Prácticas

Jonathan Carrillo[#1], Edwin Mahecha[#2], Fabio Tovar[#3]

[#]*Departamento de Sistemas e Industrial, Universidad Nacional de Colombia*

[1]joacarrilloco@unal.edu.co

[2]ermahechap@unal.edu.co.edu

[3]fstovarr@unal.edu.co

I. Introducción

La necesidad de realizar tareas y cálculos que requieren de una gran cantidad de tiempo, ha estado presente desde los inicios de la computación. Como solución a este problema, han surgido diversas propuestas desde el nivel del hardware y del software, y algunas de ellas han sido ampliamente desarrolladas actualmente, permitiendo a científicos, militares, diseñadores, estadistas, entre otros, apoyarse de las herramientas tecnológicas para hacer su trabajo más preciso y eficiente.

En el presente documento encontrará información acerca del uso de hilos POSIX para paralelizar una tarea clásica de interés computacional, como una de las diferentes técnicas para realizar cálculos de manera óptima, aprovechando las capacidades del hardware y las facilidades del sistema operativo.

De igual manera, se hará una comparativa con otras dos alternativas para desarrollar la misma tarea; OpenMP y CUDA. Encontrará también en este documento datos y gráficos mostrando el rendimiento de las diversas alternativas aquí presentadas, con algunas conclusiones al respecto.

II. marco teórico

A. POSIX & POSIX THREADS

POSIX (Portable Operating System Interface) define una interfaz estándar de sistema operativo y un entorno incluyendo un intérprete de comandos (shell) y utilidades (como por ejemplo APIs) para dar soporte a portabilidad de aplicaciones en el ámbito del código fuente.

En otras palabras, POSIX es un estándar que permite compatibilidad entre sistemas operativos, principalmente aquellos basados en UNIX, aunque Windows y otros sistemas operativos ofrecen soporte para estas. Entre las APIs y librerías que ofrece POSIX encontramos una en particular que es de nuestro interés, la cual es el API de Threads.

Los hilos de POSIX, más conocidos como POSIX Threads son el API estándar para C/C++ para el manejo de hilos. Esta permite la creación de procesos en paralelo tanto en sistemas multicore como multiprocesador, aunque sistemas uniprocesador también pueden sacar provecho de esta como por ejemplo en tareas de I/O.

Algunas de las características más importantes del API de POSIX Threads son:

- Inclusión de operaciones que permiten la creación, terminación, sincronización, scheduling, manejo de datos e interacción entre procesos/hilos.
- Un hilo no mantiene una lista de los hilos creados o que hilo es su padre. Esto es de suma importancia para el desarrollo de la práctica, ya que es importante mantener referencia a las threads creadas o de lo contrario podrían quedar tareas no deseadas en ejecución.
- Todas los hilos dentro de un mismo proceso comparten el mismo espacio de direcciones, UID, GID, directorio actual de trabajo, etc.
- Cada thread tiene un único ID, conjunto de registros, stack para variables locales, etc.

B. OPENMP

OpenMp es una API que facilita escribir aplicaciones que hagan uso de multithreading, más concretamente, es un conjunto de directivas del compilador y de librerías para la programación paralela.

OpenMp, diseñado especialmente para los lenguajes de C, C++ y Fortran, facilita de manera significativa la escritura de programas multihilos en dichos lenguajes, como vimos en el caso de POSIX, la escritura de algunos programas que hagan cómputos de forma paralela puede ser engorrosa, es por eso que OpenMP nos facilitará dicha escritura, también es importante aclarar que se hará una comparativa de los resultados obtenidos con ambos métodos.

Es importante resaltar que OpenMp trabaja bajo un modelo de memoria compartida y bajo un modelo de fork-join, éste último es un paradigma que busca dividir las tareas más grandes haciendo fork, y luego unir los resultados al hacer join.

OpenMp busca ser global e igualmente, de cierta manera, estandarizar la programación paralela, ésto se atribuye a la

gran variedad de plataformas a las que llega, es decir, tiene una alta portabilidad, algunos de los sistemas o compiladores en más importantes en los que podemos encontrarlo son: ARM, GCC, Oracle, IBM, Intel.

C. *CUDA*

Usualmente se piensa en GPUs (Graphic Processing Units) como unidades especializadas que permiten renderizar gráficos 3D de alta resolución en tiempo real, pero pueden tener más aplicaciones debido a que son sistemas multicore altamente paralelizables, lo cual permite manejar grandes bloques de datos de forma eficiente. Para ello se han desarrollado varias plataformas que permitan hacer uso de estos cores para aplicaciones de propósito general como lo es OpenCL, OpenACC, etc.

CUDA (Compute Unified Device Architecture) es una plataforma y API de programación paralela desarrollada por NVIDIA que permite hacer uso de GPUs NVIDIA para aplicaciones de propósito general. La API está diseñada para lenguajes de programación como C, C++ y Fortran.

De forma simplificada, una GPU NVIDIA tiene la siguiente arquitectura multiprocesador, la cual nos facilitará posteriormente el diseño de nuestros programas:

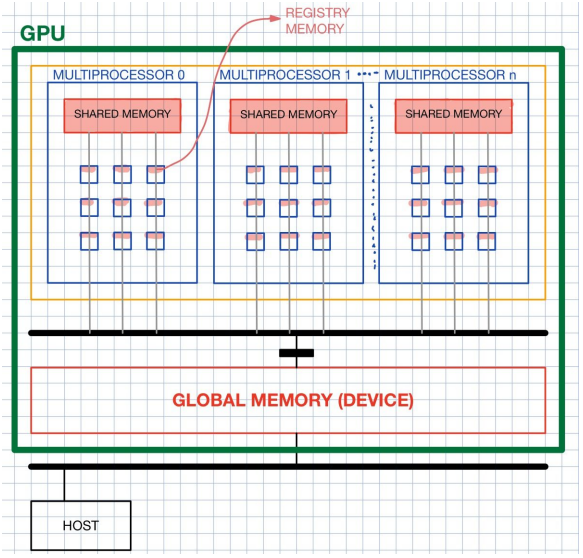


Fig. 1 Diagrama Simplificado GPU NVIDIA

En un principio tenemos una memoria global (Global Memory) para nuestro dispositivo. Esta memoria en un principio es en donde cargamos los datos que necesitamos procesar desde nuestro HOST.

Posteriormente podemos lanzar hilos, los cuales son ejecutados asignados a un core. Cabe notar que cada hilo posee una pequeña memoria (Registry Memory), por lo que hay que ser cuidadosos con las variables que se declaren y los datos que se copien a estas, debido a que resulta fácil exceder su capacidad. Para conjuntos más grandes de datos, se puede hacer uso de la memoria compartida (Shared Memory), que es una memoria que puede ser accedida por cada uno de los cores de un multiprocesador, aunque hay algo importante a mencionar y es que es posible lanzar más hilos que la cantidad de cores disponibles en un multiprocesador, y debido a esto la memoria compartida no se distribuye entre el número de cores del multiprocesador, si no entre el número de hilos que sean lanzados en el multiprocesador. En otras palabras, la memoria compartida se asigna a cada hilo, no a cada core.

