Aplicación de efecto borroso en imagen usando programación paralela

Víctor Gabriel Ramírez Caballero[#1], Daniel Augusto Cáceres Salas[*2]

[#]*Facultad de Ingeniería, Universidad Nacional de Colombia*

[1]vgramirezc@unal.edu.co

[2]daacaceressa@unal.edu.co

Bogotá D.C, Colombia

***Abstract*— Este documento provee información, ejemplos y conclusiones acerca de la paralelización del algoritmo de efecto borroso sobre imágenes de diferentes resoluciones (720p, 1080p, 4K). Las pruebas fueron realizadas con una CPU con 4 cores reales y 4 virtuales; también se realizó con una GPU Quadro K2000 la cual posee 2 multiprocesadores con 192 cores cada uno. Los temas principales a tratar son: balanceo de cargas, speedup, algoritmo box blur, hilos y paralelismo.**

***Keywords*—** Hilos, Speedup, Paralelismo, CUDA, OpenMP.

I. Introducción

El algoritmo de box blur (también conocido como filtro lineal de cuadro) es un filtro lineal de dominio espacial en el que cada píxel de la imagen resultante tiene un valor igual al promedio de los valores de sus píxeles vecinos en la imagen original. Los píxeles vecinos son aquellos que hacen parte de una matriz cuadrada en cuyo centro está el píxel que se va a calcular, el tamaño de esta matriz es llamado kernel y debe ser definido y fijado antes de la ejecución del algoritmo.

Debido a que todos los píxeles vecinos tienen el mismo peso sobre el píxel que se va a calcular, se puede optimizar este algoritmo mediante el uso de un acumulado, el cual es mucho más sencillo que utilizar una técnica como sliding window (ventana deslizante). Sin embargo para poder apreciar mejor la paralelización del algoritmo, la implementación utilizada en esta práctica no tiene ninguna optimización.

Se utilizó la librería OpenCV en C++, esta permite la manipulación de imágenes en el código representandolas como una matriz de enteros correspondientes a los valores RGB de cada píxel y modificar los valores con fines del efecto deseado.

Para la parte paralela se utilizaron las librerías lpthread para la creación, manejo y destrucción de hilos de manera manual; omp para la misma función pero automatizada bajo la inserción de directivas (pragmas); y por último el compilador de nvcc para trabajar con código de extensión .cu a la hora de utilizar CUDA.

II. Paralelización del algoritmo

El programa permite trabajar con 1, 2, 4, 8 y 16 hilos cuando se ejecuta sobre la CPU y con 48, 96, 192, 384, 576, 768 y 960 hilos cuando se ejecuta sobre la GPU. El algoritmo se encarga de dividir la cantidad total de filas en la imagen entre la cantidad de hilos usados, de tal forma que todos procesen la misma cantidad de filas. Para lograr esto se usa la función blurRows que recibe como parámetro el id del hilo que ejecuta dicha función, y con este mismo calcula que filas le corresponden. A continuación se muestra un ejemplo de la distribución de trabajo usando 4 hilos:

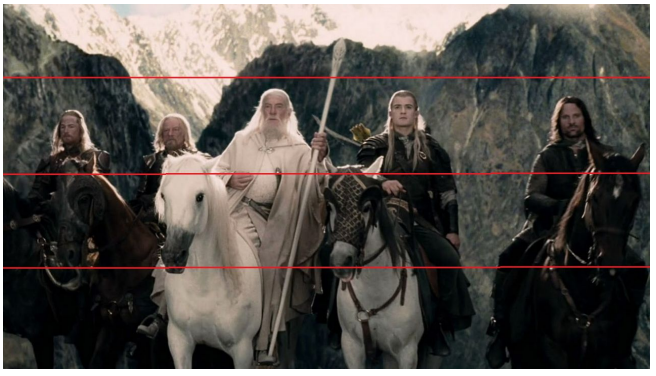

Fig. 1 Distribución de trabajo para 4 hilos.

Fig. 2 Resultado final del algoritmo de efecto borroso.

