Arquitecturas Avanzadas

Algoritmo de Floyd

Grupo AA-2-1

Julio Velasquez Cardenas 1397896

Sergio Prada Maeso 1459122

Juan Carlos Bermudez Rodriguez 1455486

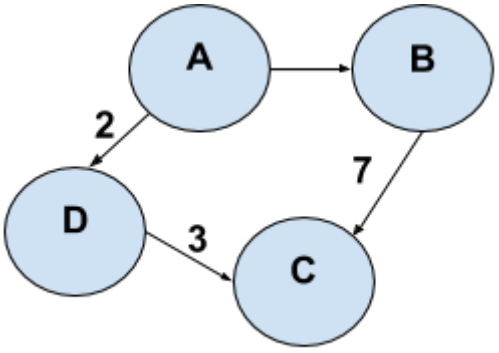
Índice

Descripción del algoritmo

Invención y usos

El algoritmo de Floyd fue publicado en el 1962 por Robert Floyd, aunque este es básicamente igual a otros algoritmos publicados años anteriores por Bernard Roy y Stephen Warshall. Uno de los usos más comunes de este algoritmo, y con el que tenemos más familiaridad como alumnos de ingeniería, es encontrar caminos mínimos entre cualquier pareja de nodos de un grafo ponderado dirigido.

En el siguiente ejemplo vemos como queremos encontrar un camino entre los nodos A y C pero no hay ninguna arista que conecte a ambos. En este caso, aplicamos el algoritmo de Floyd dándonos como resultado el camino entre A y C pasando por D ya que este es el de menor coste comparado con el camino pasando por B.



	A	B	C	D
A	0	5	?	2
B	-	0	7	-
C	-	-	0	-
D	-	-	-	0

Floyd

	A	B	C	D
A	0	5	5	2
B	-	0	7	-
C	-	-	0	-
D	-	-	-	0

Código

Describiremos ahora la funcionalidad del código:

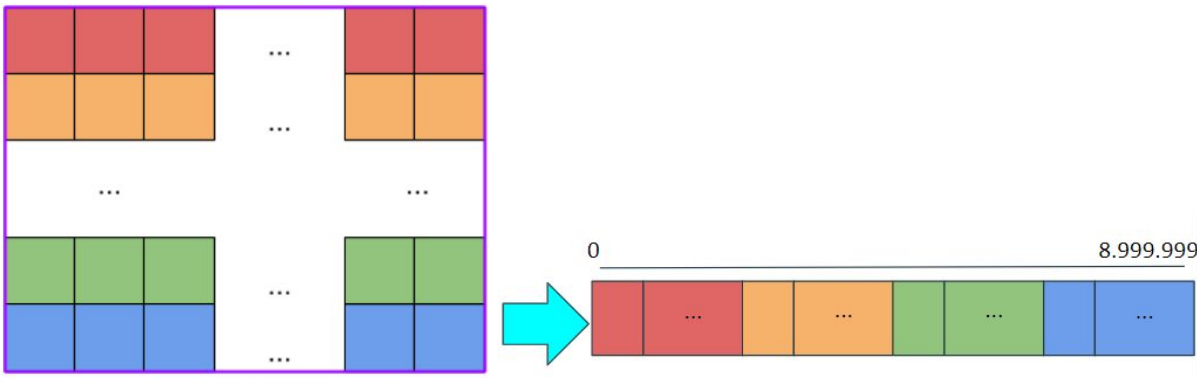
1) **Main**: Primeramente, tenemos un “main” que se encarga de (entre otras cosas) reservar memoria para la matriz (función malloc) con los parámetros especificados al ejecutar el programa (N y NHTR). Posteriormente, llama a la función “Floyd()” del programa.

2) **Floyd**: La función Floyd, consta de varios bucles for que explicaremos a continuación:
    - El primer bucle es el que se encarga de inicializar los valores de la matriz, y esto se hará con una función random la cual estará ligada a una seed (semilla) durante nuestro proyecto (para comprobar que los cambios hechos en el código no dan lugar a un resultado erróneo). Su función es dar valores aleatorios a los costes de cada nodo. Cabe resaltar además que el coste establecido para llegar a sí mismo es de 0, lo cual está indicado con un condicional “if” en el bucle más externo. Esto está hecho en un doble bucle for, por lo que la complejidad de este bucle es de N^2.
    - El segundo bucle se encarga de calcular los caminos mínimos (la funcionalidad del algoritmo de Floyd como tal). Para cada fila (que representa un punto en un mapa), calcula la distancia mínima para llegar al resto de puntos, por el camino menos costoso de todos (como el camino para llegar a sí mismo es 0, nada puede ser menor, así que la diagonal de la matriz nunca se modifica). Esto está hecho en un triple bucle for, por lo que la complejidad de este bucle es de N^3.
    - El tercer bucle es el del cálculo de nuestro Checksum. Se encarga de generar el valor que tendremos en cuenta para saber si nuestros cambios son correctos. Esto está hecho en un doble bucle for, por lo que la complejidad de este bucle es de N^2.

3) **Complejidad**: A partir de este análisis podemos determinar que la complejidad espacial es de N^2, ya que lo trabajamos con una matriz de N*N valores. La complejidad temporal es de N^3 ya que el bucle más complejo consta de un triple bucle “for”, todos ellos recorren desde 0 hasta N-1, por lo tanto, N*N*N.

4) **Acceso**: Trabajamos con una matriz de N*N, pero para visualizar mejor los accesos del código, la contemplaremos como un array de 0 a (N*N)-1 posiciones:

(Suponiendo que N = 3000)



En base a todo este análisis, empezaremos a hacer cambios en el código para tratar de aumentar la velocidad de ejecución.

