Computación paralela y distribuída Práctica 3

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, OpenMP y NVIDIA CUDA, 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 y las ventajas del uso de masiva paralelización en GPUs.**

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 la actualidad las aplicaciones de tratamiento gráfico, inclusive de tareas con intensa cantidad de computación, hacen uso de tarjetas gráficas o GPUs, las cuales cuentas con miles de núcleos no tan potentes como los de una CPU, pero lo suficientemente potentes como para hacer operaciones lógico aritméticas de forma masiva, y con la habilidad de tener cambios de contexto inmediatos. La masiva paralelización en GPUs tiene su punta de desarrollo en NVIDIA con su lenguaje de desarrollo con aplicación abierta CUDA, que hace posible programar sobre GPU sin que sea una tarea de gráficos.

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.

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

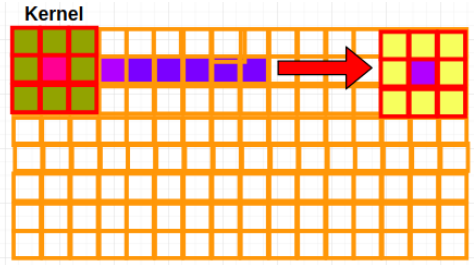

Figura 1: Recorrido del kernel en la matriz imagen haciendo convolución

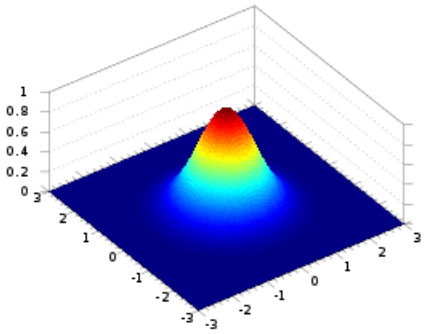

Figura 2: Función gaussiana 3D

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:

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

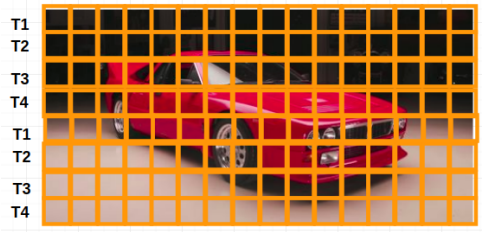

Figura 3: Asignación cíclica de hilos a cada fila

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.

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 aśi 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.

Para el caso de la aplicación de OpenMP como método de paralelización, se implementó en la fase de recorrido de la matriz imagen, de forma que se mantiene la forma de cyclic data. El recorrido es entonces dividio en el número de hilos dados, usando la directiva de parallel for, poniendo privadas para cada hilos las variables i e j que son la iteradoras en el recorrido de filas y columnas de la matriz imagen respectivamente. El ciclo for es entonces partido en pedazos que van haciendo hasta cierto punto el ciclo for, actualizando la imagen producto, para tener al final de ciclo, una actualización completa de la imagen producto, obteniendo el filtro esperado de blur.

II-A. Paralelización en GPU NVIDIA con CUDA

La estrategia usada en el caso de paralelización masiva en GPU NVIDIA, fue la de **Blockwise**, de manera que cada CUDA core, o thread en el lenguaje, aplicara el desenfoque blur a un grupo de pixeles, dados en forma de los 3 canales de color. Se obtiene entonces a la final que cada thread/CUDA core, se le asigna un grupo de pixeles a modo de bloque, sin tener ciclos, por lo que se lanzan todos los pixeles al mismo tiempo a la GPU, y la forma de dividir las tareas es tomando la cantidad total de pixeles de la imagen, dividido el doble de la cantidad de threads/cores CUDA que posee la GPU, previa una escaneada de esa cantidad de hilos por bloque y bloques por grilla, como lo recomienda NVIDIA, teniendo presente que los cambios de contexto en la GPU son inmediatos, por lo que pasar de ejecutar la multiplicación del efecto blur a un pixel y pasar al otro pixel del bloque asignado, se hace muy rápido.

De acuerdo a lo anterior, cada CUDA core recibe un **Block** de pixeles, para que le sea aplicado el efecto blur, teniendo una paralelización masiva, con todos los CUDA cores siendo usados y mantenidos en uso de forma intensiva aprovechando los inmediatos cambios de contexto que ofrece NVIDIA.

III. Experimentos y resultados.

Las pruebas se realizaron en un portatil Asus con AMD A10 de 4 núcleos a 2.5Ghz, 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.

Para el caso de las pruebas de paralelización en GPU NVIDIA con CUDA, se usó la herramienta online de Google **Colab** con la cual se puede acceder de forma sencilla a una gran GPU, como lo es la NVIDIA Tesla T4, la cual cuenta con 2560 CUDA cores, a 1.6GHz de frecuencia máxima por core, 15GB de memoria global, configurados como 128 cores por bloque, y 20 bloques por grilla, usando así siguiendo con las especificaciones y recomendaciones de NVIDIA, de lanzar el doble de hilos que cores, 5120 hilos lanzados en total.

