Informe final - Efecto borroso de una Imagen utilizando varios métodos de paralelización

Nicolás Restrepo Torres, Lizzy Tengana Hurtado
Universidad Nacional Bogotá, Colombia
nrestrepot@unal.edu.co, ltenganah@unal.edu.co

***Resumen*—El siguiente documento busca presentar el análsis de las tres prácticas en el marco de la asignatura *Computación Paralela y Distribuida* en dónde se muestra la implementación de un algoritmo de desenfoque utilizado sobre imágenes de distintos tamaños. Esto con el fin de mostrar los efectos que tiene la paralelización en el procesamiento de las imágenes y analizar los resultados obtenidos en cada una de las prácticas para poder contrastarlos de acuerdo con los objetivos de paraliezación para un último fin que es el procesamiento de imágenes. Se ha implementado el algoritmo de forma secuencial, con hilos POSIX y con la librería OpenMP para hacer análisis de paralelización en la CPU. Se ha implementado con CUDA para hacer análisis de paralelización en la GPU. Y finalmente se ha implementado en Open MPI para análizar su rendimiento como aproximación de computación distribuida para ser ejecutado en un pequeño cluster. Esto demostrará que según la cantidad de hilos o de máquinas y el tamaño del kernel se tendrán mejoras notables y otros tiempos muchos más específicos de mejora al usar la unidad de procesamiento de gráficos central por ejemplo. Se llega a la conclusión de los efectos notables que se tienen al usar diferentes técnicas de paralelismo.**

I. Librerías utilizadas

Para el procesamiento y manejo de imágenes se empleo la librería OpenCV para C/C++ la cual permitió cargar una imagen representada como una matriz de enteros (grupo de 3 enteros) correspondientes a los canales RGB y modificarla de acuerdo a los objetivos que se plantearon. Además se utilizaron otras librerías de propias de C++ para lograr el procesamiento de otras partes complementarias del programa, especialmente en la parte de CUDA.

II. Procedimiento

Se optó por implementar la idea del algoritmo Gaussiano y el algoritmo de Box Blur debido a unos detalles específicos de la implementación con CUDA en cuyo caso la distribución de los hilos es distinta y hace que cambie un poco el paradigma, detalles que se explicarán más adelante en la subsección correspondiente.

II-A. Procedimiento en CPU/Cluster

Box Blur consiste en un kernel de tamaño $n \times n$ que no es más que una matriz que circunda a uno de los píxeles en un momento dado. Este píxel se pone con en el centro con respecto a los demás pixeles que consisten en la matriz en uno de los cuatro puntos centrales o bien en el centro exacto del kernel. Esto ocurre en uno de dos casos en que el tamaño del kernel sea un número impar o un par, para un tamaño impar de kernel será siempre un único punto y en el caso de que sea par serán cuatro puntos de los cuales es indiferente escoger cualquiera de estos.

Se separan las componentes RGB que es el formato predeterminado que maneja la librería OpenCV en su valor de 0 a 255 para R, G Y B y se hace un procedimiento análogo para cada una de ellas. Se calcula un promedio simple con los pixeles que encierran al kernel descrito anteriormente y cada componente que contenga al kernel. Se verifica que el pixel es uno válido para determinar si entra en el promedio, esto es, que el tamaño del kernel en los bordes puede desbordarse e igual la imagen no va a perder ninguna fila ni ninguna columna porque se calcula el promedio con base a los pixeles que puedan ser encerrados por el kernel.

Finalmente se reemplaza el número de cada una de las componentes con el número promedio que haya sido retornado. Posteriormente se carga la información de la imagen un archivo mediante la estructura de datos Mat que genera OpenCV para notar el resultado de borrosidad. Esta librería no es más que una matriz especial de tres dimensiones con cada una de las componentes y capaz de representar una imagen en un formato usual como JPEG, PNG y demás, dados los valores para cada componente de la matriz.

II-B. Procedimiento en GPU

Aquí se tiene entonces una misma idea básica de la implementación explicada anteriormente con el Box Blur, que es tomar una serie de pixeles circundantes y mezclarlos de alguna forma con el pixel actual lo que provoca que la imagen pierda algunos detalles minúsculos y de esta forma hace que la imagen se vea más borrosa.

El algoritmo Gaussiano consiste en un kernel de tamaño $n \times n$ que no es más que una matriz que circunda a uno de los píxeles en un momento dado. Este píxel se pone con en el centro con respecto a los demás pixeles que consisten en la matriz en uno de los cuatro puntos centrales o bien en el centro exacto del kernel. Esto ocurre en uno de dos casos en que el tamaño del kernel sea un número impar o un par, para un tamaño impar de kernel será siempre un único punto y en el caso de que sea par serán cuatro puntos de los cuales es indiferente escoger cualquiera de estos.

Se separan las componentes RGB que es el formato predeterminado que maneja la librería OpenCV en su valor de 0 a 255 para R, G Y B y se hace un procedimiento análogo para cada una de ellas. Es aquí cuando tenemos la abstracción del kernel para cada uno de los píxeles con sus respectivas compontes y se aplica una función gaussiana a la matríz kernel, la cual será del tamaño que el usuario desee. Para esto se usó la ecuación:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2 y^2}{2\sigma^2}}$$

En las figura 1 se puede observar la aplicación de la función gaussiana a una matriz.