III. Experimentos y resultados

Las pruebas fueron realizadas sobre 13-15 imágenes diferentes para cada una de las resoluciones mencionadas, las imágenes se encuentran en carpetas nombradas de acuerdo a la resolución de la imagen. El script abre estas carpetas y procesa todas las imágenes con extensión .jpg, a medida que procesa una imagen, este genera tres archivos de texto (uno para cada resolución) en el cual se encuentra un log de todas las imágenes que procesó junto con el tamaño del kernel, número de hilos y el tiempo que tardó en ejecutar el algoritmo de efecto borroso.

File	Kernel	Threads	Time(s)
4K/acdc.jpg	3	1	1.9731
4K/acdc.jpg	3	2	1.0080
4K/acdc.jpg	3	4	0.9224
4K/acdc.jpg	3	8	0.4933
4K/acdc.jpg	3	16	0.5260
4K/acdc.jpg	5	1	4.5403
4K/acdc.jpg	5	2	2.2960
4K/acdc.jpg	5	4	2.1274
4K/acdc.jpg	5	8	1.1197
4K/acdc.jpg	5	16	1.1524

Fig. 3 Ejemplo del log creado para la resolución 4K ejecutado sobre la CPU

Una vez el script haya terminado de procesar todas las imágenes, este ejecuta un programa en python el cual lee los logs previamente mencionados y genera dos gráficas por resolución, una que muestra el tiempo que tarda en ejecutar el algoritmo para diferentes tamaños de kernel y diferente número de hilos, y otra gráfica que muestra el speed up que se obtiene mediante la paralelización.

A. POSIX

Como se puede observar en las siguientes gráficas, se obtuvo una mejora de tiempo considerable hasta los 4 hilos y una pequeña mejora al utilizar 8, esto se debe a que la CPU tiene 4 cores reales y 4 virtuales, una vez se llega a 16 hilos no se obtiene ninguna mejora con respecto al anterior ya que se deben realizar cambios de contexto para poder ejecutar el programa con esta cantidad de hilos.

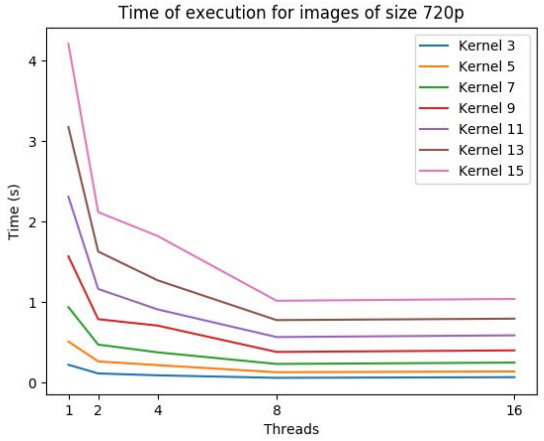

Fig. 4 Tiempo de ejecución para imágenes de resolución 720p en POSIX.

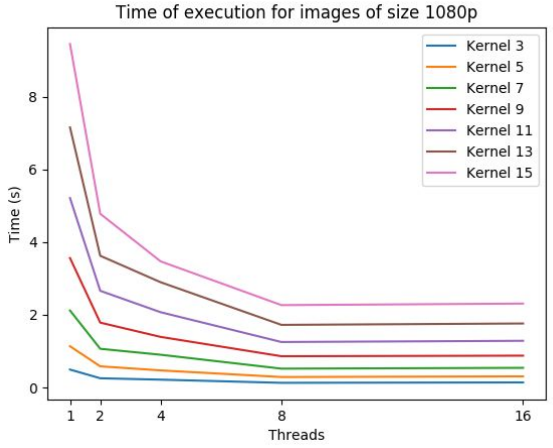

Fig. 5 Tiempo de ejecución para imágenes de resolución 1080p en POSIX.

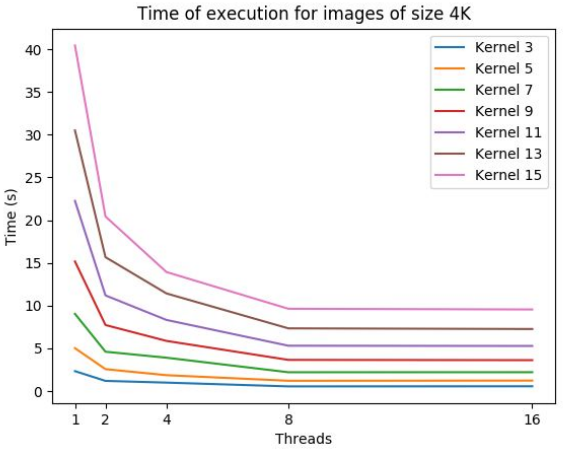

Fig. 6 Tiempo de ejecución para imágenes de resolución 4K en POSIX.

