Introducción a OpenMP

Laboratorio 1 de Programación paralela

Universitat de Barcelona
Facultat de Matemàtiques i Informàtica

Trabajo realizado por
Marcos Plaza González

Tabla de contenidos

Introducción y objetivos

Para esta primera práctica de laboratorio de la asignatura utilizaremos un algoritmo para transformar el espacio de color *GRB* a *RGBA*. Mediante este método de conversión entre ambos espacios, veremos aplicadas las primeras nociones de la librería *OpenMP* aprendidas en la teoría; las diferentes directivas, diferentes tipos de *sheduling*, etcétera.

El objetivo primario de este laboratorio es ver como la programación paralela, en este caso usando *OpenMP*, puede acelerar los cálculos de un proceso. A la vez debemos obtener los mismos resultados finales utilizando exactamente el mismo *hardware*.

De forma complementaria, tal y como ya se menciona en el apartado introductorio del enunciado, trabajaremos sobre los diferentes parámetros que pueden añadir un plus de velocidad a nuestro código, sin necesidad de utilizar la librería *OpenMP*. Usaremos *flags* de optimización de compilador y tendremos en consideración la alineación de memoria en una estructura de datos, a la vez que la localidad de datos.

En este informe se contestaran a las preguntas propuestas en el enunciado y se aportaran tiempos de ejecución en tablas para reforzar las distintas explicaciones.

La práctica se ha realizado utilizando un único equipo, de modo que los tiempos en las ejecuciones sean equiparables, utilizando *Ubuntu 20.4 LTS* como sistema operativo.

Trabajando sobre las preguntas propuestas

Indicaciones sobre el código

Para automatizar el proceso de tomar las medidas del tiempo de ejecución, se ha adaptado un poco el código de manera que se repita el proceso tantas veces como se especifique por argumento en la entrada.
Por otra parte, en algunos ejercicios se pide que se implemente alguna función de conversión con algunos cambios. Se ha utilizado un puntero que apunte a una función distinta según el número entero que se pase por argumento junto al parámetro anterior. Aunque no se específica en el enunciado, también se han creado las respectivas funciones para el uso de *scheduling* metiendo por parámetro el *chunk_size*.
De esta manera el código a entregar queda todo en un único fichero. Para más información utilizar el parámetro ***-h***, en ejecutar el código.

En el directorio llamado ***Output*** se proporcionan archivos de texto plano con la salida de cada ejercicio.

Por defecto se ha dejado el *struct uchar3* con el atributo *aligned(4)*. Si se requiere el uso del que no contempla el *memory alignement* hace falta descomentarlo y comentar el *struct* anterior, ya que llevan el mismo nombre. De la misma manera el bucle paralelizado que itera sobre *EXPERIMENT_ITERATIONS*, se ha dejado por defecto. Para utilizar el otro *for,* sin paralelizar, se debe descomentar y comentar el anterior bucle *for*.

Para realizar los experimentos del tiempo dedicado a cada ejecución del código, se han tomado 10 muestras de las que posteriormente se calcula la media.

Primer apartado

En este primer apartado se nos pide simplemente ver como funciona el código adjunto, para a continuación ejecutarlo y anotar el tiempo de ejecución.

Tal y como observamos, empleando un par de *structs* almacenaremos los diferentes valores tanto para los píxeles en codificación *GBR* como para *RGBA*. Cada uno de estos *structs* ocupará tres y cuatro bytes en memoria respectivamente.
En cuanto al algoritmo, vemos que no es nada enrevesado. Inicialmente se reserva memoria para un vector unidimensional del tipo *uchar3*, equivalente al espacio que utilizaría una imagen con una resolución *4K (3840x2160),* y se les asigna un valor a cada uno de estos píxeles. A continuación se reserva espacio para otro vector “destino” pero esta vez en *RGBA* (*uchar4*). A continuación se realiza la conversión un número fijo de iteraciones (en este primer ejercicio son 100), mediante la función *convGBR2RGBA*. Dicha función traslada los valores de *GBR* a *RGB*, y se le asigna el valor máximo (255) para el parámetro *Alpha*, que en este espacio de color indica la opacidad o transparencia del píxel.
Por último se comprueba que el resultado es correcto mediante *checkResults*.