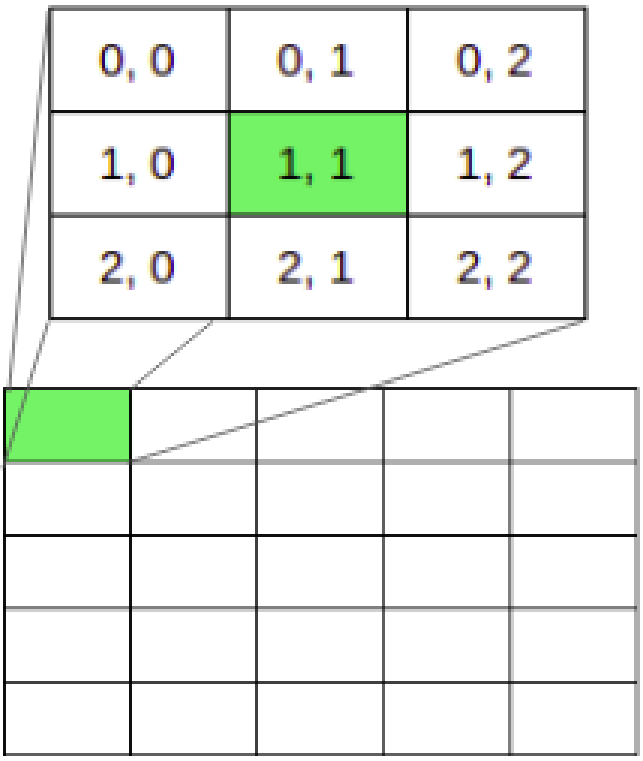


Figura 1: Matriz de posiciones para cambiar el efecto gaussiano.

Posteriormente, para aplicar el efecto gaussiano, se realiza el proceso de multiplicar la matriz kernel obtenida de la función anterior, con los pixeles de la imagen. Esto se hace tomando cada pixel y los pixeles que lo rodean en un radio de kernel$/2$. Para cada uno de estos pixeles se obtiene el valor RGB, mediante la librería utilizada, y cada valor se multiplica con el valor de la celda de la matriz gaussiana correspondiente. Despúes de esto se suman los valores resultantes de la multiplicación y este será el valor correspondiente al RGB del pixel tomado. Este proceso se realiza para cada uno de los pixeles de la imagen. En las siguientes imágenes se observa la multiplicación realizada para cada pixel asumiendo los valores para un color. Esto se puede observar en las figuras 2 y 3

III. Balanceo de carga

Se especifican distintas tácticas de balance

14	15	16
24	25	26
34	35	36

Figura 2: Valores de color verde (0-25) en una región de la imagen.

1.32638	1.77477	1.51587
2.83863	3.69403	3.07627
3.22121	4.14113	3.4107

Figura 3: Resultado de la aplicación del efecto.

III-A. CPU

El procesamiento de cualquier imagen bajo este esquema está dado por el número de pixeles precisamente porque el número de promedios a calcular. Se ejecuta entonces una columna de pixeles determinada para cada hilo, el método usado aquí sirve igualmente en caso de que se quisieran ejecutar filas para cada uno de hilos. El balanceo de carga entonces fue realizado por clases de equivalencia, esto quiere que para n hilos, donde cada hilo está identificado únicamente con un id incremental desde 0, se tiene que cada hilo procesa la columna $n + (c \times x)$ del proceso que calcula el promedio de los pixeles, donde c es el número de hilos, n es el id incremental y x es un escalar que determina cuál columna se ejecutará en la siguiente iteración del hilo dado. Esto quiere decir que para el hilo 0 entonces la clase de equivalencia se refiere a la clase de equivalencia del 0 que va a procesar las columnas que satisfagan $i + \sum_{i=i}^{x} c$, donde $0 \leq x \leq w$, siendo w el tamaño de columnas de la imagen. Análogamente se haría el mismo proceso para las filas limitado por el número de filas. Una forma gráfica de verlo es la siguiente, el hilo 0 representado por el color azul, hilo 1 representado por el color amarillo y hilo 2 representado por el color rojo. Y se ejecutan los pixeles de esa columna tal cual sea el hilo que corresponda, una explicación gráfica se puede evidenciar en la figura 4.

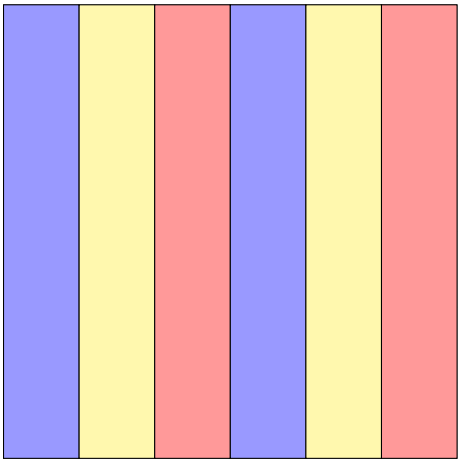


Figura 4: Particionamiento del problema (block-wise)

III-B. GPU

Para realizar la multiplicación de manera paralela, se planteó dividir la matriz en el número de procesos o hilos a utilizar, y de esta manera cada unidad de procesamiento se encargaría de operar únicamente las filas que le correspondieran.