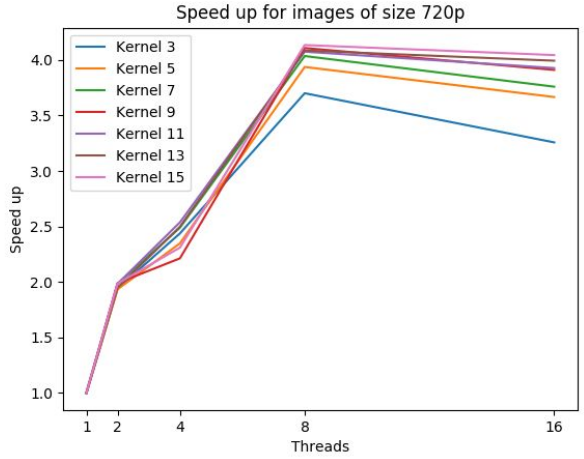

Fig. 7 Speed up obtenido en imágenes de resolución 720p en POSIX.

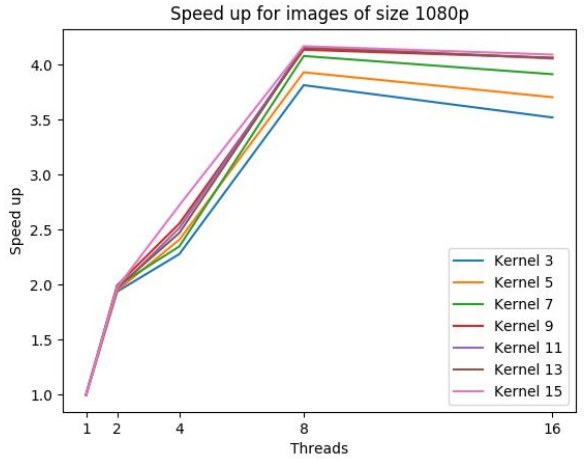

Fig. 8 Speed up obtenido en imágenes de resolución 1080p en POSIX.

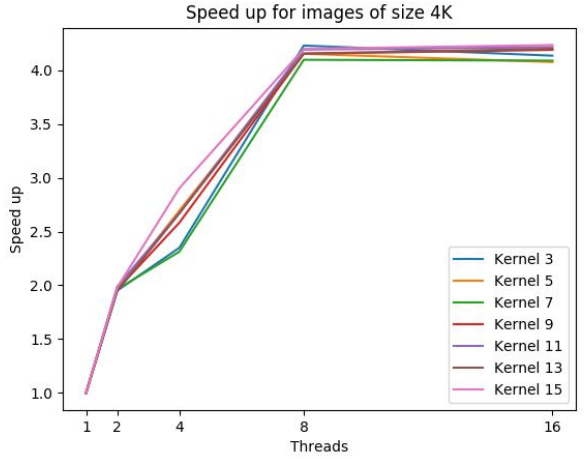

Fig. 9 Speed up obtenido en imágenes de resolución 4K en POSIX.

B. OpenMP

Si comparamos las gráficas obtenidas con OpenMP y POSIX, se evidencia que los tiempos son prácticamente los mismos, esto se debe a que OpenMP por debajo realiza lo mismo que POSIX creando, manejando y destruyendo los hilos. La ventaja que provee OpenMP es la facilidad a la hora de programar ya que con unas cuantas líneas de código se pueden obtener los mismos resultados que con muchas líneas escritas en POSIX.

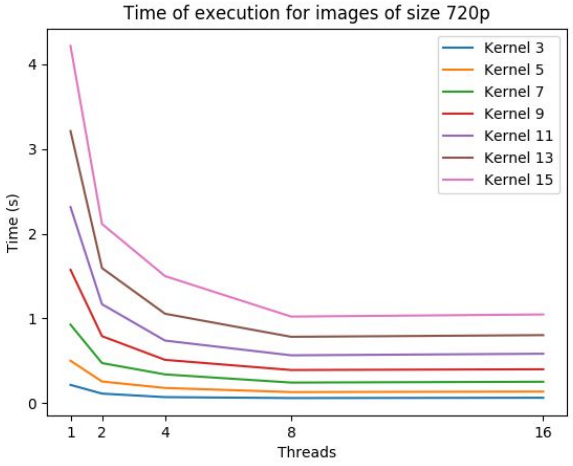

Fig. 10 Tiempo de ejecución para imágenes de resolución 720p en OpenMP.