El algoritmo utilizado es bastante simple, aunque como ya veremos podemos optimizar el código (así como la compilación de este) para llegar a tiempos de ejecución menores. Por el momento las 10 ejecuciones resultan en un tiempo medio de 5,93 segundos. Para más información consultar el archivo *Output/exercici1.txt*.

Segundo apartado

Información sobre los flags de optimización *-O*

Ahora toca experimentar un poco con los *flags* de optimización *-O*. Este *flag* controla el nivel de optimización para nuestro código. Para ello debemos hacer las modificaciones pertinentes en el *Makefile.*

Si echamos un vistazo a la documentación que hay en internet sobre estos parámetros de optimización, generalmente encontramos que, en utilizarlos, ocupamos más tiempo y recursos en tiempo de compilación (capacidad de la *CPU*, pero sobretodo memoria) para a cambio, obtener un tiempo de ejecución menor. También sacrificamos la capacidad de depurar normalmente nuestro programa.
Cuando especificamos *-O, -O2, -O3* y *-Ofast*, estamos introduciendo en realidad una batería de *flags* de optimización por debajo, que se van incrementando en número y tipo, a medida que accedemos a un nivel superior. Por ejemplo, utilizando *-O2* usamos los *flags* de optimización que se activan en la opción *-O* además de unos cuantos *flags* adicionales. Esto se da de la misma manera en *-O3* y *-Ofast*.

En *-O1* y en *-O2* incrementamos el uso de memoria para el preprocesado y así obtener un mejor rendimiento final. *-O2* es el nivel de optimización más recomendado si el sistema no requiere de necesidades especiales. Además este tipo de optimización emplea un tipo de registros de la *CPU* para realizar operaciones vectoriales; *SSE* y *AVX*.

Por otro lado el *-O3* y *-Ofast* no son tan recomendados como los anteriores. *-O3* es la más cara de las optimizaciones en términos de compilado y memoria, además de romper algunos paquetes. Los bucles en el código fuente serán vectorizados, y por ende usan el tipo de registros de la *CPU* que se han mencionado anteriormente; *AVX* y *YMM* (en usar la opción *-ftree-vectorize*). Este último, también se encarga de operar con vectores. Por último *-Ofast* es una nueva característica incorporada a partir de *GCC 4.7*. Consiste en añadir algunos parámetros más, y además rompe con estándares de compilación estrictos.

¿Riesgos de usar *-O flags*?

Así como *-O* y *-O2* pueden ayudar a optimizar nuestro código *-O3* y *-Ofast* pueden no resultar beneficiosas al fin y al cabo debido a la cantidad de memoria que se necesita. Además de esto, según la información consultada, el problema con este tipo de optimizaciones es que pueden tener un impacto negativo en la efectividad de las caches de la *CPU*. Por ese motivo el rendimiento puede verse afectado.

Aunque como hemos visto, *-O2* es el nivel más recomendado, ya que consigue el mejor equilibrio entre tiempo de compilación y de rendimiento, como conclusión, lo mejor es conocer tu programa y utilizar aquella opción que más te convenga siempre teniendo en cuenta los recursos disponibles. Para los ejercicios siguientes, se compilará activando el *flag -Ofast*.

Tiempos de ejecución sobre nuestro código utilizando diferentes *-O*

A continuación se muestra una tabla con los tiempos de ejecución medios para cada *flag* (hace falta recordar que es la media sobre 10 experimentos y tomando 100 iteraciones). También se calculará el incremento de velocidad con respecto al apartado 1 ($T_0 = 5,93s = 5929400\mu s$), calculado según la fórmula vista en teoría; $speedup = T_0 \div T_p$.

-O flags	Tiempo medio *(s)*	Incremento de velocidad
-O0	5,93	1x
-O1	2,95	2,01x
-O2	2,96	2x
-O3	2,97	2x
-Ofast	3,00	1,98x

Tal y como se observa en la tabla anterior tan solo añadiendo uno de los *flags* estudiados podemos ver que la velocidad se duplica con respecto a *-O0*. Esto se debe a la utilización de más memoria para producir un código más rápido y pequeño aprovechando a su vez todas las técnicas implementadas a nivel del procesador.
Cabe señalar que a partir de utilizar *-O*, todos los niveles superiores comportan un peor incremento de velocidad, seguramente debido a el inconveniente causado por la caché del procesador o bien por la corrupción de algún paquete en tiempo de compilación.