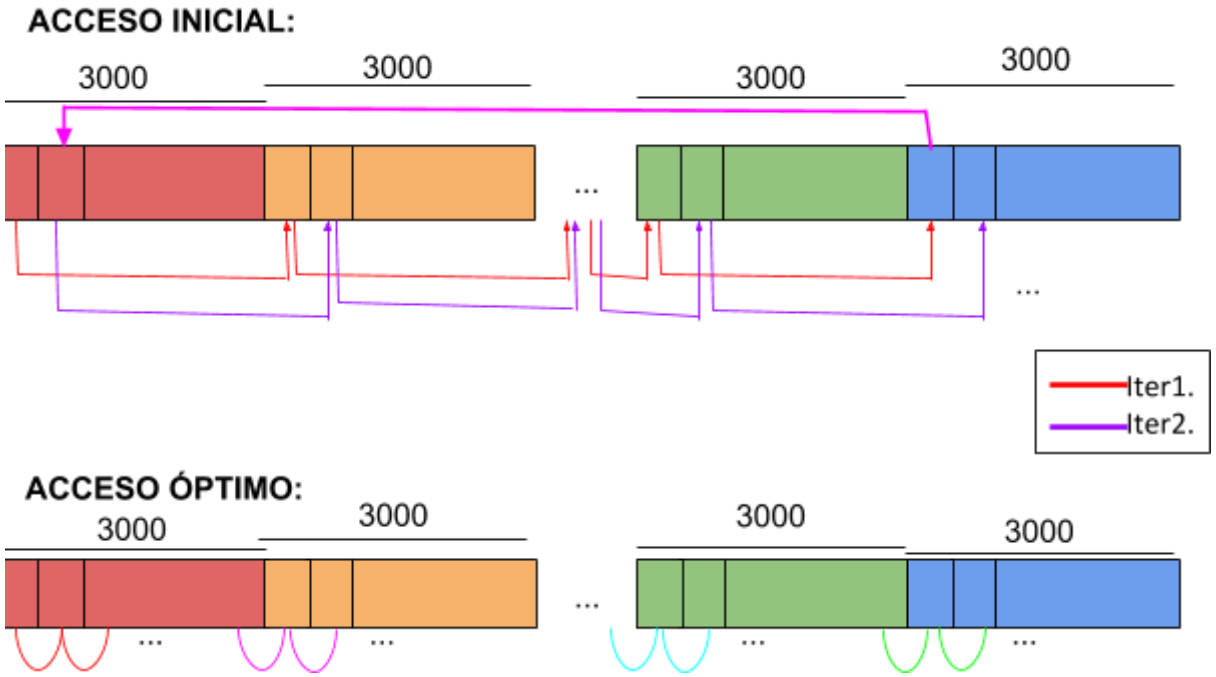
Ejecución inicial:

Comando: "*gcc -Ofast j.c -o jexe*" → "*perf stat ./jexe*"

IPC = 0,66
Inst = 244,67G
Tiempo = 134,38s
Checksum = 5271068

Optimizaciones Single Thread

1) Acceso Óptimo a la matriz. Centrándonos en el bucle con mayor complejidad temporal (N^3), observamos que el acceso a la matriz que se hacía en su interior, era MUY ineficiente en cuanto a conseguir acceder a los datos de forma consecutiva (Aprovechando al máximo el espacio de la Caché).



Tal y como se observa en el dibujo, en el acceso inicial se aprovechaba muy poco la memória de las caches, debido a que a cada paso, estas necesitaban volver a memória a por el siguiente bloque, y a su vez, necesitaban volver a cargar el bloque de forma repetida cada vez que cambiaba de iteración y volvía a empezar.

(Tal y como se observa en el dibujo, en el acceso inicial se aprovechaba muy poco la memória de las caches, debido a que a cada paso, necesitábamos el elemento de la siguiente columna (que no estaba contiguo en memoria), y se tenía que acceder a memoria a por el bloque de caché que contenía ese elemento. Esto se producía a cada iteración y desaprovechaba los datos presentes en caché.)

```
for (k = 0; k<N; k++)
        for (j = 0; j<N; j++)
                for (i = 0; i<N; i++)
                        if (Dist[i*N + j] > Dist[i*N + k] + Dist[k*N + j])
                                Dist[i*N + j] = Dist[i*N + k] + Dist[k*N + j];
```

```
for (k = 0; k<N; k++)
        for (i = 0; i<N; i++)
                for (j = 0; j<N; j++)
                        if (Dist[i*N + j] > Dist[i*N + k] + Dist[k*N + j])
                                Dist[i*N + j] = Dist[i*N + k] + Dist[k*N + j];
```

Una vez hecho el cambio, se observa que se hacen los accesos de forma consecutiva, de esta manera el programa no ha de ir sobrecargando bloques de memoria en caché que necesitará posteriormente para siguiente iteraciones, y ahora sí conseguimos aprovechar los datos presentes en la caché.

Estos cambios causan lo siguiente:

1) Las Instrucciones por Ciclo (IPC) aumenta de 0,66 ⇒ 1,85.
2) Las Instrucciones totales ejecutadas se reducen de 217,67 G ⇒ 190,34 G.

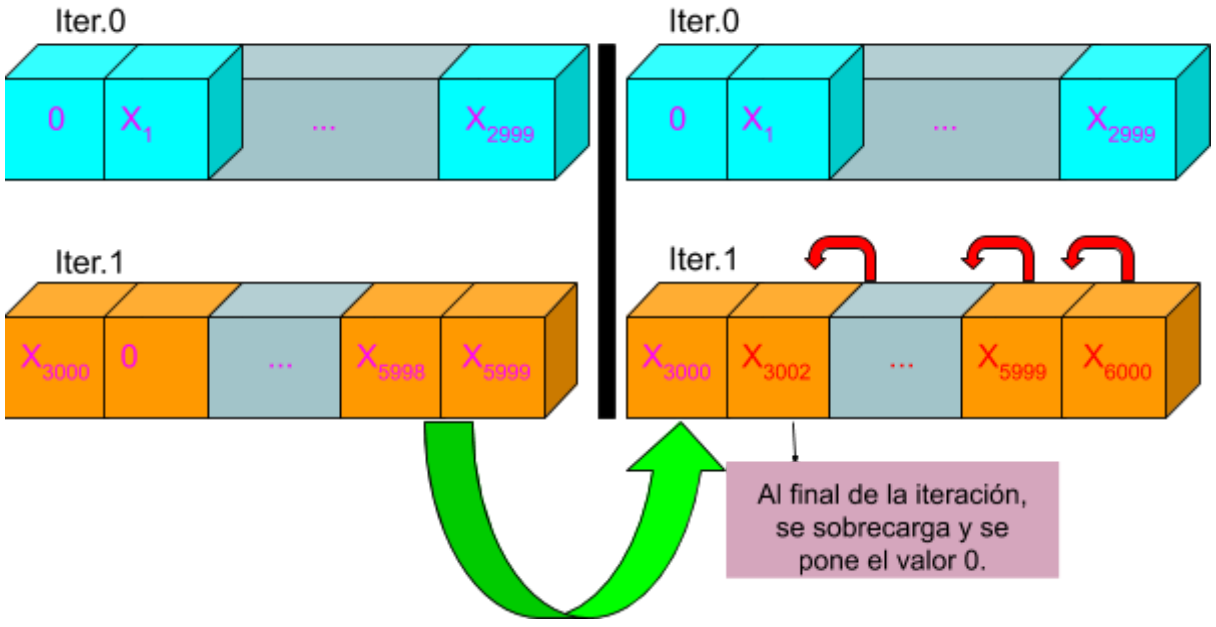
Esto provoca que:

- Tiempo: Se reduce el tiempo de ejecución de 134,39s ⇒ 37,42s .
- Speed Up: Conseguimos un Speed Up de **x3,59**.