Threads	Time (ms)
1	2147,72
2	1232,27
4	626
8	887
16	899

Figura 4: Tabla Tiempo vs Hilos imagen HD POSIX

Threads	Speed up
1	0
2	1,742897255
4	3,432392392
8	2,421327597
16	2,388566306

Figura 5: Tabla Speed up vs Hilos imagen HD POSIX

Dado que colab tiene restricciones, no se pudo hacer uso de OpenCV para el tratamiento de las imágenes, motivo por el cual se usó Libpng, la cual es una librería nativa de C++, capaz de tomar los 3 canales RGB de un pixel, y hacer transformaciones en imágenes únicamente de formato PNG. Colab debe ser configurado para instalar CUDA, y poder aśi tener la capacidad de correr programas en extensión cu. La mejor forma de sincronizar el código en desarrollo de la paralelización, fue la de ir haciendo un pull a un repositorio en github, con lo cual se podía actualizar el estatus del código a correr en colab de forma sencilla y dinámica.

A continuación se muestran los resultados obtenidos para tamaños de kernel de 7 con POSIX, OpenMP y CUDA: (para el caso de CUDA, se tomaron valore siempre con 5120 hilos, pero cambiando de 3 hasta 311 el tamaño del kernel filtro)

Los tiempos están medidos en milisegundos (ms), el tamaño del kernel por la cantidad de cuadros por lado en la matrix filtro, y el speed up como la razón entre el tiempo secuencial y el tiempo paralelizado en un ejemplo del mismo tamaño de kernel.

Conclusiones.

Se puede apreciar claramente la Ley de Amdahl, en cuanto a que lanzar una mayor cantidad de hilos, no significa que se tenga un mejor desempeño, debido a las limitantes de

Threads	Time (ms)
1	5636,51
2	3246,53
4	1644,73
8	2341,53
16	2356,22

Figura 6: Tabla Tiempo vs Hilos imagen FullHD POSIX

Threads	Speed up
1	0
2	1,736164459
4	1,736164459
8	2,407191025
16	2,392183243

Figura 7: Tabla Speed up vs Hilos imagen FullHD POSIX

Threads	Time (ms)
1	22301,5
2	11400,3
4	6473,65
8	9302
16	992,06

Figura 8: Tabla Tiempo vs Hilos imagen 4k POSIX

Threads	Speed up
1	0
2	1,95622045
4	3,44496536
8	2,397495162
16	22,47999113

Figura 9: Tabla Speed up vs Hilos imagen 4k POSIX

hardware, especialmente a la cantidad de cores disponibles en el dispositivo, por lo cual, se debe tener una clara idea de la capacidad en número de cores del procesador, para poder sacar todo el potencial en el mismo, sea usando POSIX u OpenMP como en el presente trabajo.

Adicionalmente podemos ver que a mayor cantidad de hilos a partir del momento en el que la cantidad de hilos es igual a la cantidad de cores, el rendimiento empieza a decrecer, debido a que además de tener que procesar los elementos de código del programa que está siendo ejecutado de manera paralela, el sistema operativo debe manejar los demás hilos que exceden la capacidad tope del dispositivo, haciendo que en vez de acelerar la ejecución, lo haga más lento.