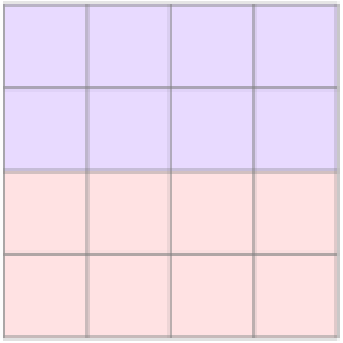


Figura 5: Subdivisión de cargas

En la figura 5, podemos ver un ejemplo de la subdivisión de cargas para dos unidades de procesamiento. En la primera, se dedicará a multiplicar las filas de color azul por la otra unidad de procesamiento rojas. Cada resultado dado, se agregará en la matriz resultado.

Finalmente se carga la información de la imagen un archivo mediante la estructura de datos Mat que genera OpenCV para notar el resultado de borrosidad. Esta librería no es más que una matriz especial de tres dimensiones con cada una de las componentes y capaz de representar una imagen en un formato usual como JPEG, PNG y demás, dados los valores para cada componente de la matriz.

IV. Implementación

IV-A. Programa secuencial

Para el programa secuencial se puede variar solo el tamaño del kernel y el script ejecuta las tres imágenes con tamaños

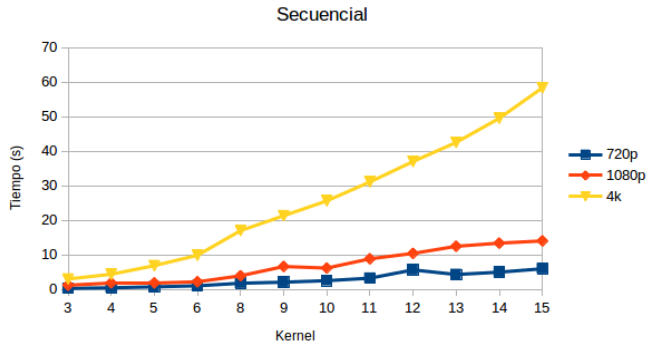


Figura 6: Ejecución programa secuencial.

de kernel $3 \leq k \leq 15$ llevando a los resultados de la figura 6.

Se puede notar una progresión bastante clara en el tiempo del efecto borroso en términos del tamaño de la imagen y en términos del tamaño del kernel.

IV-B. Programa paralelizado en CPU

Para la implementación del algoritmo se usaron dos funciones principales: BoxBlur y CheckBounds en la cual se aplica el BoxBlur. Se hacen pruebas para cada una de las imágenes y tamaños variables de kernel $3 \leq k \leq 15$ tomando el tiempo en segundos para ver como van bajando los tiempos, a veces, si la imagen es muy pequeña o el algoritmo no es tan complejo puede ser que los tiempos no reflejen una mejoría lineal. Los resultados son los siguientes

IV-C. Programa en GPU

Por cuestiones de compilación y de hacer un makefile que estuviera lo suficientemente organizado para que cada una de las librerías respectivas utilizadas se dividio el procesamiento básico del algoritmo

IV-C1. Programa principal: Este programa recopila los resultados de todos los otros programas tomando en primer lugar los parametros de una forma organizada para enviarlos a las respectivas funciones. Este llama a al procesador de OpenCV y llama a las funciones para reservar la memoria respectiva en la GPU así como la llamada a la función que contiene el kernel y el posterior procesamiento que se mencionaba, adicionalmente también llama a las funciones que libera la memoria, la idea de este programa es dar un esquema general de la arquitectura de CUDA mediante su organización con respecto a la integración con OpenCV, y ver las funciones principales que se llaman.

IV-C2. Procesamiento con OpenCV: El archivo *blur-effect.cpp* se encarga de todas las funciones que tienen que ver con OpenCV, aquí el procesamiento es el más largo e importante porque necesitamos convertir el objeto predeterminado de esta librería que es *Mat* a punteros de tipo *uchar4* para que sean mucho más manejables en el kernel de la GPU Y la separación por componentes se haga de una manera más intuitiva, además se hacen los llamados correspondiendentes a los *cudaMalloc* para reservar el espacio de memoria y se

define el tamaño del kernel del que se va a aplicar el efecto. El posprocesamiento de la imagen cuando esta ya ha sido procesada es mucho más sencillo porque solo recibe el puntero *uchar4* y lo vuelve a convertir a un objeto *Mat* para escribirlo en un nuevo archivo, finalmente hay una función que libera la memoria en este archivo

IV-C3. Procesamiento con CUDA: En el archivo *cuda_processing.cu* se hace el procedimiento para separar cada uno de los componentes o canales respectivos, la función global principal que será el kernel del programa tiene como paramétros un arreglo con los pixeles de la imagen, la cantidad de filas y columnas de la matriz original de la imagen, el tamaño del kernel. Cada hilo se encargará de procesar una fila actualizando el valor de cada pixel con el promedio de los colores de los pixeles ubicados en un radio de kernel/2 de acuerdo al balanceo de carga. La siguiente función es simplemente un separador de canales en dónde se toma cada uno de los valores del píxel del valor m que se había utilizado en la función anterior para tener la ubicación única del bloque en las distintas componentes que se necesitan, recombinar los canales como es la siguiente función es su función análoga. Ahora veamos, cada uno de los kernel de la GPU se lanzan para cada una de las submatrices asociadas a cada una de las componentes de color de la imagen, por eso se lanzan tres kernels, se decidió utilizar este acercamiento al problema porque simplificaba la abstracción sin tener que preocuparse por bloques o hilos en tres dimensiones si no, en dos dimensiones como es más natural pensarlo.

