ralentización, sabiendo que, los hilos deben tambien ser ejecutados por el procesador y sus núcleos.

En cuanto a la aplicación de filtros y efectos en imágenes, la gran mayoría, por no decir todos los sistemas de modificación de imágenes, se basan en operaciones matricicales sucesivas, las cuales requieres inmensas cantidades de operaciones, convirtiéndolas en puntos críticos en los tiempos de ejecución. Para el caso concreto del presente trabajo, la aplicación del efecto borroso o blur, se hace con base en las convoluciones de imágenes. Una convolución, es una transformación a una matriz de una imagen, cambiando os valores de los pixeles, por medio de operaciones de multiplicación, entre la matriz imagen, y un kernel o filtro, el cual es una matriz cuadrada de tamaño impar, que va recorriendo uno a uno los pixeles de la imagen, calculando un nuevo valor de acuerdo a valores obtenidos por multiplicaciones con sus vecinos, y generalmente, luego divididos por el total de la suma de todos los valores computados, para normalización.

Debido a tener una naturaleza matricial, de forma inherente, cada operación hecha a cada pixel, es independiente de las demás, lo que convierte el proceso de convolución muy paralelizable, debido a la no dependencia, factor clave a la hora de determinar el potencial de aceleración de un programa por medio de paralelización.

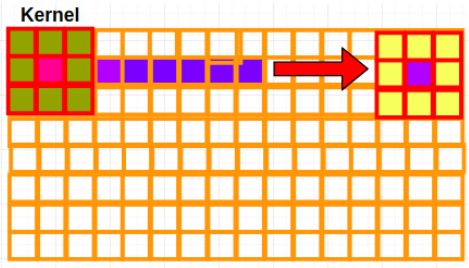

Figura 1: Recorrido del kernel en la matriz imagen haciendo convolución

A continuación se presentará el algoritmo contreto para la contrucción del kernel filtro blur para hacer la convolución, luego la estrategia de paralelización, las herramientas y librerías usadas, así como la composición del código de convolución resultante.

I-A. Convolución de imágenes.

El proceso de convolución, consiste en la multiplicación uno a uno de los valores de una matriz cuadrada impar, el kernel filtro, con una matriz, la imagen. La operación no es la típica de multiplicación matricial, sino que, el kernel es puesto sobre cada pixel de la matriz imagen, teniendo como objetivo, el pixel ubicado en el centro del kernel, el cual será el que tendrá las mayores modificaciones. Por supuesto, los valores dependen del efecto que se quiera lograr, desde detección de imagenes, poniendo valores medios en los vecinos del pixel objetivo, hasta el que se maneja en el presente trabajo, el cual es la expansión decreciente de los valores, desde el centro hacia los lados de la matriz.

En el proceso de aplicación de las convoluciones, se debe tener un criterio predefinido, en el caso de alcanzar los bordes de la imagen, teniendo ”salidas”del kernel sobre la imagen, teniendo que definir los límites con los cuales se pueda operar, es decir, los límites que permitan tener multiplicaciones de valores, uno a uno, con pixeles existentes. Adicionalmente, se debe tener en cuenta que, a mayor tamaño del kernel, mayor es el tiempo requerido para la aplicación de la convolución, debido a que se deben hacer más multiplicaciones de cada i,j valor del kernel, con cada x,y pixel de la imagen, además de luego tener que aplicar una suma de valores, y luego una

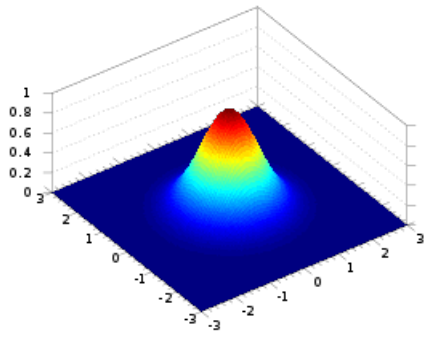

Figura 2: Función gaussiana 3D

división al actualizar el pixel objetivo. Cabe la aclaración trivial de que, a mayor resolución de la imagen, mayor cantidad de pixeles, y por tanto, el kernel debe moverse más posiciones, haciendo que se tengan aún más operaciones.

I-A1. Efecto borroso blur Gaussiano: El algoritmo seleccionado para el efecto blur, fue el gaussiano. Ese algoritmo presenta un aprovechamiento de la función gaussiana, o de la distribución normal, la cual crea una campana de Gauss, con su mayor valor en el centro de la gráfica (ver imagen).

Como podemos apreciar en la imagen, la campana de gauss se puede generalizar a n dimensiones, teniendo siempre claro, que el valor va disminuyendo a medida que se aleja del centro, a valores iguales. Adicional, el decrecimiento de los valores en los vecinos del punto más alto de la campana, es simétrico y de forma radial. Esta característica es aprovechada en la creación del kerenl filtro, debido a que, si se ponen valores altos en el centro, y se van disminuyendo a medida que se aleja del punto, el pixel se mostrará más fuerte en su centro, pero se irá degradando en sus vecinos, de forma que, en una imagen, se tendrán distribuciones normales en cada pixel, difuminando los valores de forma decreciente, haciendo ver a la imagen, luego de la transformación, como si fuera borroso.