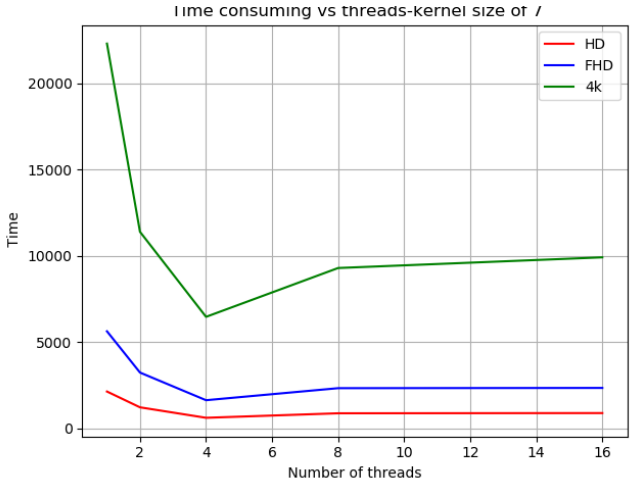

Figura 10: Tiempo vs hilos POSIX HD FHD y 4k

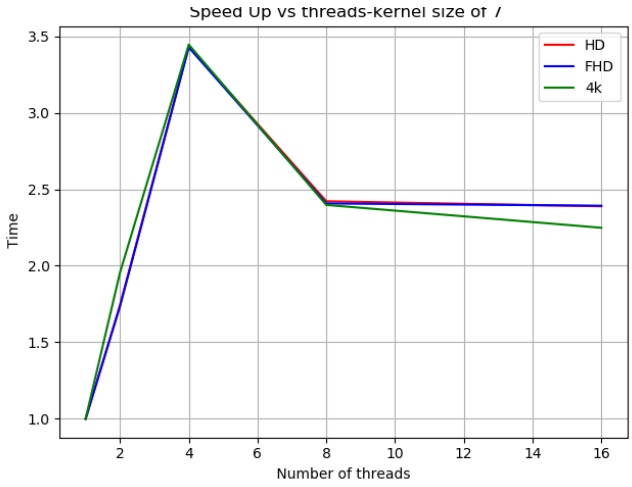

Figura 11: Speed up vs hilos POSIX HD FHD y 4k

Threads	Time (ms)
1	2126,68
2	1074,39
4	637
8	740,26
16	587

Figura 12: Tabla Speed up vs Hilos OpenMP imagen HD

Threads	Speed up
1	0
2	1,979430188
4	3,336633337
8	2,872882501
16	3,622606284

Figura 13: Tabla Speed up vs Hilos OpenMP imagen HD

Threads	Time (ms)
1	5562,64
2	2829,22
4	1637
8	1817,68
16	1816,44

Figura 14: Tabla Speed up vs Hilos OpenMP imagen FullHD

Threads	Speed up
1	0
2	1,966139077
4	3,39806964
8	3,060296642
16	3,062385766

Figura 15: Tabla Speed up vs Hilos OpenMP imagen FullHD

Threads	Time (ms)
1	22295,1
2	11231,4
4	7955,09
8	7961,69
16	8028,64

Figura 16: Tabla Speed up vs Hilos OpenMP imagen 4k

Threads	Speed up
1	0
2	1,985068647
4	2,802620712
8	2,800297424
16	2,776946033

Figura 17: Tabla Speed up vs Hilos OpenMP imagen FullHD

COmo comparacion de los dos métodos usados en la paralelización, OpenMP tiene un muy ligero mejor rendimiento en algunos casos, como lo es comparando el caso de la imagen en FUllHD donde se ve un speed up superior a POSIX, pero en 4k, POSIX es un poco más rápido, razón por la cual podemos decir que los dos métodos son comparativamente muy similares, en su performance general al momento de acelear tareas.

Sin embargo, si vemos el área de facilidad de implementación, OpenMP es mucho más sencillo, debido a que usa una única directiva para empezara a hacer el paralelismo , diferente a POSIX, en donde es necesario declarar, asignar y cerrar los hilos de manera manual y de parte del programador.