D. *MÉTODOS DE DISTRIBUCIÓN DE LOS HILOS*

Cuando se está trabajando de manera paralela, es necesario que el problema que se desea tratar pueda dividirse en subtareas que no deban ser secuenciales para obtener un resultado correcto. Esta posibilidad de división genera una nueva inquietud, ¿cuál es la manera más conveniente de realizarla?

A continuación se presentan tres métodos que abordan estos problemas de una diferente manera.

A. *Distribución por bloques*

Este método consiste en repartir la carga de trabajo del problema de maneras iguales en p hilos, por lo tanto si se tienen m datos, cada uno de los hilos deberá encargarse de procesar un *chunk* de $c = \lceil \frac{m}{p} \rceil$ datos consecuentes.

thread 0				thread 1				⋯	thread $p-1$			
0	1	2	3	4	5	6	7	⋯	$m-4$	$m-3$	$m-2$	$m-1$

Fig. 2 Diagrama de distribución por bloques

B. *Distribución por ciclos.*

La distribución por ciclos consiste en que cada hilo procese una mínima unidad de datos y al finalizar su procesamiento, continúe con el la siguiente unidad que se encuentra p posiciones más adelante. Esto se debe realizar hasta que los hilos procesen todos los datos.

threads	0	1	2	⋯	$p-1$	0	1	2	⋯	$p-1$	⋯
tasks	0	1	2	⋯	$p-1$	p	$p+1$	$p+2$	⋯	$2p-1$	⋯

Fig. 3 Diagrama distribución por ciclos

C. *Distribución bloques-ciclos*

Por último, tenemos un método que combina las dos estrategias anteriormente descritas. Consiste en dividir los m datos en c *chunks* de tamaño conveniente. Cada hilo procesará un *chunk* a la vez, y al finalizar, pasará al siguiente, el cual se encuentra *c*p* posiciones más adelante.

Un tamaño c muy pequeño, ayudará a procesar de mejor manera tareas desbalanceadas, sin embargo tiene que acceder más veces a la memoria, lo cual puede disminuir dramáticamente el rendimiento del programa. Por otro lado, un tamaño c grande es más amigable para el caché, sin embargo los hilos podrían procesar trabajos de manera muy desbalanceada.

Es conveniente entonces tomar tamaños de c que completen el caché de cada hilo, por ejemplo $c=16$ para tipos de dato de 32 bits, y $c=8$ para tipos de dato de 64 bits.

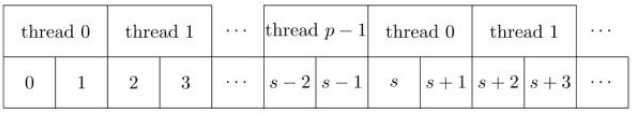


Fig. 4 Diagrama distribución bloques-ciclos

Es posible consultar el tamaño del cache line en el siguiente directorio en algunos dispositivos Linux con un kernel medianamente actualizado
/sys/devices/system/cpu/cpu0/cache/index{n}/coherency_line_size

E. CONVOLUCIONES

Las aplicaciones posibles de las convoluciones son extensas, llegan a ser tan extensas que se escapan del alcance de este documento, algunos de los campos y áreas más notorios en las que podemos encontrarlas son el procesamiento de imágenes y señales, circuitos eléctricos, comunicaciones, probabilidad, estadística, etc.

La verdadera definición de convolución está dada como un productor de dos funciones des los reales en los complejos $f(x),\ g(x)$ y para un valor real u como sigue:

$$\int_a^b f(x) * g(x+u)du$$

Los límites de integración pueden ser infinitos o incluso funciones de x. Pero dada la naturaleza del trabajo aquí presentado, vamos a definir la convolución discreta de forma análoga como

$$\sum_{n=-\infty}^{\infty} x_n * y_{n\pm i}$$

Donde x,y son dos secuencias reales o complejas, además $i \in Z$, note que para dos secuencias finitas x,y con x de tamaño $N1$ y y de tamaño $N2$, la expresión se reduce a la suma finita

$$\sum_{n=0}^{i} x_n * y_{n\pm i},\ \ i = 0, 1, ...N1 - 1$$

Por tanto podemos asociar la convolución de esas dos secuencias finitas a una nueva secuencia de tamaño $N1 + N2 - 1$. Además, podemos detallar que la operación de convolución para este caso, vista como una nueva secuencia, tiene un especial parecido a la multiplicación matricial.

F. GAUSSIAN BLUR

Para el procesamiento de imágenes, las convoluciones pueden ayudarnos a general un efecto de tipo Blur, para esto podemos calcular, con base en una imagen original, una función para cada canal de dicha imagen de forma independiente.

Naturalmente, queremos que los píxeles más cercanos a la posición que estamos calculando, tengan un mayor peso a la hora de realizar el cómputo, es por eso que, si consideramos una matriz cuadrada de tamaño $2n+1$, podemos ubicar la posición que estamos calculando en el centro de la matriz.

Ahora, como el objetivo es realizar una convolución, para cada píxel estaremos calculando un promedio ponderado de los píxeles adyacentes, y es por eso que una distribución normal es útil en este punto, para asignar los pesos que va a tener cada vecino de acuerdo a la distancia al punto en cuestión, a continuación vemos un ejemplo de una matriz de convolución, mejor conocido como Kernel de Gauss, para realizar dicha operación

$$\frac{1}{256} * \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

Fig. 5 Kernel gaussiano de tamaño 5x5

Algunas propiedades de dicha matriz son inmediatas, pues entre otras cosas, para centrar el objetivo, es necesario que el tamaño sea impar, igualmente podemos ver que la matriz está asociada a una distribución de probabilidad normal, y por tanto la suma de las probabilidades es 1. Igualmente podemos observar que para elementos a iguales distancias (Manhattan en este caso) les corresponden iguales ponderados.

Es posible crear un kernel gaussiano 2D de cualquier tamaño que se desee mediante la fórmula 1, la cual tiene como parámetros la posición x y y del elemento a calcular en el kernel, y σ que determina el ancho del filtro gausiano.

$$G_{2D}(x, y; \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Fórmula. 1 Función generadora de kernel gaussianos en dos dimensiones

III. DISEÑO

El problema propuesto fue la implementación de un filtro *blur* o de desenfoque para imágenes a color. Para dar solución al mismo, realizamos una solución secuencial, la que posteriormente usamos para realizar la paralelización de los cálculos mediante cada uno de los tres métodos de distribución descritos en la primera sección. A continuación describiremos brevemente el la manera en la cual nos aproximamos para la implementación de cada uno de los mismos.

A. Secuencial

El problema se dividió en tres partes centrales, el cálculo del kernel gaussiano para aplicar el filtro, la apertura, lectura y escritura de archivos de imágenes por píxeles, y por último, la aplicación del filtro a la imagen.

Para el cálculo del kernel, se aplicó la fórmula descrita en la sección A.A.2, cuyos parámetros son el sigma característico de la matriz y su tamaño.