A continuación, vemos un ejemplo de una matrix kernel producto de la aplicación de una función gaussiana de n=5:

$$\begin{bmatrix} 0{,}00296 & 0{,}0133 & 0{,}0219 & 0{,}0133 & 0{,}00296 \\ 0{,}0133062 & 0{,}0596343 & 0{,}0983 & 0{,}0596 & 0{,}0133 \\ 0{,}0219 & 0{,}0983 & \mathbf{0.162103} & 0{,}0983 & 0{,}0219382 \\ 0{,}0133 & 0{,}0596343 & 0{,}0983203 & 0{,}0596343 & 0{,}0133062 \\ 0{,}00296 & 0{,}01330 & 0{,}02193 & 0{,}01330 & 0{,}00296 \end{bmatrix}$$

Como se puede observar, el valor en negrilla es el centro y es el que tiene mayor valor, pensemos ese centro como el punto medio de la campana de Gauss, en donde se tiene el valor más grande de la función, de esa forma se emula ese comportamiento, en la creación del kernel filtro.

En el programa implementado, para valores que fueran pares a la hora de definir el tamaño del kernel, se le sumaba uno, con el fin de convertir siempre la entrada en un impar, para poder crear bien el kernel filtro.

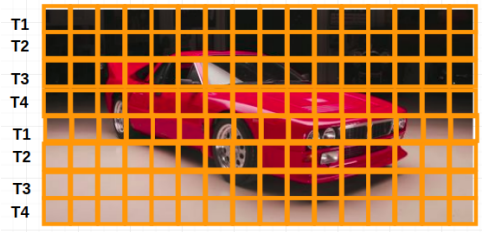

Figura 3: Asignación cíclica de hilos a cada fila

II. Paralelización del algoritmo.

La paralelización se basó en la idea de **cyclic data**, es decir, con una distribución de datos que permitiera asignar una carga de trabajo equitativa a cada hilo, aprovechando al máximo el potencial de aceleración. La idea fundamental, es la de partir en filas la matrix imagen, pensando en una dimensión, asignar a un hilo, una fila, y así hasta que, al asignar todos los n hilos a las primeras n filas de la imagen, volver y asignar una nueva fila al primer hilo con la primera fila, y asi sucesivamente hasta barrer todas las filas de la imagen. Este procedimiento permite que, en el peor de los casos, un hilo tenga apenas una fila de más en su carga de trabajo, respecto los demás hilos, lo que asegura una distribución equitativa, consistente y sobre todo optimizada para paralelización. Además, al ser la convolución un procedimiento independiente en cada iteración, no se tendrá inconsistencias ni condiciones de carrera a la hora de reconstrutir la imagen con los nuevos valores.

El barrido tipo **cyclic data** para hacer la convolución, se diferencia del block cyclic, en la medida en que se le asigna a cada hilo una fila, que no forma un bloque, sino que es una única unidad. En el momento en el que se le asigna una nueva fila, ésta no es contínua, sino que es cíclica, el seguiente hilo tomará la siguiente fila, y así sucesivamente. A continuación veremos una gráfica explicativa del proceso de asignación tipo **cyclic data** de la distribución de carga en la imagen:

Como podemos ver, se tienen 4 hilos (T1, T2, T3, T4), para las cuales se les asignan en ese orden las 4 primeras filas. Al terminar se le asignan en el mismo orden las demás filas, de forma que se hace cíclica, todos con la misma carga de trabajo en unidades pequeñas.

III. Experimentos y resultados.

Las pruebas se realizaron en una torre PC, con un AMD FX 8350 Black Edition de 8 cores físicos a 4Ghz, junto con 8GB de RAM DDR4 y corriendo Ubuntu 18.04 LTS. El programa fue hecho en C++ y usa OpenCV para hacer tratamiento de imagenes. Hay que tener en cuenta, que permite la modificación en cada pixel, de sus tres componentes RGB.

A continuación se muestran los resultados obtenidos, hasta donde se puedo correr el algoritmo con un kernel de 8:

Kernel Size	Num_threads	TIme (ms)	speed up
8	1	1677,08	0,1839804899
8	2	858,987	0,3592021765
8	4	440255	0,000365131571
8	8	249,048	0,3840793743
8	16	453,376	0,1152345956