Tercer apartado

En este tercer apartado de la práctica se nos pide implementar la función *convGRB2RGBA.* Dicha función debe mejorar la localidad de datos. Antes de ponernos a picar código, hemos estudiado este concepto. La localidad de datos en un sistema operativo, hace referencia al agrupamiento de los accesos de memoria por medio de la *CPU*. En la teoría de la asignatura se explican las modalidades de localidad de datos espacial y temporal.
La temporal se da cuando en un momento una posición de memoria particular es referenciada, entonces es muy probable que la misma ubicación vuelva a ser referenciada en un futuro cercano.
Por otro lado, la localidad espacial se da cuando una localización de memoria es referenciada en un momento concreto, entonces es probable que las localizaciones cercanas a ella sean también referenciadas pronto. Además, en el programa que hemos tratado en la práctica, tratamos con vectores, que una estructura de datos que puede ser tratada de forma secuencial. Con lo cual, si las posiciones están cercanas entre sí podemos ayudar al procesador a ser más ágil.
Si observamos la función *convGRB2RGBA* en la primera iteración del *for anidado* vamos a acceder a la posición 0 de ambos vectores *grb* y *rgba*. Por consiguiente, en la segunda iteración vamos a ir a hacer lectura y escritura a una posición alejada de la que se ha accedido anteriormente, si consideramos que el espacio del vector se reserva de forma ordenada dentro de la memoria (más concretamente iremos a acceder a la posición 3840). Como vemos estos saltos pueden ser contraproducentes, ya que los accesos se hallan lejanos entre sí (no estamos teniendo localidad de datos espacial). Además, ya que cada posición va a ser utilizada únicamente una vez, no podemos mejorar la localidad temporal explicada con anterioridad. Lo que si podemos hacer es mejorar la localidad de datos espacial.

¿Cómo mejoramos la localidad de datos espacial en *convGRB2RGBA_2*?
En realidad tenemos dos opciones en este caso particular.
La primera de todas es fijarse en como se inicializa el vector en la función principal del programa. Al tratarse de un vector unidimensional podemos acceder siempre a posiciones contiguas del vector yendo desde 0 a el máximo valor para el índice que se corresponde a *WIDTH * HEIGHT,* avanzando de 1 en 1 y realizando la conversión. Sólo necesitamos un único *for*. Lo malo de esta solución es que no estamos siguiendo el

algoritmo original. De todas maneras, a modo de curiosidad se ha dejado implementada en el código fuente adjunto, con el nombre *convGRB2RGBA_onefor*.
La última y la más correcta, es simplemente invirtiendo el recorrido del vector. En lugar de empezar a recorrerlo "a lo ancho", es decir por filas, podemos hacerlo por columnas. De esta manera pasamos de acceder de la posición 0 a la 3840, a la posición de 0 a 1 hasta llegar a *width*, después pasar de *width*+0 a *width*+1, etcétera. Comprobamos que el incremento de velocidad sea superior a las anteriores mediciones compilando con *-Ofast* para 100 iteraciones.

T_0 con distintas *-O (s)*	T_p *-Ofast* y *convGRB2RGBA_2 (s)*	Incremento de velocidad
5,93	0,47	12,62x
2,95	0,47	6,28x
2,96	0,47	6,3x
2,97	0,47	6,32x
3,00	0,47	6,38x

Como vemos el incremento de velocidad con respecto el apartado anterior (incluyendo el apartado uno con *-O0*) es mucho mayor que antes. Tal y como se menciona en el enunciado, como hemos obtenido tiempos menores al segundo, si incrementamos el número de iteraciones a 1000 con *-Ofast* obtenemos un tiempo medio de 4,72s.
Como vemos, la localidad de datos no es algo que podamos tomar a la ligera. Como programadores, tener esto en cuenta es de vital importancia para obtener un rendimiento notablemente mejor, al margen de *flags* de optimización.

Cuarto apartado

En este ejercicio, se propone partir de las optimizaciones de los anteriores. Es decir, mantenemos la compilación con *-Ofast* activado y utilizamos la función de conversión *convGRB2RGBA_2*. Simultáneamente mantenemos las 1000 iteraciones para cada experimento.
La *CPU* realiza lecturas y escrituras en la memoria de manera más eficiente cuando los datos están alineados de manera natural. Generalmente, alineación de memoria, significa que la dirección de los datos localizados en esta, corresponde a un múltiplo del tamaño de dichos datos.
Por otro lado, se dice que una dirección de memoria *a* está alineada en *n* bytes cuando *a* es un múltiplo de *n* bytes (donde *n* es necesariamente una potencia de 2).