Luego, para la lectura y escritura, decidimos usar una librería liviana y pública (licencia MIT) para C y C++ llamada STB, esta permite realizar las operaciones de lectura y escritura de imágenes de manera sencilla en los formatos más comunes.

Por último, la aplicación del filtro se hizo mediante una función que recibe como parámetros principales, las dimensiones de la imagen, un vector de entrada y uno de salida. Por cada uno de los representados en el vector de entrada se calcula el filtro como corresponde y se guarda el nuevo valor en un arreglo de salida. Al final, con ayuda de la librería, se genera la imagen a partir de este arreglo.

B. Particionado Blockwise

Usando como base el modelo secuencial descrito anteriormente, se creó una estructura en C que tuviera la información necesaria para que cada thread, de manera independiente, pudiese realizar su cómputo.

Ahora, debido a que se tenía de manera fija el tamaño de la imagen a ser procesada, fácilmente se particionó el cómputo en bloques continuos, donde la cantidad total de bloques es igual al número de hilos, y los tamaños de los bloques son lo más parecidos posible.

C. Particionado Block-Cyclic, Clásico

Se definieron chunks de tamaño 64 debido a que la estructura en la que se guardan los pixeles es de 1 byte por cada pixel. Como la máquina en la que se realizaron las pruebas tenía 64 bytes de caché, este es un tamaño acorde.

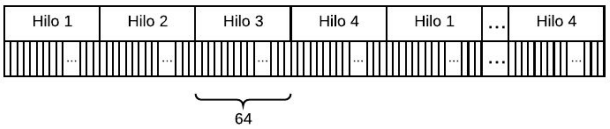


Fig. 6 Diseño del particionado block-cyclic para abordar el problema del filtro gaussiano

Ahora, se lanzan cada uno de los hilos y se define como sección crítica una nueva función cuyo parámetro principal es el tamaño del chunk y el tamaño de la imagen, con estos dos parámetros por cada hilo se ejecuta el chunk determinado y se pasa al siguiente hasta alcanzar el final.

D. Particionado Block-Cyclic, Thread Pool

El diseño de esta arquitectura es en parte similar al particionado Block-Cyclic definido en la sección anterior. En un principio la idea es usar hilos de tal forma que la repartición de píxeles sea eficiente y se eviten efectos que puedan afectar el desempeño como false sharing.

Algo que no hace el diseño de la sección anterior es que cada hilo ya tiene definido las acciones ejecuta lo cual puede ser malo en entornos más grandes, puesto que no permite que un thread ejecute otro trabajo apenas se desocupe (reutilizar el thread). Debido a esto se decide explorar esta aproximación.

Un Thread Pool es una forma de paralelismo que permite que varias tareas (no necesariamente iguales) sean ejecutadas por hilos que pueden ser reutilizados.

Usualmente un Thread Pool se compone de Workers (work crew), quienes ejecutan las acciones y puede o no existir otro actor Boss quien se encarga de asignar las acciones. Este diseño se puede ver claramente en el siguiente diagrama:

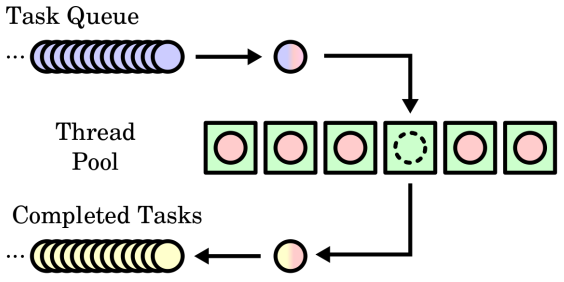


Fig. 7 Diseño estándar de una arquitectura de tipo Thread Pool[5].

Los recuadros verdes corresponden a los workers que vendrían a ser los hilos que esperan a que se les asigne una nueva tarea de una cola FIFO de tareas. Cuando ya no qu

Cada tarea se define por un conjunto de parámetros y la tarea/función como tal a ejecutar. En nuestro caso el conjunto de parámetros es principalmente entre que rangos del arreglo de pixeles se va a procesar (recordando lo que se menciona del tamaño del cache en la sección B). La tarea a ejecutar siempre va a ser la misma por lo que se puede simplificar un poco la arquitectura para que asuma la misma tarea.

La definición de cada worker por otro lado es realmente un hilo que va a ejecutarse "infinitamente" hasta que no existan más tareas pendientes en la cola. Es importante tener en cuenta tareas que requieren un **mutex** como por ejemplo al momento de sacar un elemento de la cola, ya que podrían suceder cosas como que dos o más threads ejecuten el mismo trabajo, por lo que la sincronización es de vital importancia.

E. Particionado Multiprocesador, GPU

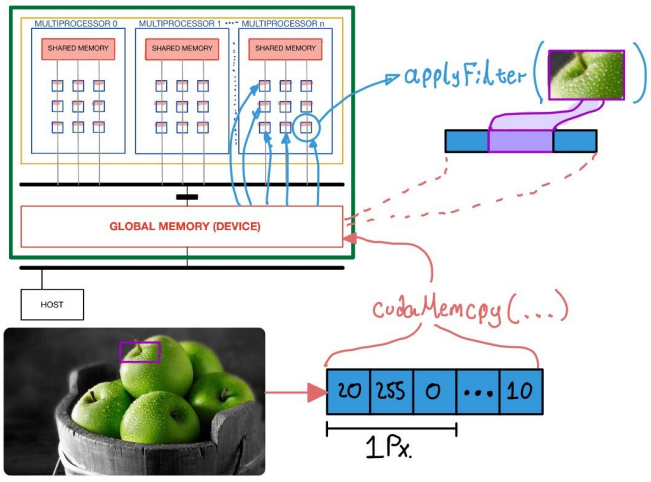


Fig. 8 Image blurring en GPU.

Para éste diseño se trabajó con el modelo de distribución por bloques, además fué realizado en plataforma COLAB, por lo que fué necesario instalar CUDA. Es importante resaltar que para manejar un adecuado balanceo de carga, éste debía ser dinámico, pues al tratarse de una plataforma en la nube, la cantidad de hilos disponibles ó la cantidad de multiprocesadores, podría variar cada vez que se ejecutara el programa. Para hacer ese balanceo de carga dinámico se usaron algunas utilidades que provee CUDA, como lo son device y cuda runtime.

La diferencia principal de éste diseño, con los anteriores, es que contempla el uso de una cantidad mucho mayor de hilos, si bien los anteriores diseños usaban un máximo de 32 hilos, éste diseño, al correr sobre una GPU, está especialmente pensado para gestionar una cantidad mucho mayor de hilos.

Para hacer un cómputo mucho más dinámico, se definió el chunk también de forma dinámica. Es así que se calcula el chunk más óptimo en función del número de pixeles a ser computados y del número de hilos que se pretende lanzar. Permitiendo así una mayor ocupación de la capacidad de cómputo de la GPU.