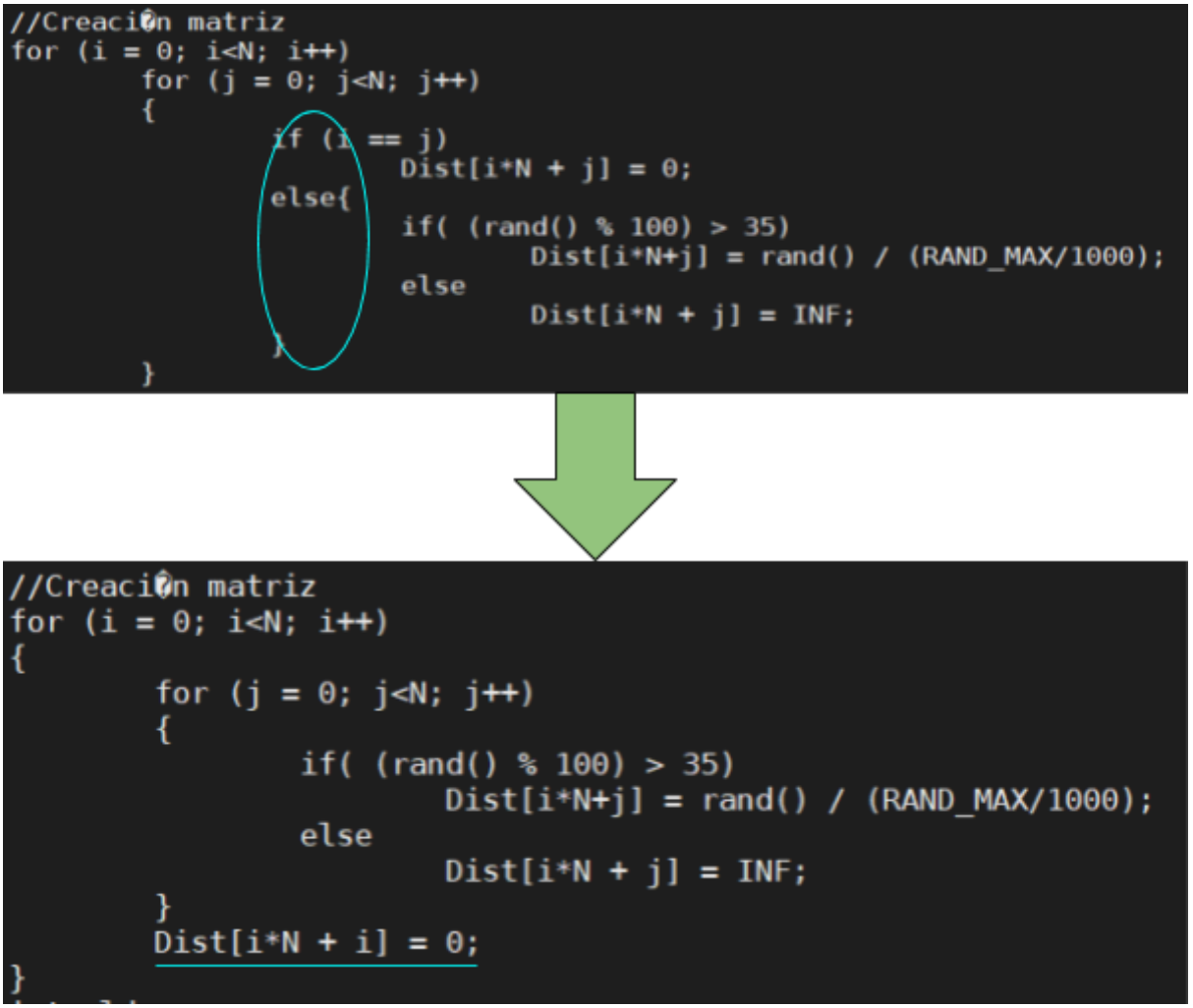
2) Cambio del Checksum: Inicialmente, vimos que en el primer bucle (De complejidad N^2), el "if" podría interrumpir al predictor de saltos a mitad de ejecución, lo cual debería perjudicar la ejecución del programa. Por ello decidimos aplicar esta solución para inicializar la matriz de la misma forma sin que ello provoque error en la inicialización de la matriz:

```
//Creación matriz
for (i = 0; i<N; i++)
        for (j = 0; j<N; j++)
        {
                if (i == j)
                        Dist[i*N + j] = 0;
                else{
                        if( (rand() % 100) > 35)
                                Dist[i*N+j] = rand() / (RAND_MAX/1000);
                        else
                                Dist[i*N + j] = INF;
                }
        }
```

Consecuencias del cambio: El checksum varía debido a que ahora se genera un valor "random" para los valores:



Tal y como se observa en el dibujo, la versión inicial, carga 0 en las matrices, sin embargo, nuestro cambio utiliza el siguiente valor generado por la semilla de random para poner valores a la **diagonal** (La cual debe tener el valor 0), sin embargo, solucionamos ese error cargando el valor 0 al acabar la iteración "j" del bucle.



Pensamos que este cambio produciría mejoras en el código, sin embargo, no se observan cambios resaltables en el rendimiento de la ejecución.

Estos cambios causan lo siguiente:

- Checksum: Pasa de valer 5.271.068 a valer 4.906.722

3) Vectorización: Se nos presentó una modificación del código (por parte del profesor) que debería conseguir que este código en versión Single Thread pudiera ejecutar Operaciones Vectorizadas (SIMD).

Esta modificación pasaba por:

- Eliminar la **escritura condicional** de manera que el compilador pueda vectorizar asegurando que el resultado será correcto. Para ello (al no poder eliminar totalmente el condicional). *(Para la versión MultiThread necesitamos que ella dependa de variables de cada Thread de forma **privada**.)*

  La modificación requiere que tengamos que hacer la siguiente modificación.

- La creación de variables que cada Thread pueda consultar de forma única.

Sin embargo, esta modificación no ha sido algo sencillo de implementar. Por ello hemos pasado por una serie de errores y falsas conclusiones que hemos tenido que resolver antes de llegar a la versión final. Pasaremos por ellas y explicaremos los fallos/errores cometidos en ellos y cómo se han resuelto.