¿Qué problemas pueden surgir debido a la alineación de memoria?
El problema es que si no hay una alineación correcta de los datos en memoria, la *CPU* va a necesitar muchos más ciclos de acceso para poder usar dichos datos. Normalmente la memoria es mucho más lenta que la *CPU* y si además vamos a necesitar varios ciclos extra para mover la información, el impacto negativo en el rendimiento de la aplicación será notable.

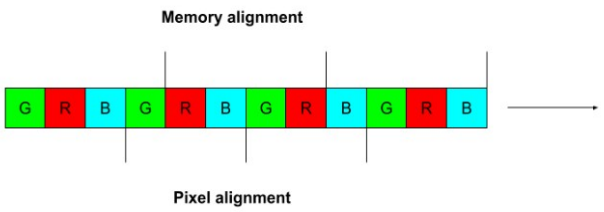

Figura 1. Esquema alineación de memoria, conjuntamente con el esquema de el almacenaje del vector *gbr*.

Como observamos en la figura anterior, el problema con el *struct uchar3* es que almacena tres valores por cada pixel (cada uno del tamaño de 1 *byte*, ya que son valores sin signo que van del 0 al 255. En total 3 *bytes*). Dado a este numero impar de valores en el *struct* vamos a tener problemas con la alineación de memoria, ya que como hemos dicho al principio, queremos que el *uchar3* quede almacenado en una posición divisible por su tamaño en bytes. Como vemos en el esquema, si no ponemos remedio a este problema, esta condición no se va a cumplir. A razón de que la posición de memoria óptima, siempre va a ser una potencia de dos, vamos a necesitar utilizar *__attribute__((aligned (4)))*.

Si buscamos en internet, podemos encontrar que *__attribute__* se utiliza para especificar propiedades especiales de variables, parámetros de función o *structs*, *union* y, en C ++, miembros de clase. Por otro lado, el *aligned(x)* (donde *x* es el número de *bytes*) hace referencia a que para cada *struct* vamos a establecer un mínimo de memoria límite, en este caso vamos a forzar que la alineación sea de 4 en 4 *bytes*. Con esto conseguimos alinear el almacenamiento de los pixeles con la segmentación usual de la memoria, de manera que las transferencias en *RAM* y en *CPU* sean menos costosas y más rápidas.

Para terminar con este ejercicio, como hemos dicho, especificaremos el atributo *aligned(4)* para el *struct uchar3.* A continuación mediremos los tiempos para 10 experimentos y calcularemos la media final. Recordemos que en el anterior apartado, llegamos a un tiempo medio $T_0 = 4,72s$. Si calculamos la media de las nuevas ejecuciones es de un $T_p = 2,45s$. Si calculamos el incremento de velocidad, nos sale: $speedup = T_0/T_p = 4,72/2,45 = 1,93x$.

Quinto apartado

Nuevamente se pide que se mejore aun más la función de conversión a partir de las modificaciones anteriores. Para ello, ahora si, utilizaremos la librería *OpenMP*. De momento comparamos cual de los dos *loops* da unos tiempos menores en ejecutar. También probaremos ambos paralelizados a la vez. Para llevar a cabo la tarea, emplearemos dos directivas básicas de *omp.h* a la vez, que ayudan a paralelizar *fors; #pragma omp parallel for*. Se muestran los resultados en la siguiente tabla.

loops* en *convGRB2RGBA_3	**Tiempo medio de ejecución *(s)***
Primer *for*	2,25
Segundo *for*	3,66
Ambos *for*	1,80

¿Qué bucle es mejor paralelizar?
El mejor resultado llega parelizando ambos *for* simultáneamente. Pero en respuesta a la pregunta que se realiza en el informe, el mejor rendimiento resulta en aplicar la directiva en el primer bucle *for*. Llegados a este punto, podemos concluir que en este problema específico, si paralelizamos ambos *for,* el tiempo se reduce significativamente, debido a que maximizamos la cantidad de tareas que son ejecutadas concurrentemente y el coste de generar hilos por ahora compensa.

¿Qué *scheduling* es mejor?