IV. Experimentos

Los experimentos se realizaron haciendo uso de varios scripts que nos permitieron separar la ejecución de cada prueba por lo que si algo fallaba solo era necesario volver a ejecutar el script correspondiente a la tarea que falló. Los resultados también fueron exportados en texto plano para evitar tener que volver a ejecutar el código y perder el trabajo realizado.

Dicho esto, se ejecutaron 3 experimentos para cada diseño, cada uno con los siguientes tamaños de imagen:

- **720p**: 1280 x 720 px
- **1080p**: 3968 x 2976 px
- **4k**: 5028 x 3352 px

En cada ejecución de los modelos POSIX y OpenMp se evaluaron los tres diseños (Blockwise, Block-Cyclic y Thread Pool), ejecutando para cada tamaño de kernel y número de threads 5 veces y promediando para evitar inconsistencias.

Los tamaños de kernel y número de threads que fueron considerados fueron los siguientes:

- Kernel = {3, 5, 7, 9, 11, 13, 15}
- Threads = {1, 2, 4, 8, 16, 32}

En cuanto al modelo basado en CUDA, se ejecutaron las mismas pruebas pero esta vez, sólo se usó el modelo de distribución por bloques, por ser el más consistente con el balanceo de carga dinámico que se implementó.

Otra cosa importante que se consideró es que era necesario ejecutar el código secuencial para tenerlo como base al momento de calcular el speedup. Para este se tuvo en cuenta la misma consideración de ejecutar 5 veces y promediar para evitar inconsistencias.

V. Resultados

A. Secuencial

Como se mencionó en la sección anterior, para cada tamaño de imágen se ejecutó el problema secuencialmente, ya que nos sirve de referencia para realizar el cálculo del speedup de cada uno de los métodos probados. A continuación presentamos el resultado de ejecutar esto para una imágen 5028x3352px.

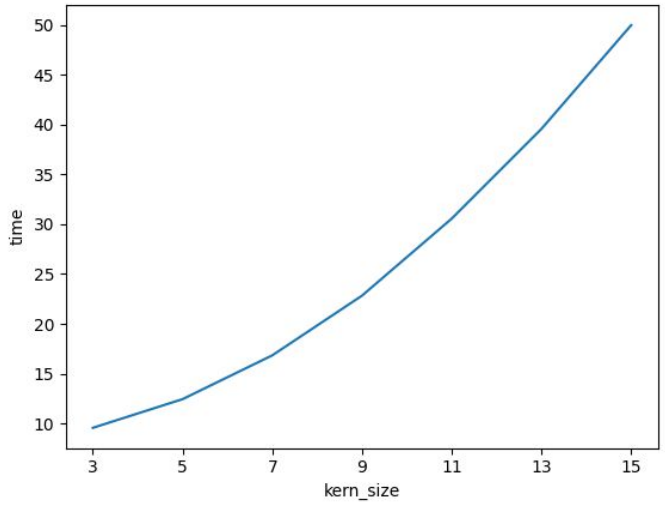


Fig. 9 Ejecución secuencial para 4k.

Como se puede ver en la figura cada vez que se aumente el tamaño del kernel la duración de la ejecución aumenta, llegado a durar cerca de 50 segundos por imagen para un kernel de 16 píxeles de lado.

Este comportamiento es similar para los otros tamaños de imagen como se puede ver en estas gráficas:

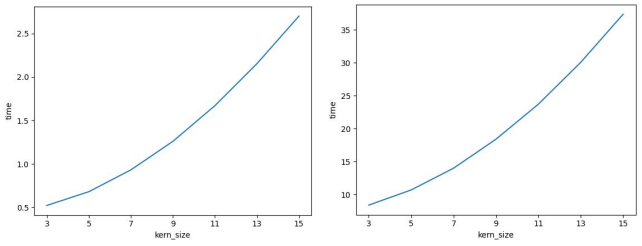


Fig. 10 Ejecución secuencial para (a)720p y (b)1080p.

B. Blockwise

Este primer método como se explicó anteriormente particiona la cantidad de pixeles a procesar de forma igual para cada thread. A continuación se puede ver una gráfica que muestra el tiempo de ejecución para cada tamaño de kernel y el número de threads utilizados en una imagen 4k.

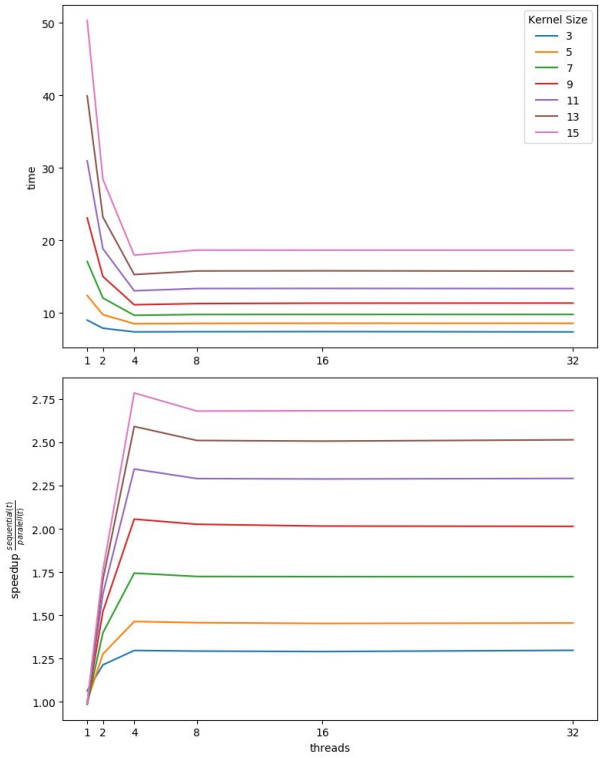
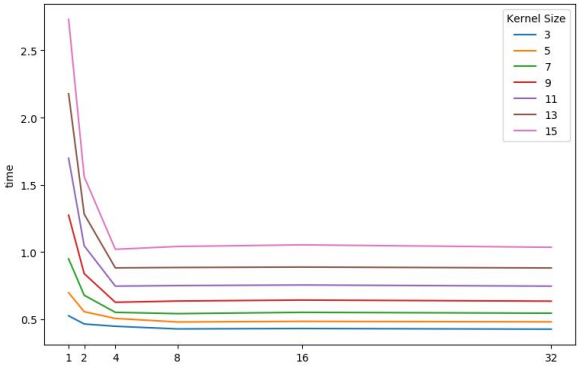


Fig. 11 Resultados ejecución Blockwise para 4k.

A primera vista se puede ver que el tiempo de ejecución es el mismo o similar al gastado sin utilizar hilos, por lo que el speedup es mínimo o cercano a cero, sin embargo al aumentar la cantidad de hilos el tiempo baja considerablemente. Esto se puede ver desde el punto de vista del speedup en donde también se puede ver existe un tope en donde aumentar la cantidad de hilos ya no presenta mejora alguna en el desempeño.