Al hacer la primera versión del nuevo código, creamos las variables **fuera** del triple bucle for:

```
int old;
int new;
for (k = 0; k<N; k++)
{
        for (i = 0; i<N; i++)
        {
                for (j = 0; j<N; j++)
                {
                        old = Dist[i*N + j];
                        new = Dist[i*N + k] + Dist[k*N + j];
                        if (old > new)
                                old = new;
                        Dist[i*N + j] = old;
                }
        }
}
```

Sin embargo, al mostrar los resultados de la ejecución, el programa mostraba los siguientes resultados del tiempo de la ejecución:

1) El IPC pasa de 1,85 a 2,11 (Mejora).
2) Las Instrucciones Totales ejecutadas pasan de 190,34G a 325,47G.

Ello provoca que:

- Tiempo: Cambie 37,42s ⇒ 54,26s.
- Speed Up: Baje a x0,69.

(Estos cambios no se hacen efectivos ya que seguimos intentando encontrar una solución óptima)

Al observar el código ensamblador:

```
 0,00          add    -0x20(%rsp),%rax
               nop
 3,30    258:  movss  (%rcx,%rbx,4),%xmm0
13,77          addss  (%rdi,%r8,1),%xmm0
 8,34          cvttss2si (%rcx),%r10d
12,70          cvttss2si %xmm0,%edx
 1,28          pxor   %xmm0,%xmm0
 3,50          cmp    %r10d,%edx
13,69          cmovg  %r10d,%edx
 1,16          add    $0x4,%rcx
28,35          cvtsi2ss %edx,%xmm0
12,12          movss  %xmm0,-0x4(%rcx)
 0,44          cmp    %rax,%rcx
 1,26          jne    258
 0,01        ↑ jmp    224
               nop
```

Esta parte (Que representa el bucle más interno), hace 3 Loads y 1 Store (Normal tal y como tenemos el código), se destaca la instrucción “cvttss2si” y “cvttsi2ss”, las cuales son conversiones de números Float a Int y de Int a Float respectivamente.
Estas instrucciones son bastante complejas y hacen al compilador tardar más. Según nuestro razonamiento, son las causantes de que el número de instrucciones aumente considerablemente en el programa.
Una vez vista la conversión, y con ayuda del profesor, concluimos los siguientes errores:

a) Utilizar variables de tipo **entero** para guardar datos de tipo **flotante** supone pérdida de precisión en los valores (No muy relevante para este caso) y un coste adicional que al principio se subestima ya que no se le da demasiada importancia pero realmente la tiene.
b) Crear las variables dentro del bucle más interno, ya que de esta forma aseguramos no interactuar con valores guardados de iteraciones anteriores.

Ello nos lleva al siguiente código modificado:

```
for (k = 0; k<N; k++)
{
        for (i = 0; i<N; i++)
        {
                for (j = 0; j<N; j++)
                {
                        float old;
                        float new;
                        old = Dist[i*N + j];
                        new = Dist[i*N + k] + Dist[k*N + j];
                        if (old > new)
                                old = new;
                        Dist[i*N + j] = old;
                }
        }
}
```

Procedemos a ejecutarlo y observamos que los nuevos resultados que nos dan:

```
CheckSum: 4906722.00000

 Performance counter stats for './jexe2':

        35.020,64 msec task-clock                #    1,000 CPUs utilized
               15      context-switches          #    0,000 K/sec
                1      cpu-migrations            #    0,000 K/sec
            3.905      page-faults               #    0,112 K/sec
   96.207.274.295      cycles                    #    2,747 GHz                      (83,33%)
   14.102.503.967      stalled-cycles-frontend   #   14,66% frontend cycles idle     (83,33%)
   11.512.517.269      stalled-cycles-backend    #   11,97% backend cycles idle      (66,67%)
  190.285.361.795      instructions              #    1,98  insn per cycle
                                                 #    0,07  stalled cycles per insn  (83,34%)
   27.305.292.377      branches                  #  779,691 M/sec                    (83,34%)
       13.283.732      branch-misses             #    0,05% of all branches          (83,33%)

      35,031873811 seconds time elapsed

      35,010026000 seconds user
       0,011000000 seconds sys
```

A pesar de haber hecho las modificaciones, lo que conseguimos es:

Sin embargo, al mostrar los resultados de la ejecución, el programa mostraba los siguientes resultados del tiempo de la ejecución:

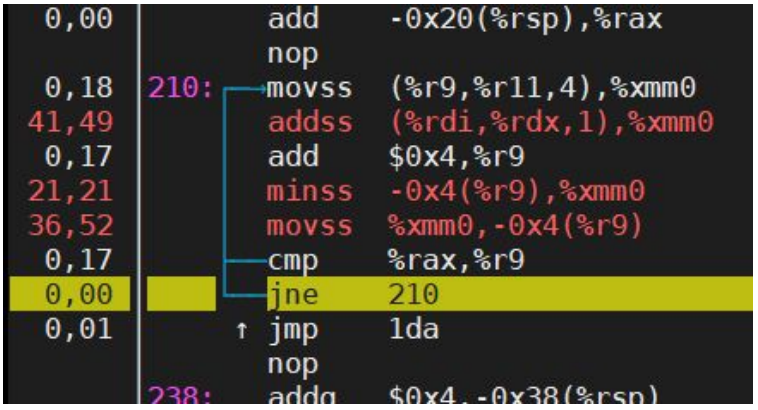
1) El IPC pasa de 1,85 a 1,98 (Mejora).
2) Las Instrucciones Totales ejecutadas pasan de 190,34G a 190,29G.

Ello provoca que:

- Tiempo: Cambia 37,42s ⇒ 35,01s.
- Speed Up: Sube a x1,06.

(Estos cambios no se hacen efectivos ya que seguimos intentando encontrar una solución óptima)

Lo que implica que mejoramos el tiempo en base al inicial pero **no de forma correcta**, debido a que el programa NO utiliza instrucciones SIMD:



Nos indica que sigue haciendo 3 Load y 1 Store (Normal) pero el compilador NO efectúa operaciones SIMD (Se ve de forma clara porque las instrucciones son “ss” y no “ps”). Por ello hemos de plantearnos cómo hacer para que el compilador nos permita la vectorización y para ello hemos de entender por qué no es capaz de vectorizar.