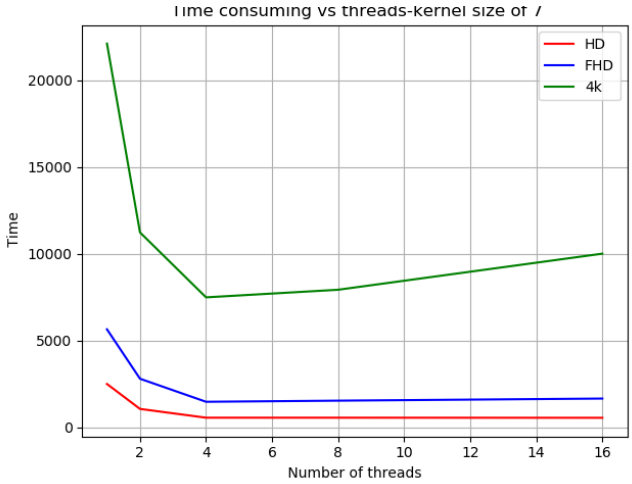

Figura 18: Tiempo vs hilos OpenMP HD FHD y 4k

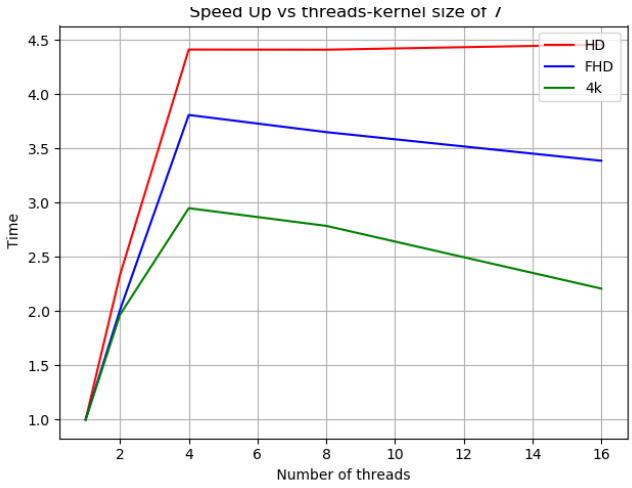

Figura 19: Speed up vs hilos OpenMP HD FHD y 4k

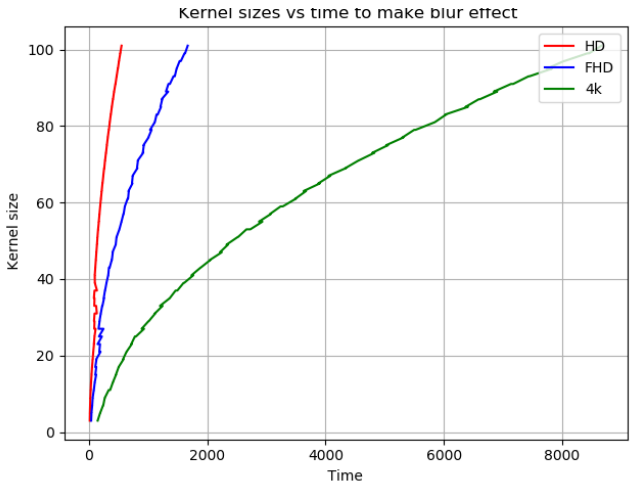

Figura 20: Tamaño kernel vs Tiempo GPU NVIDIA Tesla T4 (hasta 100 tamaño kernel, 5120 hilos)

Para el caso de CUDA, se ve claramente que las velocidades

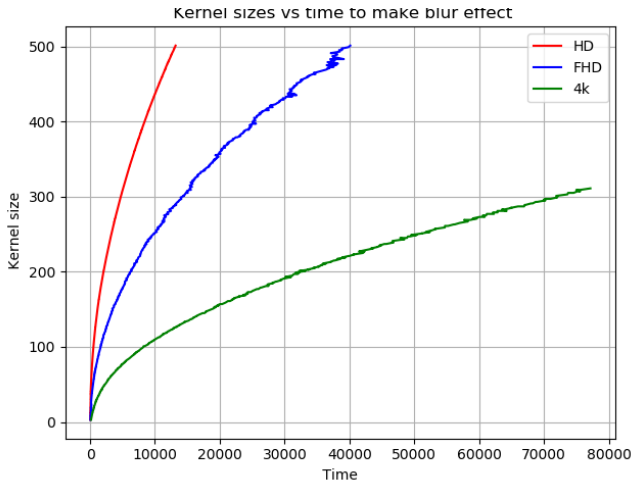

Figura 21: Tamaño kernel vs Tiempo GPU NVIDIA Tesla T4 (hasta 311 y 500 tamaño kernel, 5120 hilos)

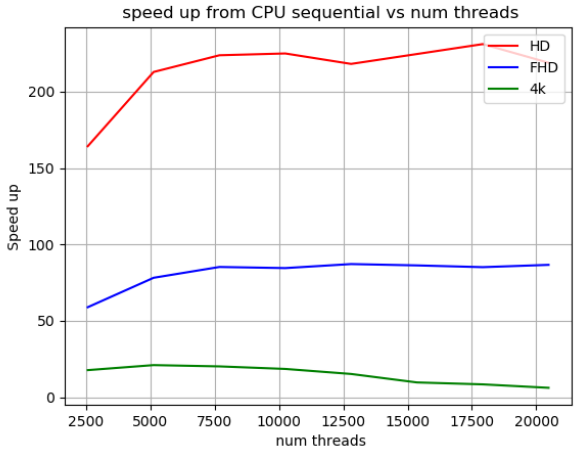

Figura 22: Speed up vs Tiempo GPU NVIDIA Tesla T4 (desde tiempo secuencial CPU kernel size de 15)

en GPU son asombrosas, están muy lejos de los tiempos de CPU, teniendo increíbles resultados, donde en CPU el mejor tiempo es exponencialmente mayor al tiempo promedio en GPU, lo que demuestra que usar GPUs para paralelización masiva tiene una ventaja realmente superior, vale la pena usar GPUs para paralelización, los miles de cores con los que cuentan las GPU le dan una ventaja enorme a los algoritmos de procesamiento de imágenes como en el trabajo presente.

La posibilidad de usar miles de cores sencillos al mismo tiempo, es realmente útil, ahorrando tiempo de forma exponencial, haciendo que lo engorroso de programar en CUDA para GPUs, valga la pena con los resultados y las aceleraciones respecto a CPU que se obtuvieron.

Como conclusión final entonces, paralelizar con GPUs en CUDA es mucho más rápido y efectivo que usar CPUs, los tiempos demuestran de forma certera lo potentes que son las GPUs para tareas de paralelización masiva, y que además de tareas gráficas, en la actualidad son usadas para procesamiento de datos e investigación en machine learning debido a ésta grandísima ventaja de ser superior a las CPU.