A continuación presentamos solamente los plots de tiempo para 720p y 1080p puesto que la gráfica de speedup es similar:



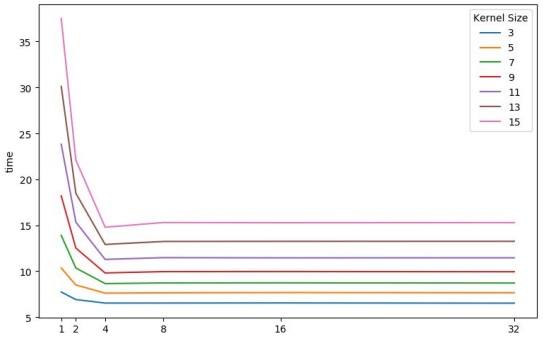


Fig. 12 Resultados ejecución Blockwise para (a)720p y (b)1080p .

C. *Block-Cyclic*

Para block cyclic data se puede ver un resultado similar, de hecho el speedup máximo que se logra es casi el mismo que usando particionamiento blockwise.

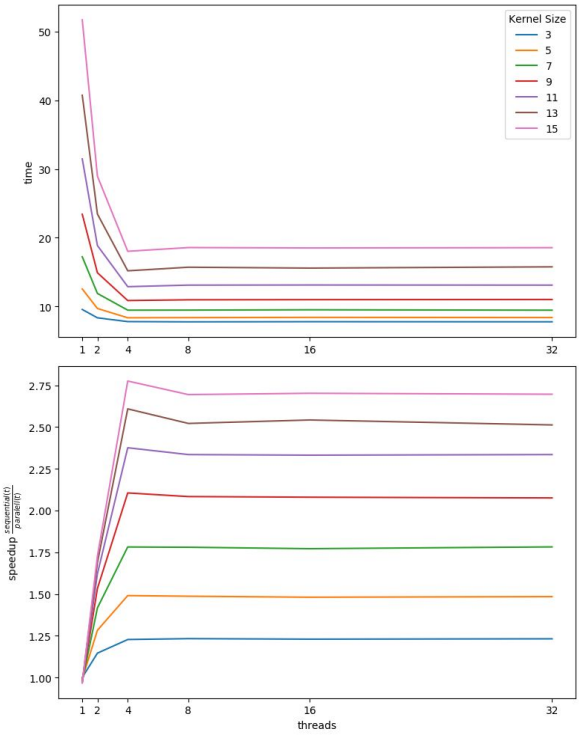


Fig. 13 Resultados ejecución Block-Cyclic para 4k.

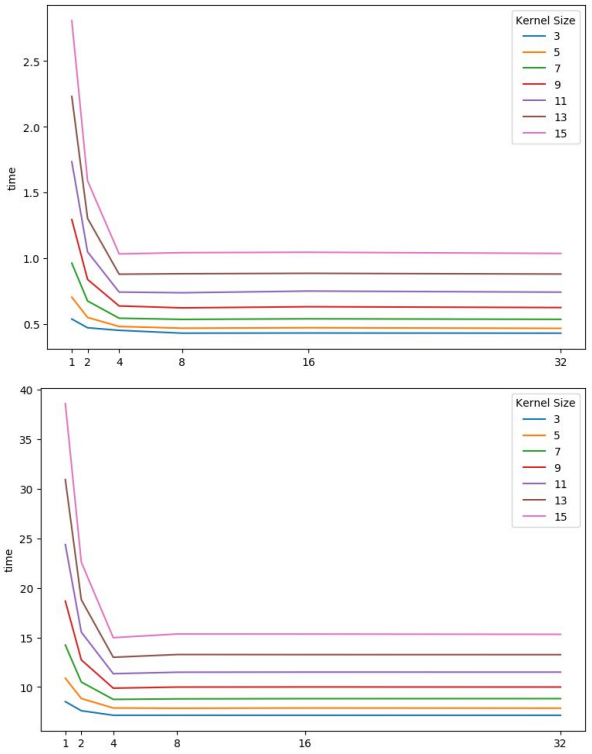


Fig. 14 Resultados ejecución Block Cyclic para (a)720p y (b)1080p .

D. *Thread Pool*

Thread Pool es un caso especial del que hablaremos más detalladamente dado que su mejoría como se puede apreciar no es tan alta.

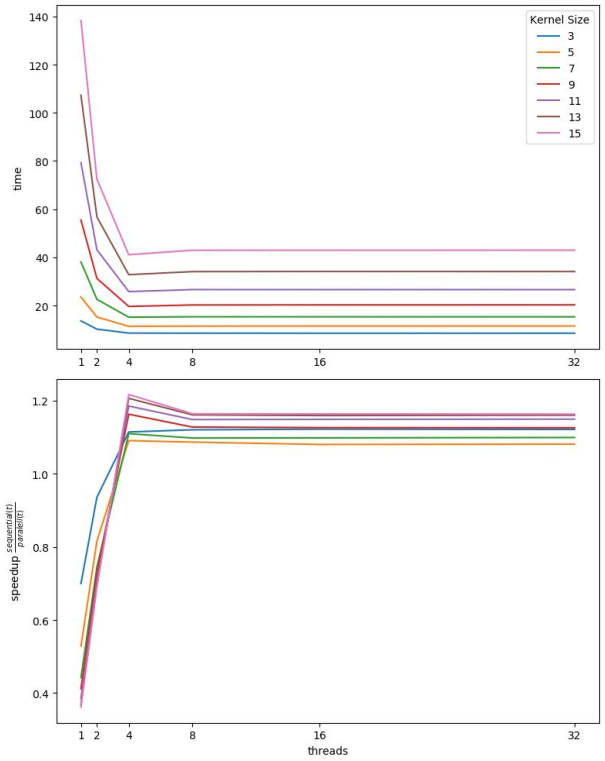


Fig. 15 Resultados ejecución Thread Pool para 4k.

Como se puede apreciar en la gráfica superior, solo se presenta mejoría únicamente después de aumentar la cantidad de hilos después de 4, Pero la mejoría en el desempeño no llega ni a la mitad del speedup presentado por los otros métodos. Esto se puede justificar principalmente debido a que como mencionamos en el diseño del pool, son necesarias más acciones para evitar que dos o más threads tomen el mismo trabajo por lo que es necesario bloquear los threads mientras se le asigna un trabajo a determinado hilo, por lo que el rendimiento se ve afectado.

Esto no quiere decir que Thread Pool sea ineficiente, si no que este código en particular no tiene sentido que sea ejecutado de esta forma.

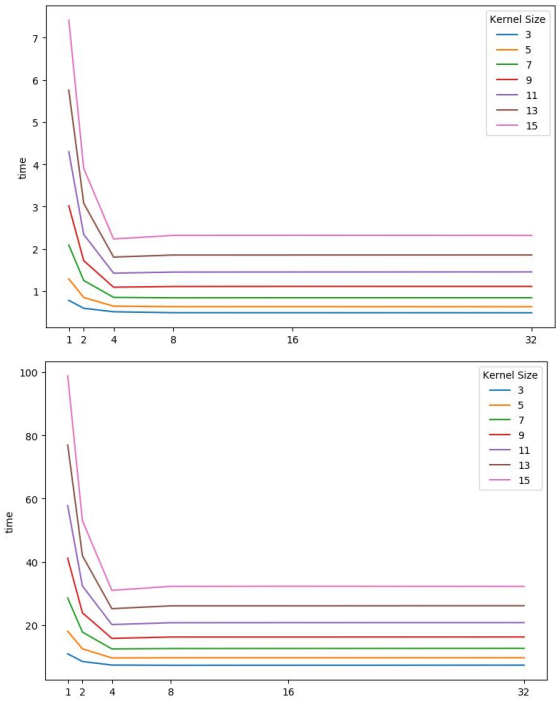


Fig. 16 Resultados ejecución Thread Pool para (a)720p y (b)1080p .