V. Resultados

Todas las pruebas realizadas para este ejercicio son para tres imagenes:

1. 720p ⇒ 720 filas y 1280 columnas
2. 1080p ⇒ 1080 filas y 1920 columnas
3. 4K ⇒ 2160 filas y 3840 columnas

V-A. Hilos POSIX

Figuras 7, 8 y 9

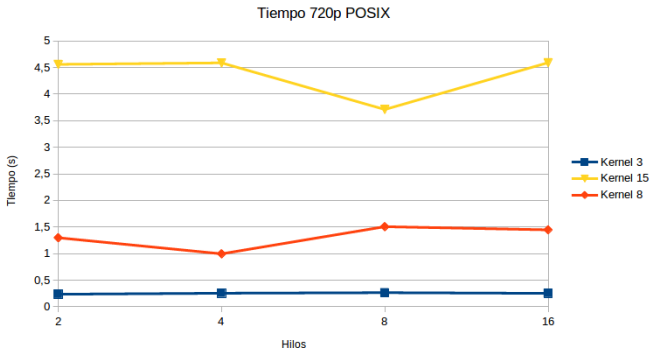


Figura 7: Tiempo para la imagen 720p con un kernel de tamaño 3, 8 y 15

V-B. OpenMP

Figuras 10, 11 y 12

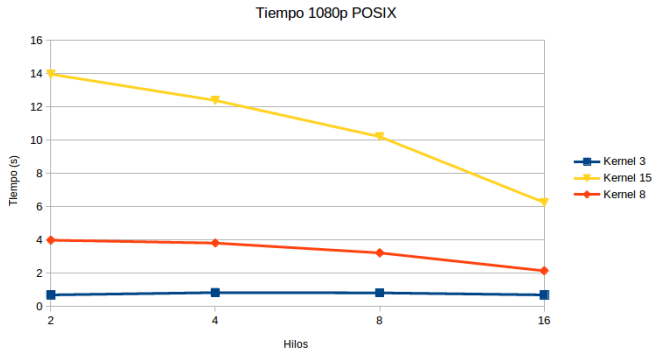


Figura 8: Tiempo para la imagen 1080p con un kernel de tamaño 3, 8 y 15

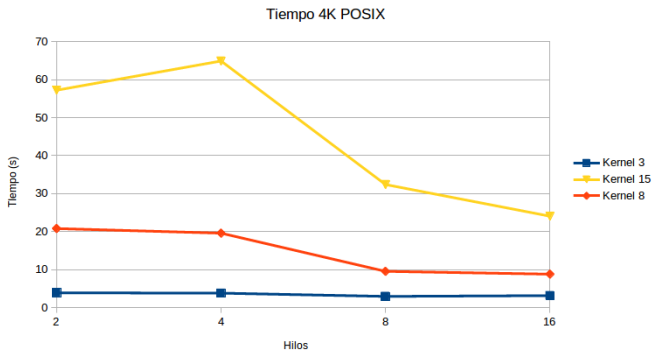


Figura 9: Tiempo para la imagen 4K con un kernel de tamaño 3, 8 y 15

V-C. CUDA

Con kernels de 3, 5, 7, 9, 11, 13 y 15 para cada imagen y a su vez se paralelizó con 1, 2, 4, 8 y 16 hilos. El procedimiento se realizó un total de 10 veces para combinación, donde se tomó el tiempo de cada ejecucion del programa, se obtenia el promedio por combinacion, se llenaron las siguientes tablas con esos datos que posteriormente se gráficaron.

Cuadro I: Tiempo en segundos para aplicar el efecto en imagen de 720px

Hilos	Kernel 3	Kernel 5	Kernel 7	Kernel 9	Kernel 11	Kernel 13	Kernel 15
1	0,389	1,013	1,823	3,273	4,569	6,244	7,807
2	0,269	0,663	1,299	1,818	2,877	3,726	4,492
4	0,253	0,556	1,189	1,82	2,569	3,092	3,857
8	0,252	0,526	1,191	1,812	2,541	3,071	3,86
16	0,251	0,522	1,061	1,731	2,527	3,092	3,841