Figura 4: Tiempos y speedup con kernel 8 (8+1) imagen HD

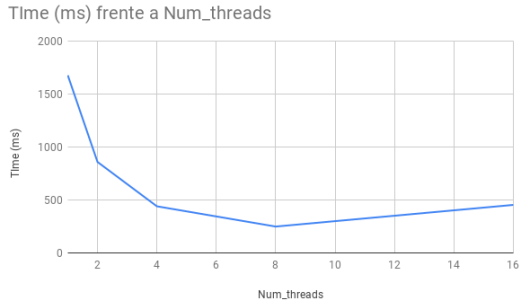

Figura 5: Gráfica tiempo vs hilos imagen HD

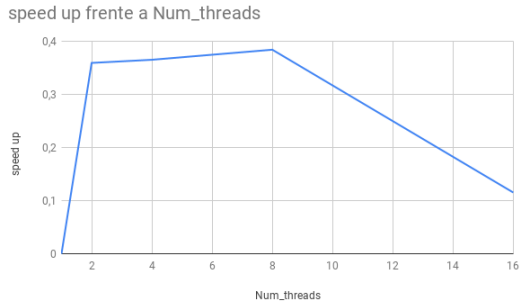

Figura 6: Gráfica speed up vs hilos imagen HD

Kernel Size	Num_threads	TIme (ms)	speed up
8	1	4390,84	181,3197475
8	2	2251,12	0,3536666193
8	4	1193,1	0,6672919286
8	8	641,368	1,241324793
8	16	1239,07	0,6425351272

Figura 7: Tiempos y speedup con kernel 8 (8+1) imagen FHD

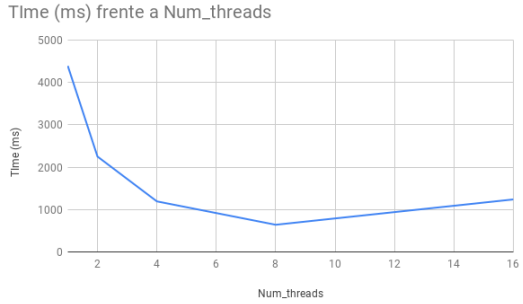

Figura 8: Gráfico tiempo vs hilos imagen FHD

CONCLUSIONES.

Como se puede apreciar, luego de 8 hilos, el speed up y los tiempos de respuesta no siguen mejorando, sino que

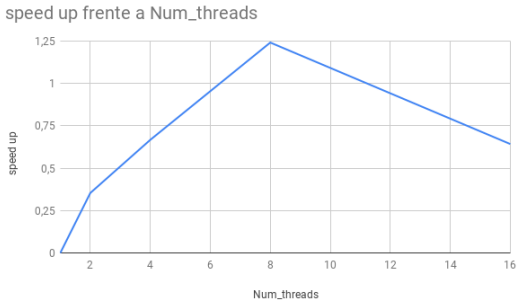

Figura 9: Gráfico speed up vs hilos imagen FHD

Kernel Size	Num_threads	TIme (ms)	speed up
8	1	17448,6	0,1804242174
8	2	6906,95	0,4557945258
8	4	4519,85	0,6965164773
8	8	2526,62	1,24599267
8	16	4967,37	0,6337659566

Figura 10: Tiempos y speedup con kernel 8 (8+1) imagen 4k

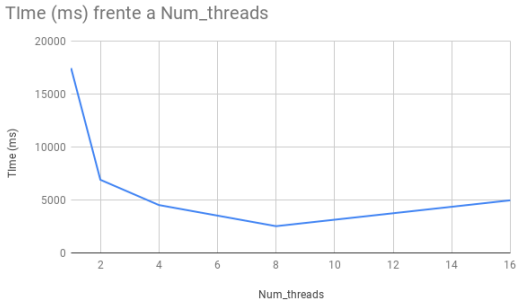

Figura 11: Gráfico tiempo vs hilos imagen 4k

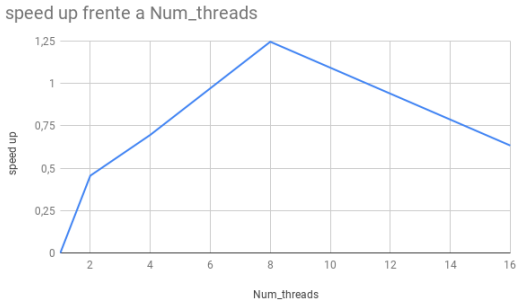

Figura 12: Gráfico speed up vs hilos imagen 4k

incluso empeoran respecto al comportamiento que llevaban, ésto se explica en la medida en que hasta 8 son los núcleos con los que contaba el sistema, además de que, a medida que iba aumentando la cantidad de hilos, éstos se convertían más en estorbo para el procesador, por lo que los tiempos no mejoraban.

Adicionalemente, podemos observar que el factor de aceleración máximo que se alcanzó, fue de 1.25x de mejora, lo que en realidad es un valor importante, pero no uno como

se podría pensar de hasta 8 veces más rápido, en realidad se cumple a cabalidad la Ley de Amdahl, por lo cual, no tiene sentido mandar más de 8 hilos en el caso del sistema usado para el presente trabajo.

Como conclusión, debemos señalar que vemos que no es funcional enviaran una gran cantidad de hilos por si, sino que hay que saber que existe un límite de mejora en la paralelización, por lo que, primero hay que saber hasta cuántos hilos es esa frontera, pero además, que si se quiere mejorar por factores muchos mayores, la mejora en el algoritmo más que en la cantidad de hilos es donde se tendrá un mejor resultado.