Este planteamiento pasa por entender qué instrucciones pueden hacer que el compilador desconfíe de la posibilidad de vectorizar el código.

```
for (k = 0; k<N; k++)
{
        for (i = 0; i<N; i++)
        {
                for (j = 0; j<N; j++)
                {
                        float old;
                        float new;
                        old = Dist[i*N + j];
                        new = Dist[i*N + k] + Dist[k*N + j];
                        if (old > new)
                                old = new;
                        Dist[i*N + j] = old;
                }
        }
}
```

Partiendo del código tenemos:

a) Las instrucciones de creación de las variables “old” y “new”, las cuales se guardarán en registros y no deberían representar un problema para proceder a la vectorización del compilador.
b) Carga a la variable “old” un valor de memória (El cual tiene una dependencia directa del valor “j” con el cual iteramos en el bucle), según esto, no debería haber problemas al vectorizar ya que cada iteración cogerá un valor diferente que no depende de otro.
c) Carga a la variable “new” 2 valores de memória:
    i) Un valor que **no** depende del valor “j” sinó de los valores con los que iteramos los bucles externos.
    ii) Un valor que depende de “j”, por lo tanto no debería afectar a la vectorización.
d) Una comparación que solo afecta a las variables creadas por esa iteración. No representa ningún problema al Vectorizar.
e) Guardar en memoria un valor que depende de la “j” i de la variable “i” con la que iteramos el bucle intermedio.

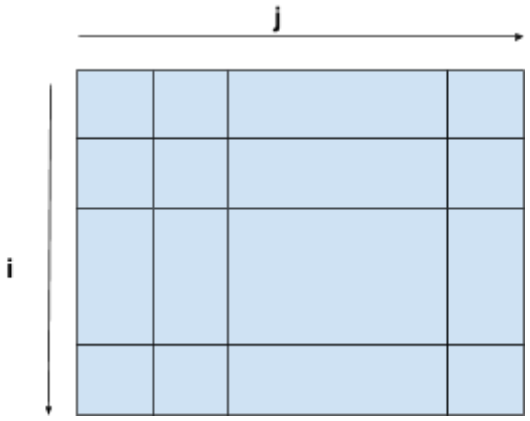
Una vez hecho esta observación, descartamos que a), b), c) y d) NO afectan a esta imposibilidad de vectorizar ya que son instrucciones de LOAD o condicionales de variables propias para cada iteración.
Algo debe pasar en el punto del código e), así que procedemos a analizar sus accesos a memoria:
Tal y como se expuso en c.i), su acceso no depende de la variable “j”, lo que nos hace pensar en un inicio que dicho acceso puede consultarse una única vez (fuera del bucle for de “j”) y ahorrarse N instrucciones LOAD.

Pese a esta pérdida, inicialmente no le dimos mucha importancia, al consultar con el profesor y hacer un análisis más profundo, nos dimos cuenta de que realmente **sí** que evita que el compilador pueda vectorizar, y ahora trataremos de explicar de qué manera lo impide:

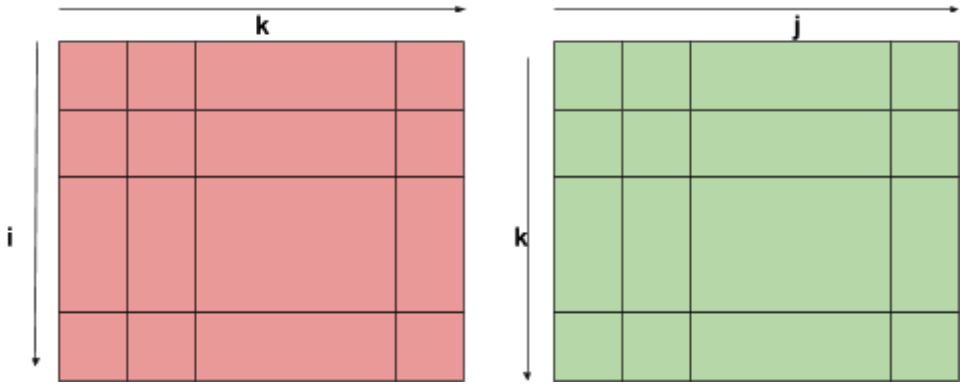
Para vectorizar, el ensamblador ha de estar seguro de que el resultado que proporciona el código que optimiza será correcto, a partir de eso, debemos analizar de forma detallada los accesos que hacen a memória las variables y si puede haber alguna situación conflictiva. Para ello nos fijamos en los accesos del código a memória.

Tenemos 3 Load y 1 Store en el bucle más interno, para que no se pueda vectorizar, tiene que haber un problema con la escritura (Store) vectorizada. Para ello hemos de analizar sobre el "tipo" de matrices sobre el que itera cada LOAD y STORE:

-Las instrucciones **" *old = Dist[i*N+j]* "** y **" *Dist[i*N+j] = old* "** actúan sobre la misma posición en cada iteración del bucle "j". Su recorrido de la matriz es:



-Las instrucciones **" *new* = *Dist[i*N+k]* + *Dist[k*N+j]* "** actúan diferentes posiciónes en cada iteración del bucle "j". Sus recorridos son:



Para acabar de entender el concepto por el cual no vectoriza, hemos de ver qué situación problemática con la instrucción de "Store" provoca el conflicto. Estos conflictos pueden ser dados cuando:

1)Se cumple que (i=k): Los cuales no son "críticos" ya que iteramos sobre la variable "j" ⇒ En caso que sobrecargar un valor en memória, tendremos margen suficiente para leerlo una vez terminemos de iterar con el bucle "j" → No hay problemas.

2)Se cumple que (j=k): Son los conflictos críticos ya que la sobrecarga de un valor en el "Store" que se tenga que leer a lo largo de las iteraciones, al estar vectorizado, no podrá ser leída de nuevo y por ello, habrá actuado con un valor **no actualizado**. → Puede haber problemas.

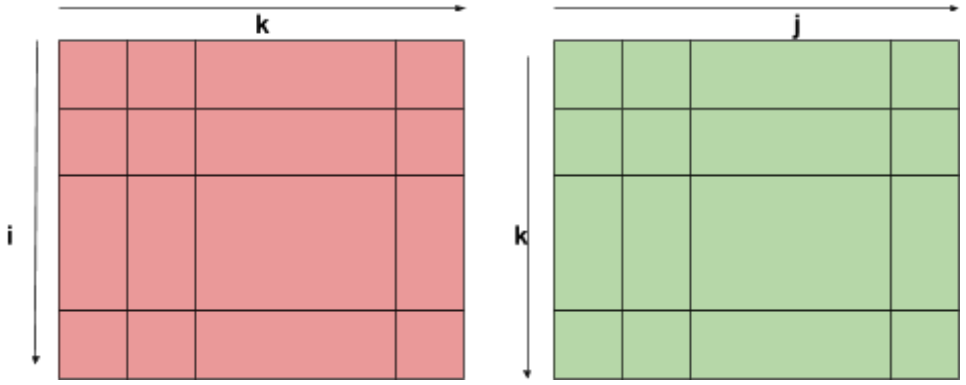
Es complejo de explicar, pero lo acabamos planteando de la siguiente forma:
Supongamos que la ejecución va por los valores k=498, i=9, j=496 (Por poner un ejemplo). Analizaremos ahora qué pasaría <u>si el compilador pudiera vectorizar</u>:

Al poder vectorizar, hacemos 4 iteraciones en una (Ya que podemos leer y escribir los valores de 4 instrucciones a la vez y operar con ellos).
Por ello:

K = 498					
i = 9					
Inst.→Inst. Assembly		j = 496	j = 497	j = 498	j = 499
old=Dist[i*N+j]	Load	D[9*N+496]	D[9*N+497]	D[9*N+498]	D[9*N+499]
new= Dist[i*N+k] + Dist[k*N+j]	Load	D[9*N+498]	D[9*N+498]	D[9*N+498]	D[9*N+498]
	Load	D[498*N+496]	D[498*N+496]	D[498*N+496]	D[498*N+496]
Dist[i*N+j]=old	Store	D[9*N+496]	D[9*N+497]	D[9*N+498]	D[9*N+499]

La explicación de porqué el compilador NO vectoriza se debe a que en algunos casos como el expuesto en la tabla, el compilador **NO** puede leer un valor correcto para j=499 debido a que j=498 podría haber cambiado el valor que ha de leer (Decimos podría ya que depende del condicional). En este tipo de casos el compilador NO puede asegurar que se lean valores correctos para las demás "iteraciones" de la variable j. Y por ello, ello pueda resultar en resultados erróneos.

Esto explica porqué el compilador **NO** vectoriza, pero si lo analizamos más profundamente desde una visión genérica de la funcionalidad del código, básicamente **solo** escribiremos un valor nuevo si **lo que sumamos en "new"** es mayor que lo que hay en "old" en este caso, cuando j=k, significa que: **El valor que haya en " *Dist[k*N+j]* " sea >0, siempre escribiremos un valor nuevo.**
Ahora, recuperando el dibujo anteriormente expuesto en el que explicamos los accesos de la instrucción :



Nos fijamos que en el caso de **j = k**, la matriz de ***Dist[k*N+j]*** solo es afectada en la **diagonal**. Y si recordamos la estructura de una matriz, en Floyd los costes de ir desde un punto hacia sí mismo son de **0** en nuestro caso. Por lo que:
El valor que haya en " *Dist[k*N+j]* " NUNCA va a ser >0 ya que la Diagonal de esta matriz representa los costes de un punto hacia sí mismo.

Con esto sabido, ahora hemos de buscar soluciones (si las hay) para que hacer que el compilador sepa que puede vectorizar sin problemas ya que esta vectorización está "controlada" por tal y como definimos nuestra matriz.
La solución a este problema es sencilla, ya que sabemos que el "Load" que nos está causando problemas no va a cambiar de valor durante la ejecución del bucle más interno. Dicho esto, procedemos a hacer el Load **fuera** del bucle más interno y guardamos su valor en un registro:

```
for (k = 0; k<N; k++)
{
        for (i = 0; i<N; i++)
        {
                float t = Dist[i*N + k];
                for (j = 0; j<N; j++)
                {
                        float old;
                        float new;
                        old = Dist[i*N + j];
                        new = t + Dist[k*N + j];
                        if (old > new)
                                old = new;
                        Dist[i*N + j] = old;
                }
        }
}
```

Y ahora, con esta última versión, la ejecución SI nos dá el resultado que queríamos ya que al mirar el ensamblador, podemos observar cómo las instrucciones ejecutadas SON vectoriales.

```
 0,00          nop
 0,80  150:    movups  (%r8,%rdx,1),%xmm0
87,04          movups  (%rcx,%rdx,1),%xmm3
 1,20          addps   %xmm2,%xmm0
 2,22          minps   %xmm3,%xmm0
 1,65          movups  %xmm0,(%rcx,%rdx,1)
 0,67          add     $0x10,%rdx
 0,90          cmp     %rdi,%rdx
 1,38          jne     150
 0,00          cmp     %r12d,%ebx
 0,00          je      1d9
```

Y el resultado se ve en la siguiente ejecución:

```
CheckSum: 4906722.00000

 Performance counter stats for './jexe':

         19.681,54 msec task-clock                #    0,999 CPUs utilized
                15      context-switches          #    0,001 K/sec
                 1      cpu-migrations            #    0,000 K/sec
             3.317      page-faults               #    0,169 K/sec
    53.653.760.521      cycles                    #    2,726 GHz                      (83,33%)
    32.083.931.715      stalled-cycles-frontend   #   59,80% frontend cycles idle     (83,34%)
    31.840.863.798      stalled-cycles-backend    #   59,35% backend cycles idle      (66,67%)
    55.290.352.145      instructions              #    1,03  insn per cycle
                                                  #    0,58  stalled cycles per insn  (83,34%)
     7.050.090.142      branches                  #  358,208 M/sec                    (83,33%)
        13.348.400      branch-misses             #    0,19% of all branches          (83,34%)

      19,699975870 seconds time elapsed

      19,661033000 seconds user
       0,021000000 seconds sys
```

Al mostrar los resultados de la ejecución, obtenemos los siguientes cambios:

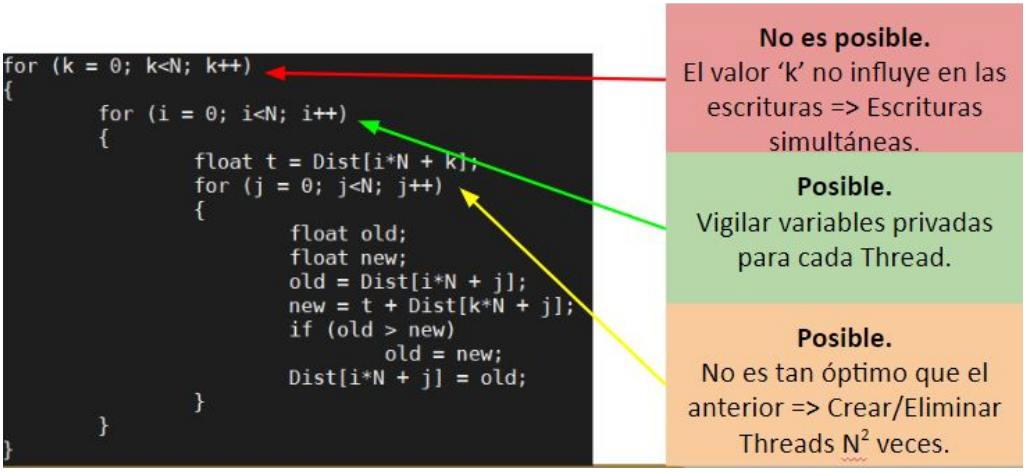
1) El IPC pasa de 1,85 a 1,03 (Empeora).
2) Las Instrucciones Totales ejecutadas pasan de 190,34G a 55,29G (Mejora muy notoriamente).