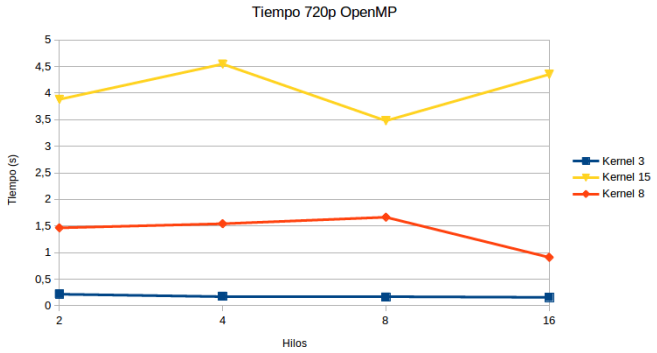


Figura 10: Tiempo para la imagen 720p con un kernel de tamaño 3, 8 y 15

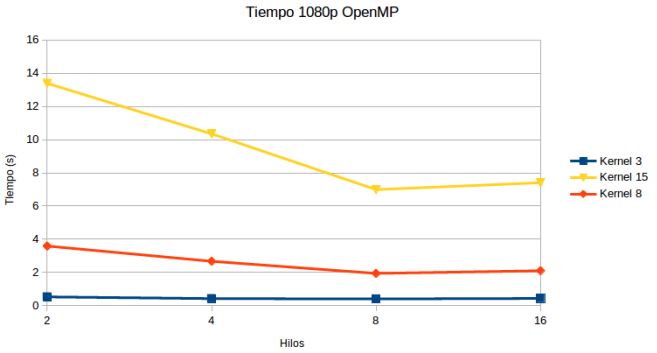


Figura 11: Tiempo para la imagen 1080p con un kernel de tamaño 3, 8 y 15

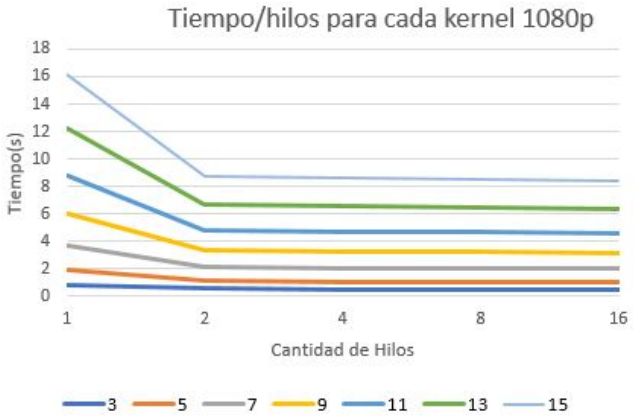


Figura 14: Grafica de tiempos para cada kernel en funcion de la cantidad de hilos.

V-D. *Open MPI*

Figuras 16, 17 y 18

VI. Speed Up.

El Speed-up es una medida de rendimiento mediante la cual se busca aumentar la eficiencia de ejecución de un proceso

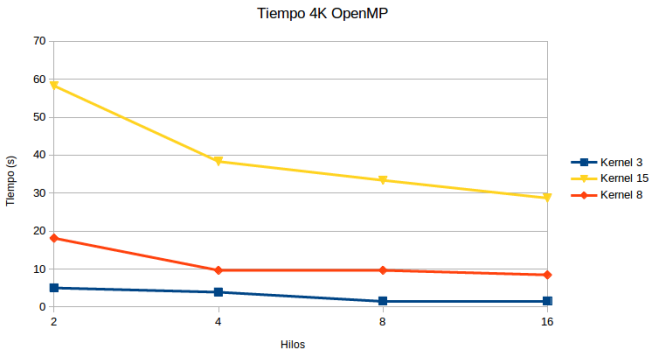
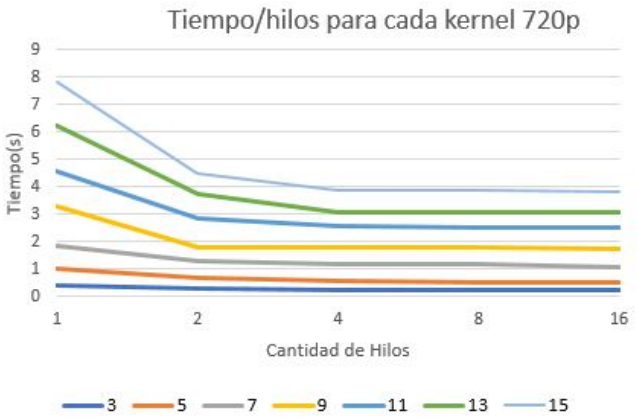


Figura 12: Tiempo para la imagen 4K con un kernel de tamaño 3, 8 y 15

Figura 13: Grafica de tiempos para cada kernel en funcion de la cantidad de hilos.



ejecutado de dos formas diferentes, en este caso, éstas serán de forma secuencial y para con diferentes cantidades de hilos, este valor es calculado con la siguiente función tomando los datos descritos en las anteriores tablas.

VI-A. *CPU*

VI-A1. Hilos POSIX: Figuras 19, 20 y 21

VI-A2. OpenMP: Figuras 22, 23 y 24

VI-A3. Open MPI: Figuras 25, 26 y 27

VI-B. *CUDA*

El Speedup se puede ver en las tablas junto con las gráficas.

Cuadro II: Tiempo en segundos para aplicar el efecto en imagen de 1080px

Hilos	Kernel 3	Kernel 5	Kernel 7	Kernel 9	Kernel 11	Kernel 13	Kernel 15
1	0,389	1,013	1,823	3,273	4,569	6,244	7,807
2	0,269	0,663	1,299	1,818	2,877	3,726	4,492
4	0,253	0,556	1,189	1,82	2,569	3,092	3,857
8	0,252	0,526	1,191	1,812	2,541	3,071	3,86
16	0,251	0,522	1,061	1,731	2,527	3,092	3,841

Cuadro III: Tiempo en segundos para aplicar el efecto en imagen de 4k

Hilos	Kernel 3	Kernel 5	Kernel 7	Kernel 9	Kernel 11	Kernel 13	Kernel 15
1	0,802	1,908	3,724	5,96	8,814	12,255	16,206
2	0,604	1,165	2,091	3,295	4,83	6,672	8,741
4	0,481	1,06	2,02	3,217	4,713	6,527	8,582
8	0,46	1,054	1,994	3,197	4,628	6,495	8,549
16	0,45	1,035	1,956	3,128	4,583	6,34	8,432

Figura 28: Gráfica del Speedup para cada imágen procesada con un kernel 3.