E. Blockwise OMP

Esta versión es similar al diseño anterior que tenemos con hilos POSIX. De hecho los tiempos de ejecución son similares por lo que el speedup logrado se encuentra en el mismo rango.

Esto nos da un indicio de que si bien los hilos con POSIX pueden llegar a ser más rápidos que hacer uso de la implementación de OMP, si obtenemos un performance similar OMP es un buen tradeoff entre rendimiento y complejidad del código. Obviamente esto depende de la aplicación esperada y de qué tanto control sobre el paralelismo se quiera tener.

A continuación presentamos los resultados obtenidos:

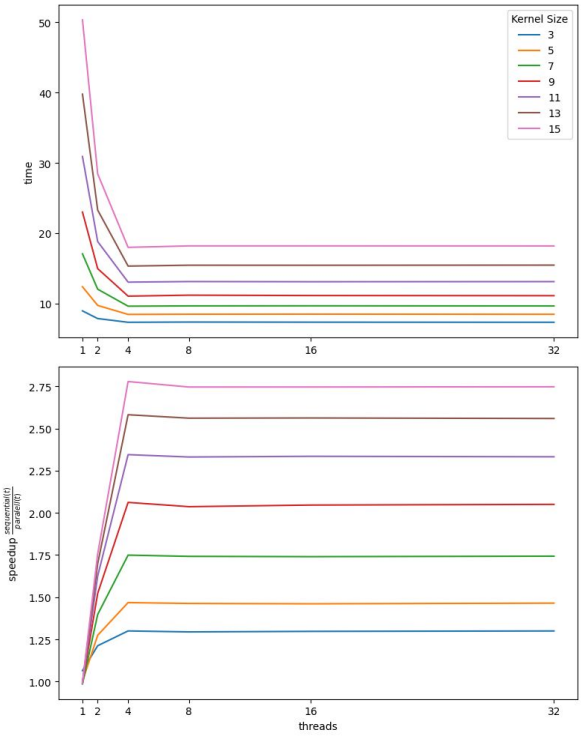


Fig. 17 Resultados ejecución Blockwise POSIX para 4k.

Y para las otras dos pruebas:

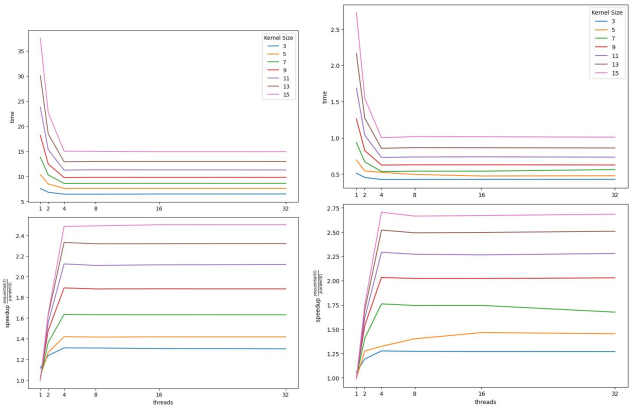


Fig. 18 Resultados ejecución Blockwise POSIX para (a)720p y (b) 1080p .

F. Block-Cyclic OMP

Los resultados de esta ejecución son similares a los obtenidos en la ejecución de block-cyclic con POSIX.

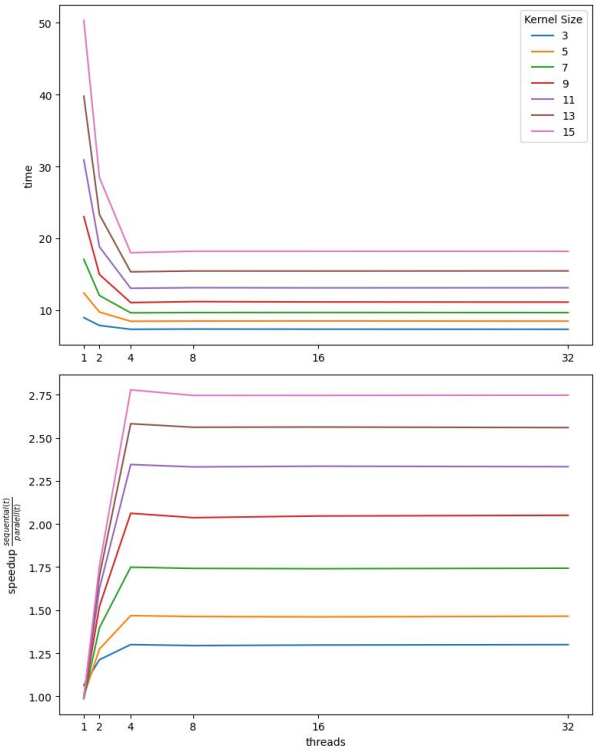


Fig. 19 Resultados ejecución Block-Cyclic POSIX para 4k.

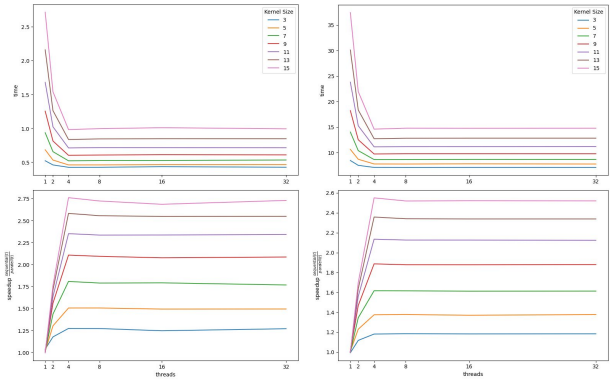


Fig. 20 Resultados ejecución Block-Cyclic POSIX para (a)720p y (b)1080p .

G. CUDA

La mejora de rendimiento haciendo uso de CUDA para procesar de forma eficiente la imágen en una GPU nos permite procesar imágenes más rápidamente debido a que tenemos más threads procesando en conjunto las diferentes secciones de la imágen.

Debito a esto realizamos pruebas con tamaños de kernel más grandes (hasta 45) como podemos ver a continuación.