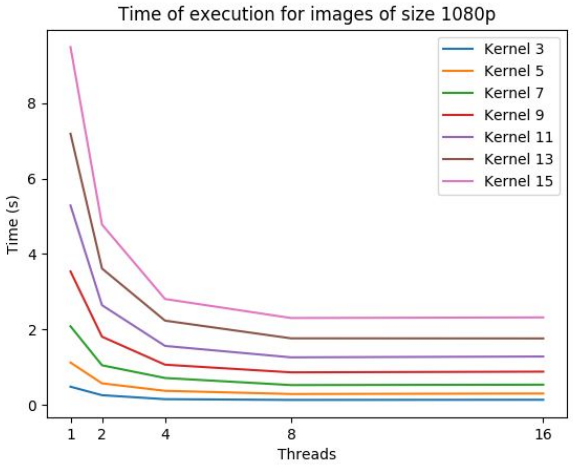

Fig. 11 Tiempo de ejecución para imágenes de resolución 1080p en OpenMP.

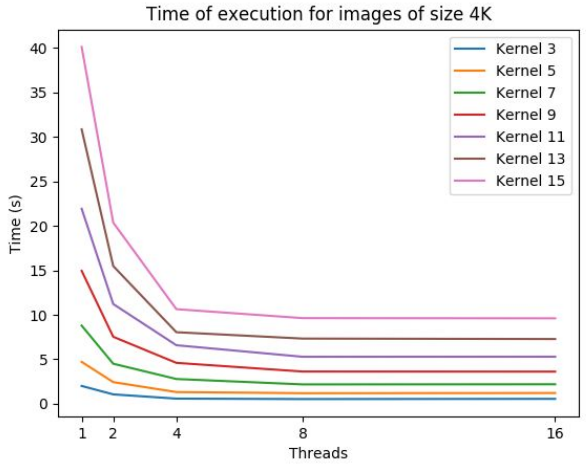

Fig. 12 Tiempo de ejecución para imágenes de resolución 4K en OpenMP.

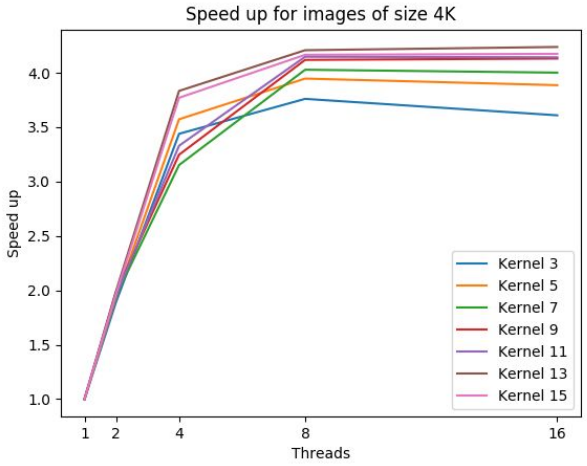

Fig. 15 Speed up obtenido en imágenes de resolución 4K en OpenMP.

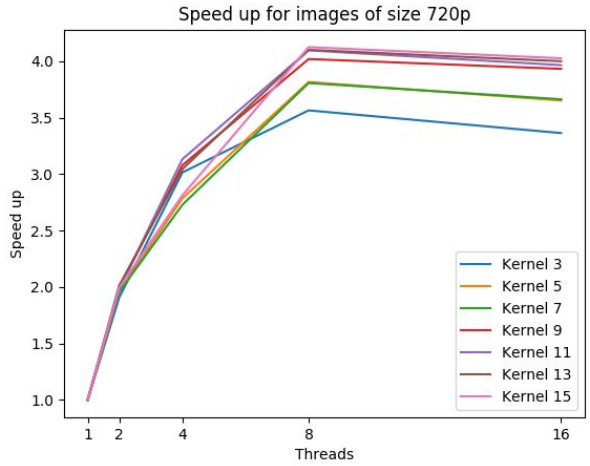

Fig. 13 Speed up obtenido en imágenes de resolución 720p en OpenMP.