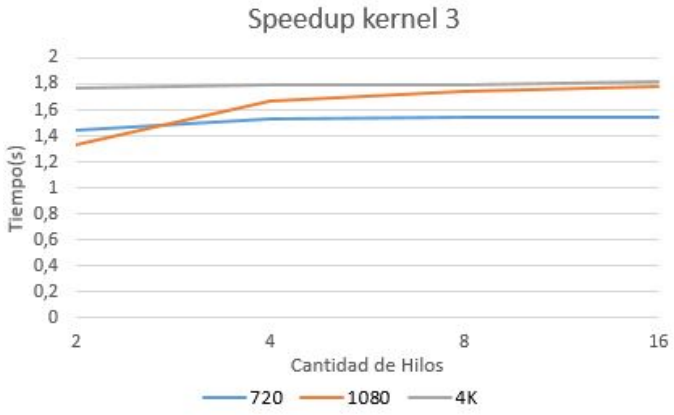


Figura 29: Gráfica del Speedup para cada imágen procesada con un kernel 9.

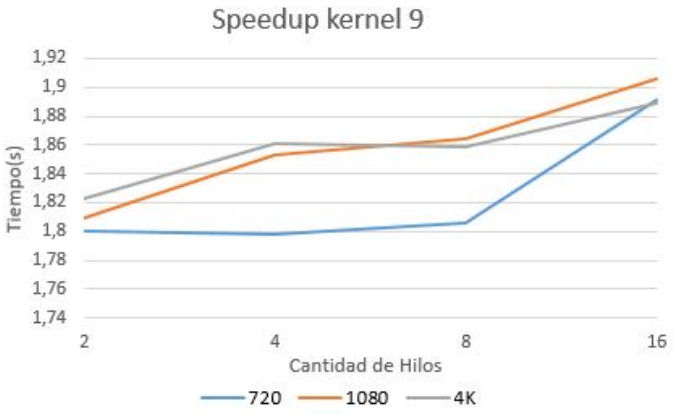
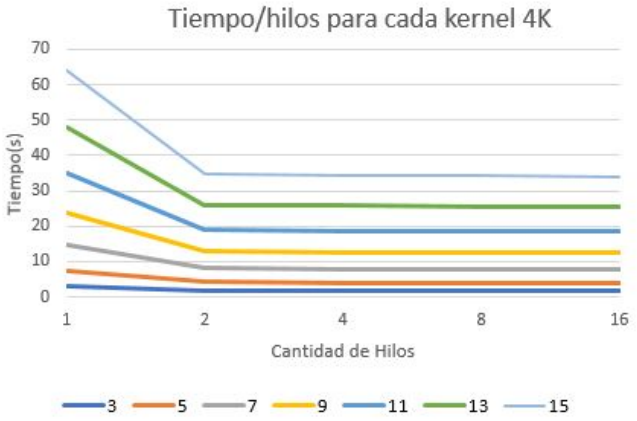


Figura 15: Grafica de tiempos para cada kernel en funcion de la cantidad de hilos.



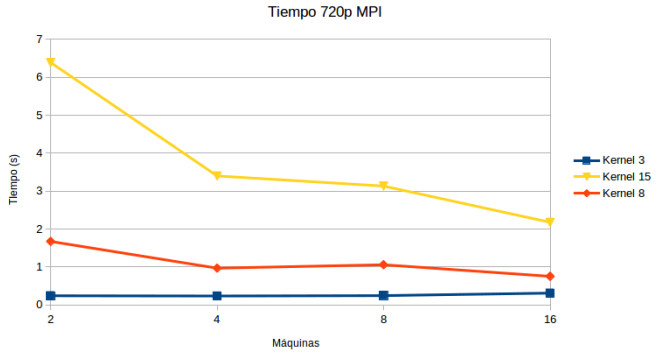
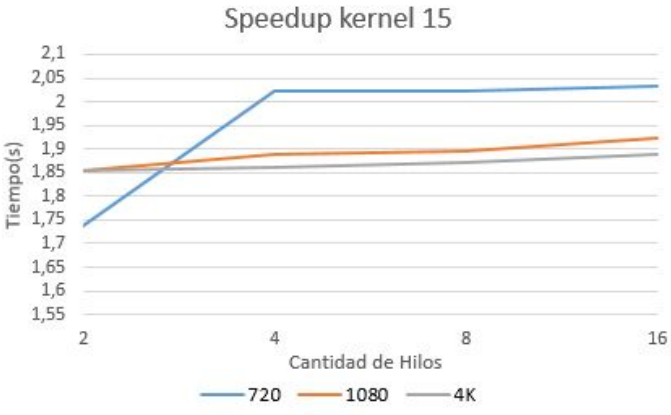


Figura 16: Tiempo para la imagen 720p con un kernel de tamaño 3, 8 y 15

Figura 30: Gráfica del Speedup para cada imágen procesada con un kernel 15.



VII. Experimentos y resultados CUDA

Conclusiones.