Ahora toca ver como funcionan los diferentes tipos de *scheduling* en un *loop construct*. El *scheduling* no es más que un tipo de planificación para los diferentes *threads* que van a trabajar concurrentemente en un bucle. Es decir, en el momento en el que nos encontramos con la directiva, vamos a repartir las iteraciones del bucle *for* entre todos los *threads* del equipo, de una manera distinta y específica a como lo hacíamos hasta ahora. Aquí entra en juego una nueva variable llamada *Chunk Size*, que corresponde a la cantidad de trabajo que se le va a asignar a un *thread* en concreto. Vamos a utilizar los *scheduling* explicados en clase y experimentaremos jugando con las variables numéricas; *chunk_size*, el tamaño de la imagen (*WIDTH* y *HEIGHT*), y el número de iteraciones (*EXPERIMENT_ITERATIONS*). Se van a representar los resultados en la siguiente tabla:

Scheduling type	*Chunk_size*	*(WIDTH, HEIGHT)*	*EXPERIMENT_ITERATIONS*	Tiempo medio *(s)*
static	1	(3840, 2160)	1000	4,86
dynamic	1	(3840, 2160)	1000	5,08
guided	1	(3840, 2160)	1000	5,13
static	5	(3840, 2160)	1000	4,87
dynamic	5	(3840, 2160)	1000	5,17
guided	5	(3840, 2160)	1000	5,03
static	15	(3840, 2160)	1000	5,17
dynamic	15	(3840, 2160)	1000	5,03
guided	15	(3840, 2160)	1000	4,88
static	1	(3840, 2160)	100	0,49
dynamic	15	(3840, 2160)	100	0,51
guided	15	(3840, 2160)	100	0,51
static	1	(7680, 2160)	1000	9,87
dynamic	15	(7680, 2160)	1000	10,09
guided	15	(7680, 2160)	1000	10,09
static	1	(7680, 2160)	1000	9,87
dynamic	15	(7680, 2160)	1000	10,09
guided	15	(7680, 2160)	1000	10,09
static	15	(7680, 2160)	1000	9,82
dynamic	1	(7680, 2160)	1000	10,11
guided	1	(7680, 2160)	1000	10,33

En realidad el *scheduling* que mejor funciona es *auto*, que es aquel que funciona sin especificar ningún tipo. Por otro lado, si miramos la tabla, en líneas generales, aquel que consigue un menor tiempo es el *static*, cuya función es repartir de manera equitativa y uniforme las iteraciones a los *threads* siguiendo una estrategia *round-robin* (cuando el número de iteraciones dividido entre el *chunk_size* no resulta en un residuo igual a 0). Claramente si aumentamos el tamaño de la imagen o aumentamos el número de iteraciones el tiempo de

computo va a resultar mayor. El aumento o disminución del valor de estas variables parece ser independiente al *chunk_size.* Para *chunk_size*, o el número de iteraciones por hilo, parece ser que aumentar su valor solo condiciona a que el comportamiento de *guided* se asemeje al de *static* ya que vamos haciendo que el set de iteraciones no pueda bajar de un determinado número. De manera global, el utilizar estos *schedules* no ayuda demasiado, ya que probablemente se están generando el mismo número de *threads* y a la vez hay muchos cómputos intermedios. El resultado de su rendimiento no varia demasiado en variar el valor de *chunk_size*.

Por último, vemos que si ponemos todas las variables como privadas, los resultados de la conversión son incorrectos. Por defecto ya son compartidas o *shared*, y el programa funciona normalmente. Podemos pensar que los punteros a las imágenes en *rgb* y en *rgba* no pueden ser variables privadas ya que de este modo estamos creando copias de las mismas, y a nosotros nos interesa operar sobre las imágenes originales (accesibles a través del puntero). Algo que sorprende es que aún poniendo estas dos variables como compartidas a la vez que *width*, *height* y *chunk_size* como privadas los resultados siguen siendo erróneos. No esperaba este resultado debido a que si se generan copias de dichas variables, igualmente los valores de estas no se ven modificados. Esto es así ya que cada hilo no se guarda una copia del valor que tienen las variables declaradas como privadas antes de entrar al bucle. Para guardar el valor antes, deberíamos declararlas como *firstprivate.*

Sexto apartado

El tiempo medio tras 10 experimentos, que se obtiene al paralelizar el *for* que itera sobre la constante *EXPERIMENT_ITERATIONS* es de 0,99 segundos. Como vemos, obtenemos un mejor rendimiento en paralelizar la ejecución de la función de conversión antes que paralelizar los bucles internos de dicha función. Esto puede ser debido a que el número de hilos y por ende el número de tareas que se crea y se destruye dentro de la función es mayor (*thread overhead*). Por lo tanto en colocar la directiva *#pragma omp parallel for* en el bucle que itera sobre *EXPERIMENT_ITERATIONS* estamos aumentando la eficiencia.