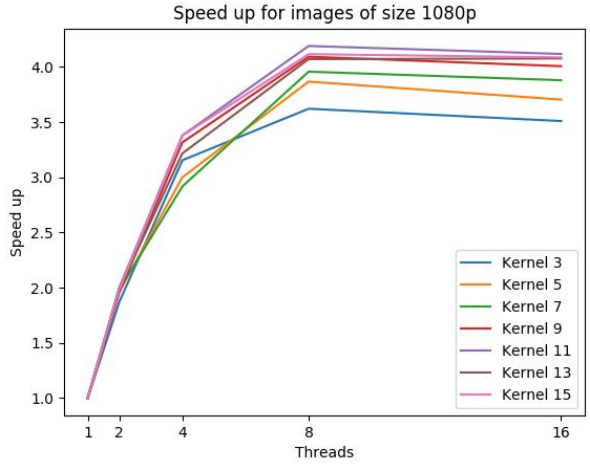

Fig. 14 Speed up obtenido en imágenes de resolución 1080p en OpenMP.

C. CUDA

Al ejecutar sobre la GPU se tiene la posibilidad de utilizar más hilos sin necesidad de cambios de contexto, para este caso se utilizó una Quadro K2000 con 2 Multiprocesadores y 192 cores en cada uno, de esta manera se puede evidenciar en las gráficas que el aumento de tiempo es considerable hasta los 768 hilos, una vez se llega a 960 el tiempo empieza a aumentar ya que el costo que tiene el cambio de contexto extra (que no era necesario en 768) no es suficiente para sacar provecho de la cantidad de hilos dada la cantidad de filas que procesa cada uno de estos.

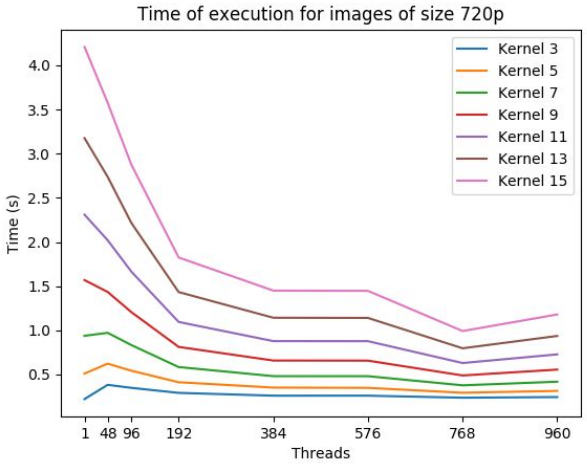

Fig. 16 Tiempo de ejecución para imágenes de resolución 720p en CUDA.

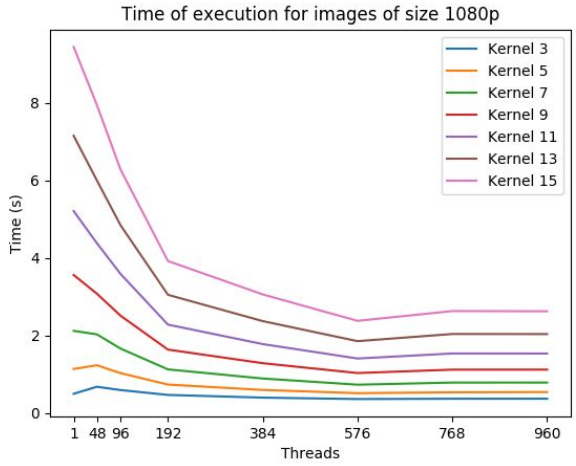

Fig. 17 Tiempo de ejecución para imágenes de resolución 1080p en CUDA.