1. El tiempo de ejecución aumenta a medida que se aumenta el tamaño del kernel, sin embargo para un mismo tamaño

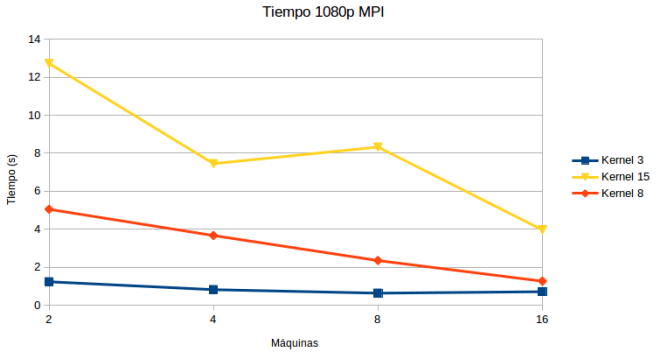


Figura 17: Tiempo para la imagen 1080p con un kernel de tamaño 3, 8 y 15

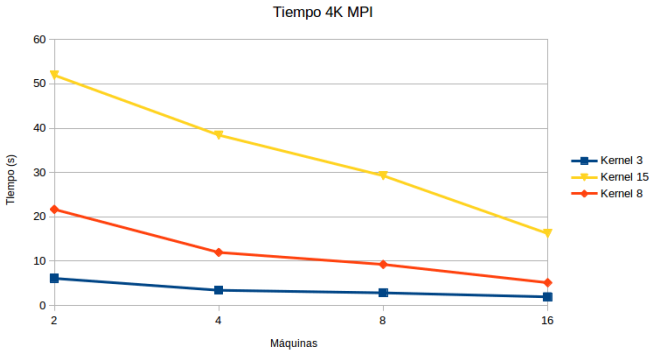


Figura 18: Tiempo para la imagen 4K con un kernel de tamaño 3, 8 y 15

de kernel, el tiempo disminuye a medida que se aumenta el número de hilos.

2. El tiempo de ejecución del procedimiento mejora o empeora con diferentes cantidades de hilos, depende totalmente de los recursos que se tengan a disposición.
3. La capacidad de medir si el tiempo de ejecucion del procedimiento mejora o empeora con diferentes cantidades de hilos, depende totalmente de los recursos que se tengan a disposición.
4. En cuanto a las gráficas de speedup se puede observar que dados los limitados recusos computacionales con los que se cuentan, no se observa una gran diferencia conforme se incrementan la cantidad de hilos, debido a que al paralelizar solo encola los nuevos procesos.
5. La paralelización aplicada a problemas de procesamiento de imagenes aumenta el Speed up considerablemente porque en general se están tratando con problemas altamente paralelizables.

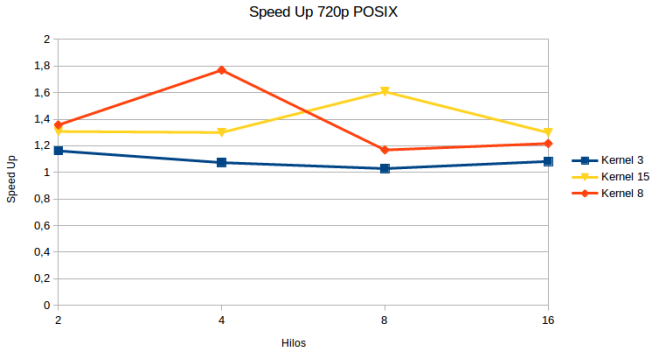


Figura 19: Tiempo para la imagen 720p con un kernel de tamaño 3, 8 y 15

Figura 20: Tiempo para la imagen 1080p con un kernel de tamaño 3, 8 y 15

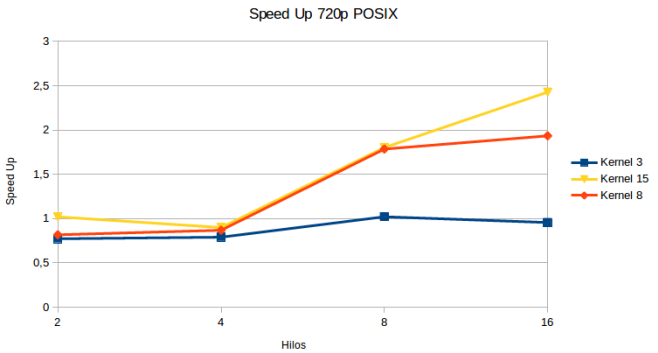


Figura 21: Tiempo para la imagen 4k con un kernel de tamaño 3, 8 y 15

Cuadro IV: Resultado de realizar el Speedup con un kernel de tamaño 3

Hilos	720	1080	4k
2	1,45	1,33	1,77
4	1,54	1,67	1,79
8	1,54	1,74	1,80
16	1,55	1,78	1,82