Séptimo apartado

A continuación el *output* resultante de imprimir el *id* de cada hilo generado:

```
marcosplaza@marcosplaza-MS-7C56:~/Escritorio/ProgramacioParalela/Lab1OpenMP$ make
g++ main.cpp -o main -fopenmp -Ofast
marcosplaza@marcosplaza-MS-7C56:~/Escritorio/ProgramacioParalela/Lab1OpenMP$ ./main 2 1
------------------------------------------------------------------
CONFIGURATION
        The number of executions is setted to 1.
        The conversion function selected is convGRB2RGBA_2.
------------------------------------------------------------------
EXECUTION
        Hey I'm the thread with id 3.
        Hey I'm the thread with id 0.
        Hey I'm the thread with id 9.
        Hey I'm the thread with id 4.
        Hey I'm the thread with id 5.
        Hey I'm the thread with id 8.
        Hey I'm the thread with id 10.
        Hey I'm the thread with id 7.
        Hey I'm the thread with id 11.
        Hey I'm the thread with id 6.
        Hey I'm the thread with id 1.
        Hey I'm the thread with id 2.
        convGRB2RGBA_2 time for 1000 iterations = 1040000µs
        Executed!! Results OK.

        The mean of time (execution) in 1 executions is 1.04e+06µs
------------------------------------------------------------------
```

Figura 2. *Output* generado al imprimir los *id* de cada hilo.

Como podemos observar, se crean 12 hilos que ejecutaran concurrentemente el *for.* Para imprimir cada *id* se deben hacer unas modificaciones en el código, de manera que separamos el *#pragma omp parallel for* en dos

partes. Para que se cree el grupo de hilos inicialmente utilizamos únicamente *#pragma omp parallel*. Imprimiremos una única vez los nombres de cada hilo utilizando *#pragma omp critical*, de modo que todos los *threads* creados entren de uno en uno para imprimir la región donde ponemos los *cout*. Posteriormente repartimos la carga de trabajo para cada miembro del equipo utilizando la directiva *#pragma omp for*.

Conclusiones finales de la práctica

Como hemos podido ver, manteniendo el tamaño de la imagen a una resolución *4K* y el número de iteraciones, un problema que inicialmente tardaba casi 5 segundos (teniendo en cuenta los resultados a partir del tercer apartado con 1000 iteraciones), ha pasado a realizar la misma tarea en menos de un segundo. Por lo tanto tenemos que el $speedup = T_0/T_p = 4,72/0,99 = 4,77x \approx 5x$. Nada más y nada menos que 5 veces más rápido.

Como idea principal, a través de la realización de la práctica hemos visto que:

- OpenMP es una buena herramienta para programación paralela a través de nuestra *CPU*.
- Existen muchas técnicas para mejorar el rendimiento de nuestros programas. Hemos visto lo importante que es tener en cuenta la *data locality* y el *memory alignement*, así como los *flags* de compilación.
- Conocer como funciona realmente nuestro código y como usamos el *hardware* que tenemos a disposición es realmente importante. *C/C++* son buenos lenguajes para aprender este tipo de comunicación *software-hardware*.
- Siempre podemos optimizar aún más nuestro código.

Referencias

- https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
- https://wiki.gentoo.org/wiki/GCC_optimization#-O
- https://www.wikiversus.com/informatica/procesadores/que-es-mmx-sse-avx/
- https://es.wikipedia.org/wiki/Cercan%C3%ADa_de_referencias
- https://gcc.gnu.org/onlinedocs/gcc/Variable-Attributes.html
- https://asm86.wordpress.com/2010/08/28/alineacion-de-memoria/
- https://es.qaz.wiki/wiki/Data_structure_alignment
- https://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Variable-Attributes.html#Variable-Attributes
- https://stackoverflow.com/questions/7055495/what-is-meaning-of-the-attribute-aligned4-in-the-first-line
- https://www.openmp.org//wp-content/uploads/OpenMP4.0.0.pdf
- http://jakascorner.com/blog/2016/06/omp-for-scheduling.html
- Diapositivas de teoría del Campus Virtual