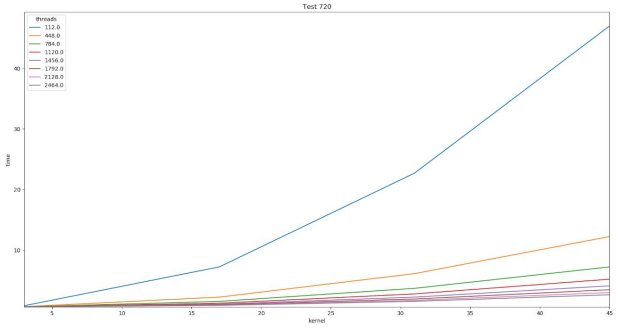


Fig. 21 Resultados ejecución CUDA para 720p.

Haciendo uso de la GPU pudimos obtener por lo tanto un speedup mucho más grande a medida que aumentamos los tamaños de los kernels.

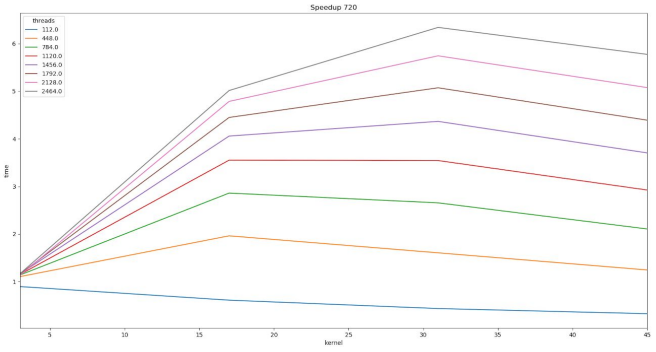


Fig. 22 Speedup ejecución CUDA para 720p.

En general este comportamiento se puede ver a lo largo de las otras ejecuciones con los diferentes tamaños de imágen.

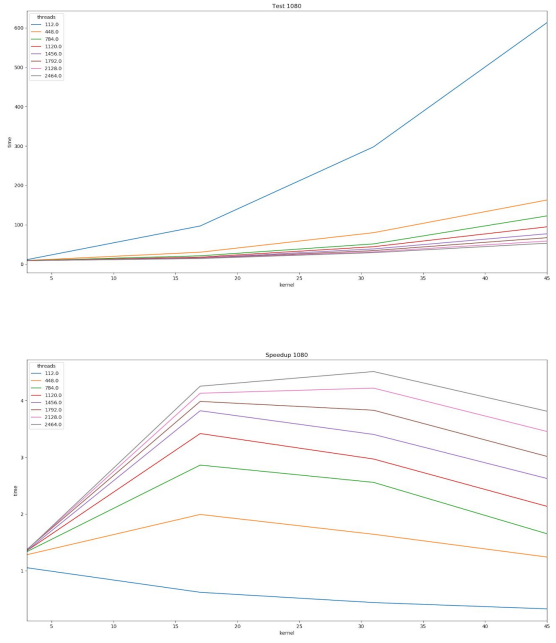


Fig. 23 Tiempos(a) y Speedup(b) ejecución CUDA para 1080p.

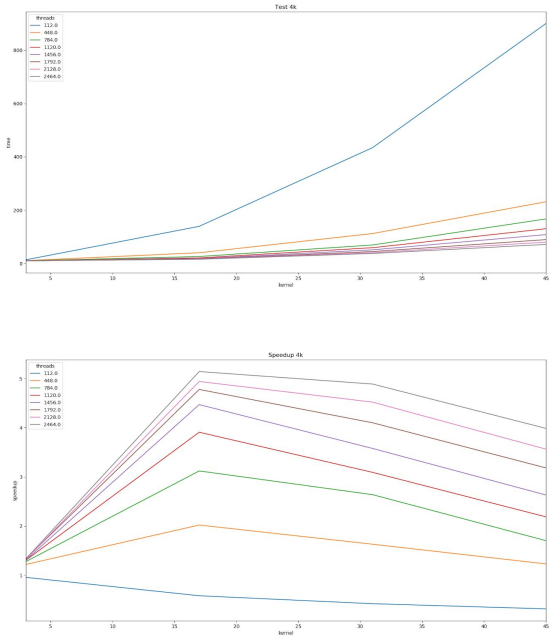


Fig. 24 Tiempos(a) y Speedup(b) ejecución CUDA para 4k.

Cabe notar que para las gráficas de speedup no ejecutamos todo el código secuencial, esto debido a que ejecutar el blur para un kernel bastante grande puede resultarnos muy costoso(posiblemente no seríamos capaces de ejecutar todo debido a la cantidades de operaciones a realizar), por ello lo que decidimos hacer fue tomar los resultados secuenciales para los kernels en el rango 3-17 y extrapolamos los tiempos de ejecución hasta kernels de tamaño 45.

Otra cosa importante a mencionar es las especificaciones de nuestra tarjeta gráfica (NVIDIA TESLA V100). Nuestro código muestra las especificaciones de la tarjeta gráfica antes de ejecutar:

```
Detected 1 CUDA Capable device(s)
MapSMtoCores for SM 7.5 is undefined.  Default to use 128 Cores/SM
MapSMtoCores for SM 7.5 is undefined.  Default to use 128 Cores/SM
MapSMtoCores for SM 7.5 is undefined.  Default to use 128 Cores/SM
40 Multiprocessors, 128 CUDA Cores/MP | 5120 CUDA Cores
Maximum number of threads per block: 1024
```

Fig. 25 Specs TESLA V100.

También logramos ejecutar el profiler de CUDA (nvvp) de forma local por lo que fue posible visualizar que los tiempos de ejecución de nuestra tarjeta y ocupación de esta. Ciertamente las especificaciones de nuestra tarjeta local es diferente, por lo que tal vez los resultados varíen un poco.


```
Detected 1 CUDA Capable device(s)
3 Multiprocessors, 128 CUDA Cores/MP | 384 CUDA Cores
Maximum number of threads per block: 1024
```

Fig. 26 Ejecución nvpp con kernel 33 y imagen 720p usando todos los threads y cores (a) y Specs máquina local(b).

En general se puede ver que hay un speedup considerable, teniendo en cuenta que el tamaño de kernel con el que se probó es bastante grande y probablemente tardaríamos mucho tiempo en calcular esta imágen de forma secuencial. El cudaMemcpy es debido al acceso que tienen los cores a la memoria global y podemos ver pequeños ticks que corresponden las transacciones de copiar la imágen de Host a Device (GPU) y lo contrario, de Device a Host.

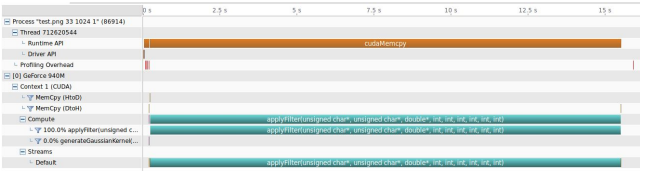


Fig. 27 Ejecución nvpp con kernel 33 y imagen 720p usando 1 core con 1024 threads.

El speedup se ve afectado considerablemente si no aprovechamos la tarjeta gráfica en su totalidad como se puede ver en la imágen anterior con un solo core de 1024 threads, sin embargo, algo que se mantiene constante es el hecho de que las operaciones para copiar de Device a Host y viceversa siguen consumiendo el mismo tiempo que en la prueba anterior, siendo lo que más consume tiempo la función que aplica el filtro. La función que genera el kernel también la ejecutamos en GPU, sin embargo no es tan intensiva en los cálculos que hay que ejecutar por lo que bien se podría realizar en CPU y despues copiar el kernel al Device.