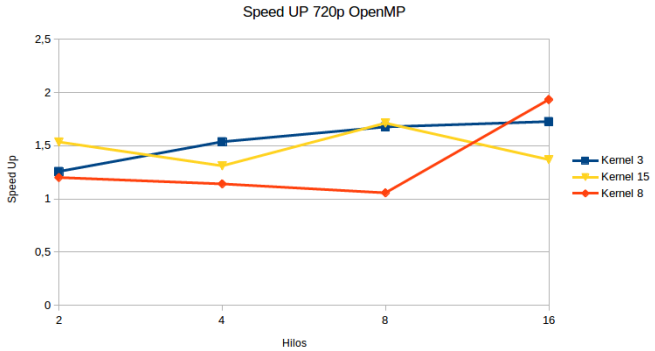


Figura 22: Tiempo para la imagen 720p con un kernel de tamaño 3, 8 y 15

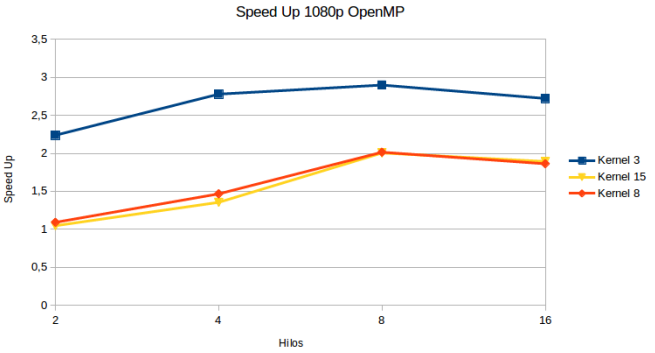


Figura 23: Tiempo para la imagen 1080p con un kernel de tamaño 3, 8 y 15

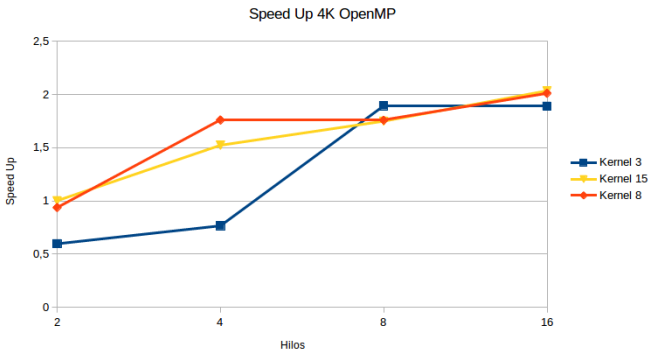


Figura 24: Tiempo para la imagen 4K con un kernel de tamaño 3, 8 y 15

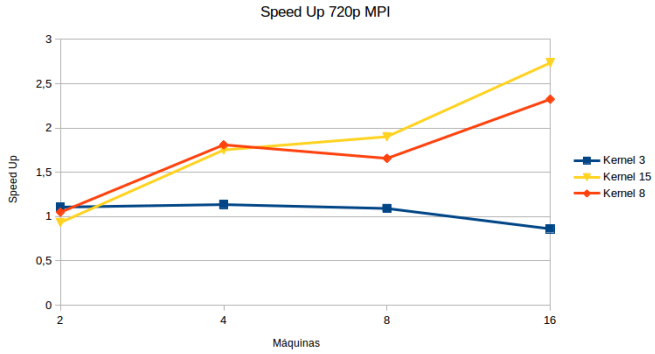


Figura 25: Tiempo para la imagen 720p con un kernel de tamaño 3, 8 y 15

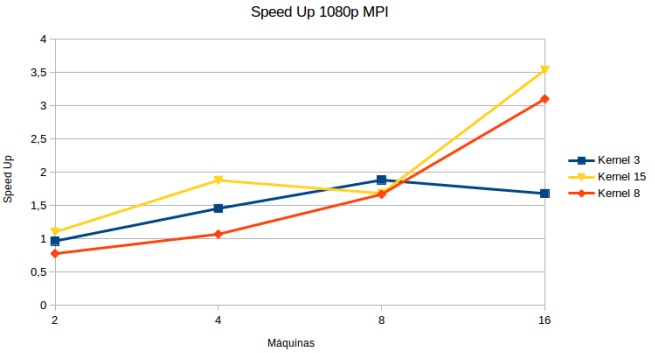


Figura 26: Tiempo para la imagen 1080p con un kernel de tamaño 3, 8 y 15

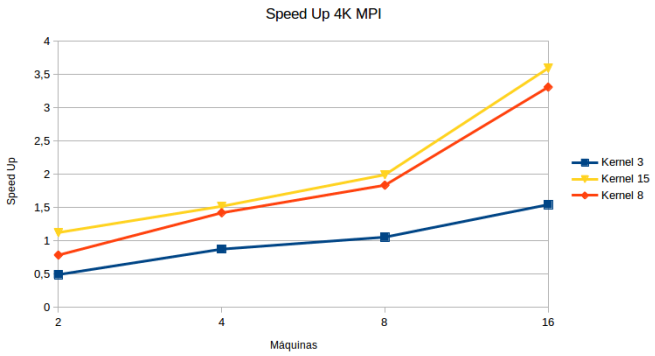


Figura 27: Tiempo para la imagen 4K con un kernel de tamaño 3, 8 y 15

Cuadro V: Resultado de realizar el Speedup con un kernel de tamaño 9

Hilos	720	1080	4k
2	1,80	1,81	1,82
4	1,80	1,85	1,86
8	1,81	1,86	1,86
16	1,89	1,91	1,89

Cuadro VI: Resultado de realizar el Speedup con un kernel de tamaño 15

Hilos	720	1080	4k
2	1,74	1,85	1,86
4	2,02	1,89	1,86
8	2,02	1,90	1,87
16	2,03	1,92	1,89