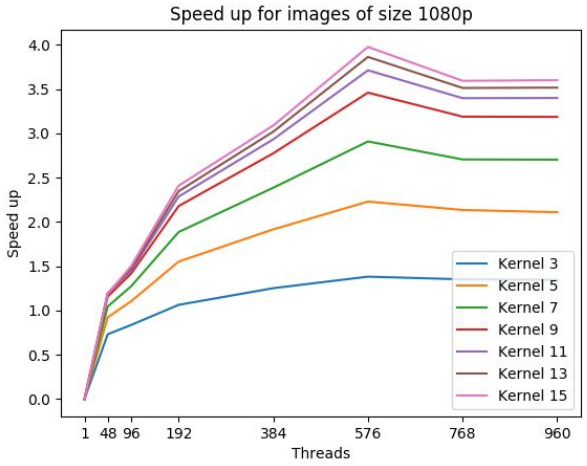

Fig. 20 Speed up obtenido en imágenes de resolución 1080p en CUDA.

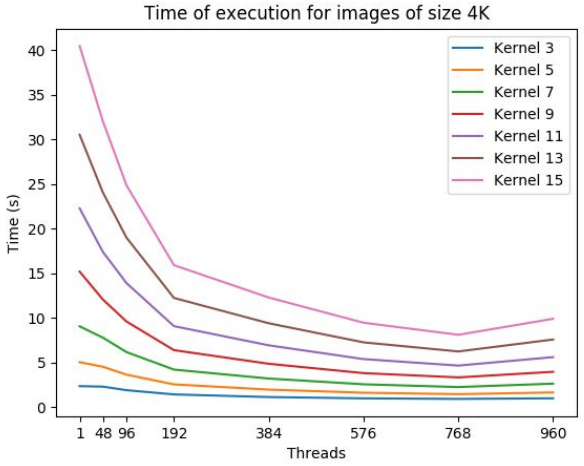

Fig. 18 Tiempo de ejecución para imágenes de resolución 4K en CUDA.

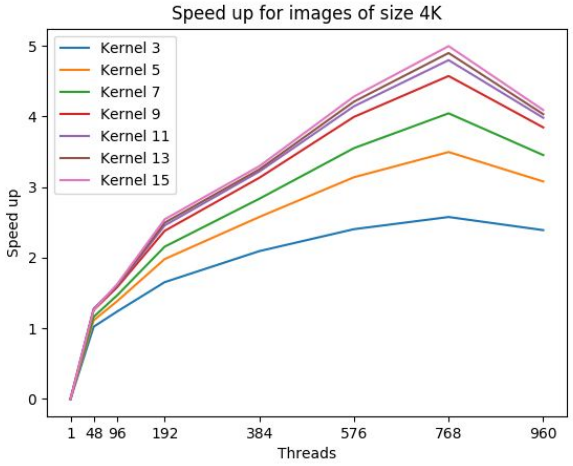

Fig. 21 Speed up obtenido en imágenes de resolución 4K en CUDA.

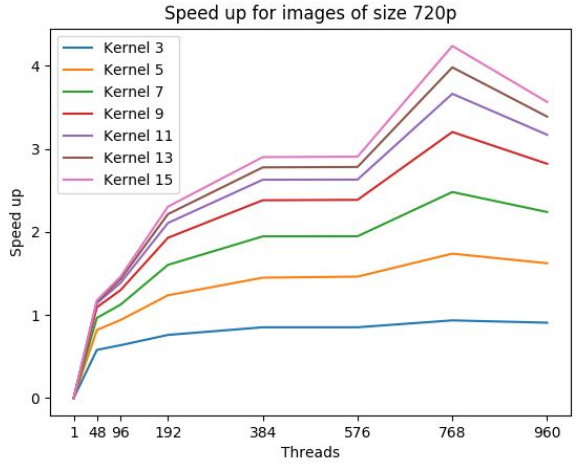

Fig. 19 Speed up obtenido en imágenes de resolución 720p en CUDA.

IV. Conclusiones

El algoritmo box blur es un claro ejemplo del concepto de embarazosamente paralelo (Embarrassingly Parallel), en el cual se necesita poco o ningún esfuerzo para separar el problema en una serie de tareas paralelas. Esto ocurre ya que no existe ninguna dependencia de datos ni necesidad de comunicación entre tareas paralelas.

Al realizar la paralelización es natural encontrarse con el resultado de que el tiempo de ejecución disminuye a medida que la cantidad de hilos aumenta, sin embargo existe un límite para este decremento, el cual está dado no sólo por la cantidad de código que se ejecuta de manera secuencial (tal y como fue explicado en clase), sino también por la capacidad física de la máquina.