VI. Análisis

Para empezar nuestro análisis podemos partir por observar que las gráficas para cada tamaño de kernel, en los 3 casos, nunca se intersectan, es decir, que a mayor tamaño de kernel, pese a que hay más tiempo de ejecución, también, hay un menor speed up. Pues el orden en el que aparecen las curvas es el inverso. Así, podemos deducir la existencia de una relación inversa entre el tiempo de ejecución y el speed up.

Note que dicho análisis se hacía para cada uno de los 3 casos de forma independiente, sin embargo, un valor agregado a dicho análisis es observar que para las imágenes con menor tamaño, los tiempos de ejecución son menores, pero también lo es el speed up. Esta comparación se realiza en contraste con las imágenes de mayor tamaño. Así pues, a mayor tamaño de imagen, mayor tiempo de ejecución, sin embargo, también un mayor speed up.

Ahora bien, analizando las pendientes de las gráficas de tiempo de ejecución, podemos observar que dichas pendientes son no crecientes, es por ello que nos encontramos con una

función convexa, en otras palabras, en cuanto menor es el número de threads, se presenta una mayor mejora en tiempo de ejecución al añadir otro thread. Además, un punto de inflexión en la función convexa que refleja el por qué a partir de ese punto es totalmente innecesario añadir más threads al programa en cuestión.

En cuanto a las gráficas de speed up, de igual manera podemos ver que se tratan de funciones convexas, esto, explicado directamente por la relación inversa de ambas variables discutida anteriormente. Igualmente, dicho fenómeno explica por qué unas pendientes son no decrecientes, mientras que otras son no crecientes. Algo a resaltar es el hecho de que usar una cantidad muy baja de hilos conlleva a un speed up inferior a 1. Por lo que de igual manera, asignar una cantidad muy baja de hilos no es adecuado, algo interesante es que esa cantidad mínima de hilos es mayor para mayores tamaños de kernel.

En comparativa, podemos observar que los particionados Blockwise y Block-Cyclic tienen un performance muy similar, esto se explica principalmente por el hecho de que ambos hicieron uso de la misma librería y ambos se basaron en el mismo programa de procesamiento secuencial. Hay que resaltar que ambos resultan tener un mejor performance que el tercer particionado; thread pool. De hecho, esta última distribución de procesamiento alcanza un speedup de hasta 1.2, mientras que las otras llegan a un speedup de hasta 2.75. Por ello podemos observar la notoria diferencia de performance entre distintas formas de distribución de procesamiento para los hilos.

En cuanto a la comparativa de rendimiento computacional entre POSIX y OpenMp, podemos ver que en todos los diferencias son mínimas, está afirmación aplica para todos los diseños aquí presentados, sin embargo, es importante señalar que en general los códigos de OpenMp resultaron más sencillos que los de POSIX, también más cortos, si bien no es una ventaja computacional sí es una ventaja de implementación.

Para los experimentos que se realizaron en CUDA se usaron parámetros mucho más robustos, ésto nos da una idea de la diferencia computacional entre ésta alternativa y las dos anteriores. Para ser más enfáticos, el programa en CUDA era fué testeado con un Kernel de tamaño hasta 45, dicho procesamiento sería impráctico en POSIX y OpenMP con una cantidad de hilos tan limitada.

VII. CONCLUSIONES

El uso de hilos puede realizar mejoras de rendimiento muy importantes en tareas que pueden descomponerse en otras más pequeñas y que pueden resolverse de manera independiente.

Sin embargo, el paso de una tarea de su versión secuencial a su versión paralela puede ser bastante engorroso, por lo tanto, el programador debe asegurarse antes de tomar cualquier decisión, que la tarea que desea paralelizar tenga una oportunidad de mejoría considerable. Puesto que para tareas muy pequeñas, el rendimiento de un algoritmo secuencial versus uno paralelo es muy similar.

Cuando se desea paralelizar una tarea, es necesario también conocer bien acerca de la arquitectura del computador y la manera en que el sistema operativo gestiona los hilos, ya que de estas especificidades puede depender también una ejecución óptima del programa. Para ejemplificar, en el caso de los sistemas operativos Linux, el *false sharing* es un problema bastante común, pero que puede ser evitado siempre y cuando se tenga conocimiento sobre él y sobre cómo son gestionados los hilos a un nivel más bajo.

Otro punto a resaltar es que realizando la implementación de un esquema de distribución block-cyclic es posible emular los otros dos, variando el tamaño del chunk. Por lo tanto, debería ser la opción a elegir cuando se desee comparar el rendimiento de los diferentes esquemas.

Por último, la implementación de un thread pool es bastante útil para cierto tipo de problemas y de sistemas que necesiten gestionar muchas tareas en segundo plano, sin embargo para problemas como este, donde se necesitaba resolver un problema pequeño, es una solución sobredimensionada debido al esfuerzo que requiere y su mínima diferencia en rendimiento con los métodos tradicionales.

OpenMP resulta ser una alternativa más sencilla que POSIX. Sin implicar una pérdida significativa en eficiencia computacional en la mayoría de los casos, OpenMP resulta ser más sencillo de programar por las funcionalidades que nos ofrece dicho API. Además OpenMP, en su búsqueda de crear un estándar para la programación multihilos ha logrado una globalización parcial y un soporte multilenguaje y multiplataforma bastante robusto, lo cual, según sea el caso, puede significar otra ventaja.

Con los experimentos realizados con CUDA, pudimos visualizar la importancia del cómputo en GPU, pues si bien logramos resultados significativos con los dos experimentos anteriores, podemos ver que, siendo concretos, los resultados del procesamiento en GPU no son comparables a los de procesamiento en un procesador principal. Y en general, la eficiencia de muchas de las tareas paralelizables, depende directamente del número de hilos y de la gestión del cómputo en los mismos. Y es precisamente por esa razón que el procesamiento en GPU existe, porque para ciertas tareas computacionales, como la aquí tratada, implica descomunales mejoras en eficiencia computacional.

Referencias

[1] Alejandro Dominguez Torres, The origin and history of Convolution I

[2] Schmidt, B., Gonzalez-Dominguez, J., Hundt, C. and Schlarb, M., 2018. Parallel Programming. 1st ed. Massachusetts: Morgan Kauffman Publishers.

[3] 2020. The Gaussian Kernel. [ebook] University of Wisconsin. Disponible en : <http://pages.stat.wisc.edu/~mchung/teaching/MIA/reading/diffusion.gaussian.kernel.pdf.pdf> [Accessed 3 April 2020].

[4] IEEE and The Open Group POSIX definition, disponible en: https://pubs.opengroup.org/onlinepubs/9699919799/

[5] Carnegie Mellon School introduction to POSIX in Linux, disponible en: https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html

[6] https://en.wikipedia.org/wiki/Thread_pool