Ello provoca que:

- Tiempo: Cambie 37,42s ⇒ 19,66s.
- Speed Up: Suba a x1,90.

Optimizaciones a Nivel Multi Thread

Para comenzar con las optimizaciones Multi Thread, hemos de localizar desde qué bucle es posible o no realizar la creación y ejecución de la zona con varios Threads:



Tras analizarlo, concluimos lo siguiente:

1) La creación de Threads en el 1er bucle for, cuyos threads iterarían sobre una variable "k" produce problemas debido a que las variables "i" y "j" coinciden entre Threads y eso hace que la instrucción de escritura "*Dist[i*N + j] = old;"fuera accedida por todos los Threads a la vez.*
2) En el 2o bucle for (El que itera con la variable "i") es el más óptimo en cuanto a ser el más externo posible (Cuanto más externo, menos Threads se crean y destruyen innecesariamente) y no presenta errores en la ejecución.
3) El 3er bucle for es válido para proceder a una ejecución Multi Thread pero concluimos que la creación y destrucción constante de Threads, empeoraría el tiempo innecesariamente.

Por ello, decidimos aplicar la siguiente directiva a nuestro código:

```
for (k = 0; k<N; k++)
{
        #pragma omp parallel for private(i,j)
        for (i = 0; i<N; i++)
        {
                float t = Dist[i*N + k];
                for (j = 0; j<N; j++)
                {
                        float old;
                        float new;
                        old = Dist[i*N + j];
                        new = t + Dist[k*N + j];
                        if (old > new)
                                old = new;
                        Dist[i*N + j] = old;
                }
        }
}
```

Que consta de la directiva “parallel for” básica para bucles for con un añadido especificando que las variables “i” y “j” son privadas para cada Thread.

En base a estas modificaciones, procedemos a ejecutar la versión Multi Thread pero **con diferentes versiones**, ya que cuando nosotros hicimos la experimentación a nivel de Multi Thread, aún no habíamos acabado de optimizar la versión actual final de Single Thread con Vectorización, por ello, vamos a analizar **3 casos diferentes**:

Las siguientes ejecuciones están ejecutadas en la cola de Trabajos de “aomaster”:

- 6 Cores.
- 2 Threads/Core.

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                24
On-line CPU(s) list:   0-23
Thread(s) per core:    2
Core(s) per socket:    6
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 44
Model name:            Intel(R) Xeon(R) CPU           E5645  @ 2.40GHz
Stepping:              2
CPU MHz:               2393.952
BogoMIPS:              4787.90
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              12288K
NUMA node0 CPU(s):     0-5,12-17
NUMA node1 CPU(s):     6-11,18-23
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdp
```

1) Multi Thread del Caso Erróneo:

```
int old;
int new;
for (k = 0; k<N; k++)
{
        for (i = 0; i<N; i++)
        {
                for (j = 0; j<N; j++)
                {
                        old = Dist[i*N + j];
                        new = Dist[i*N + k] + Dist[k*N + j];
                        //if (Dist[i*N + j] > Dist[i*N + k] + Dist[k*N + j])
                        if (old > new)
                                old = new;
                        Dist[i*N + j] = old;
```

Pensando erróneamente que el código aplicado de esta forma optimiza (más tarde nos dimos cuenta de nuestro error), basamos la ejecución Multi Thread en base a este código:

```
int old;
int new;

for (k = 0; k<N; k++)
{
        #pragma omp parallel for private(old,new,i,j)
        for (i = 0; i<N; i++)
        {
                for (j = 0; j<N; j++)
                {
                        old = Dist[i*N + j];
                        new = Dist[i*N + k] + Dist[k*N + j];
                        //if (Dist[i*N + j] > Dist[i*N + k] + Dist[k*N + j])
                        if (old > new)
                                old = new;
                        Dist[i*N + j] = old;
```

En base a esta ejecución (La cual empeoraba la ejecución Single Thread), decidimos realizar la ejecución con diferentes Threads:

Valor	Single Thread	2 Threads	4 Threads	6 Threads	12 Threads
Tiempo (seg)	55’81	27’92	14’38	10’48	8’15
IPC por Thread	2’11	2’16	2’14	2’08	1’43
IPC_{Total}	2’11	4’32	8’56	12’48	17’16
Uso de CPU_s	1’00	1’99	3’94	5’87	11’64

Con estos resultados, vemos una mejora ideal del tiempo. Esto se debe a que esta versión (en comparación al resto de versiones que veremos) no está suficientemente optimizada a nivel Single Thread. Ya que de ser así, el Speed Up no llegaría a ser ideal.

El hecho que al ejecutar 2 Threads por core obtengamos una mejora ideal ya nos dá una pista de que esta versión tiene un problema con la versión Single Thread que se “compensa” en la versión de 12 Threads. Generalmente, la versión de doble thread por core no debería optimizar en algoritmos como el nuestro.

2) MultiThread con Base NO Vectorizada.

Partiendo de la siguiente versión del código:

```
for (k = 0; k<N; k++)
        for (i = 0; i<N; i++)
                for (j = 0; j<N; j++)
                        if (Dist[i*N + j] > Dist[i*N + k] + Dist[k*N + j])
                                Dist[i*N + j] = Dist[i*N + k] + Dist[k*N + j];
```

Pensamos en hacer una ejecución Multi thread. La cual aplicando el mismo pragma:

```
for (k = 0; k<N; k++)
        #pragma omp parallel for private(i,j)
        for (i = 0; i<N; i++)
                for (j = 0; j<N; j++)
                        if (Dist[i*N + j] > Dist[i*N + k] + Dist[k*N + j])
                                Dist[i*N + j] = Dist[i*N + k] + Dist[k*N + j];
```

Obtenemos los siguientes resultados:

Valor	Single Thread	2 Threads	4 Threads	6 Threads	12 Threads
Tiempo (seg)	37’42	19’90	10’25	7’24	4’65
IPC por Thread	1’85	1’85	1’81	1’79	1’52
IPC_{Total}	1’85	3’70	5’43	10’74	18’24
Uso de CPU_s	1’00	1’985	3’883	5’816	11’369
Speed Up	x1	x1’88	x3’65	x5’17	x8’05

Podemos observar que al aumentar el número de Threads, conseguimos un Speed Up aproximado al ideal. Esta aproximación se debe a la creación/gestión/destrucción de Threads y que en algunas ocasiones, la caché L3 retire los datos que otro Thread necesita para su ejecución y los tenga que ir a buscar posteriormente de nuevo (esta diferencia resalta más a medida que nos aproximamos a los 6 Threads). Por último, al ejecutar 2 Thread por core (12 Threads) observamos que además de

interrumpirse en la caché L3, cada Thread del core lucha por tener sus datos en caché L1 y L2 y ello hace que se produzcan más fallos a memória.

3) MultiThread con Base Vectorizada.

Partiendo de la versión explicada con la Vectorización, decidimos aplicar nuestra versión multiThread para comparar qué le ocurren a los tiempos con diferentes Threads cuando **cada thread** puede vectorizar su propia ejecución:

```
for (k = 0; k<N; k++)
{
        #pragma omp parallel for private(i,j)
        for (i = 0; i<N; i++)
        {
                float t = Dist[i*N + k];
                for (j = 0; j<N; j++)
                {
                        float old;
                        float new;
                        old = Dist[i*N + j];
                        new = t + Dist[k*N + j];
                        if (old > new)
                                old = new;
                        Dist[i*N + j] = old;
                }
        }
}
```

Esta optimización nos lleva a los siguientes resultados:

Valor	Single Thread	2 Threads	4 Threads	6 Threads	12 Threads
Tiempo (seg)	19’70	10,88	6’95	6’95	6’77
IPC por Thread	1’03	1’09	0’8	0’56	0’3
IPC_{Total}	1’03	2’18	2’4	3’36	3’6
Uso de CPU_s	1’00	1’973	3’888	5’813	11’55
Speed Up	x1	x1’99	x2’83	x2’83	x2’91

A partir de estos resultados, podemos ver que al aumentar a 2 Threads, mejora idealmente el Speed Up de la ejecución pero al aparecer más Threads, no obtenemos el beneficio ideal que podríamos esperar.

Esto se debe a que la versión Vectorizada del código, opera con varios datos a la vez. Por lo que más Threads supone que cada uno opere en un rango de memória y trabaje con varios datos a la vez:

Cuando se ejecutan 4 Threads, se hacen 4 iteraciones por Thread ⇒ 16 iteraciones

Todo ello, en el tiempo que una ejecución Single Thread sin Vectorizar haría 1 Iteración.

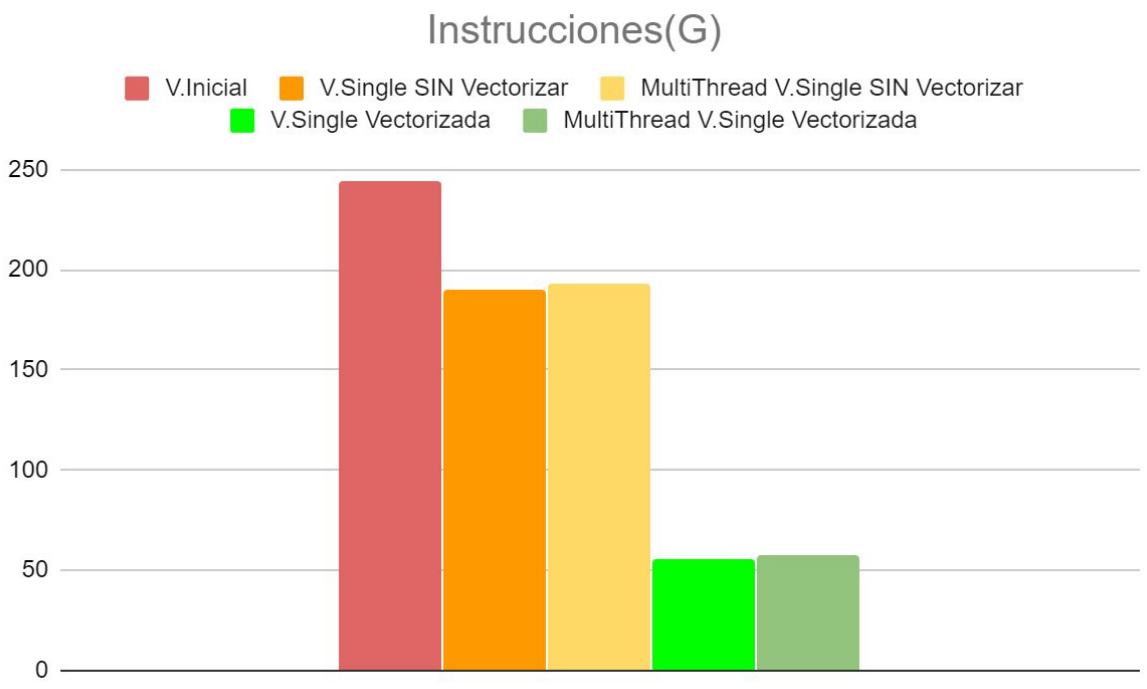
Al hacer más operaciones, se hace mayor uso de memória por Iteración y Thread, y contando los conflictos en caché que puede haber entre los Threads por querer acceder a diferentes zonas de memória a la vez (Y todas las zonas no caben en la caché) y la lentitud del acceso a memória comparado con la ejecución de los Threads, conlleva a tener que esperar a memoria. Sumado al hecho del gasto de creación y gestión de Threads, la mejora que conseguimos al aumentar el número de Threads se ve muy reducida.

Cuando subimos a 2 Threads por Core (12 Threads), tenemos el riesgo de que la caché de memoria compartida por los Threads sobrescribe los datos necesarios del otro Thread, de esta manera podemos explicar por qué casi no hay mejora respecto la versión con 6 Threads (Ya que la versión con 1 Thread por core, podrían tener conflictos en caché de niveles más altos. Pero al añadir un Thread en el mismo core, ahora esos problemas se pueden trasladar a conflictos en la misma caché **privada** del core).

Analisis de los resultados

Por último, procederemos a plasmar cada cambio que hemos hecho en el código y qué efecto ha tenido (Tiempo de ejecución, Instrucciones, IPC…) y explicaremos el porqué de estos cambios de forma resumida.
También explicaremos los diferentes Speed Up que obtenemos con cada versión (Ya que tenemos diferentes versiones) y expondremos cuál de ellos llega a una ejecución más rápida del código.

I-Instrucciones:



→ Partimos de la versión base (roja) con cerca de 250 G instrucciones a ejecutar.
→ Podemos ver que las optimizaciones hechas en la versión Single Thread SIN Vectorizar, consigue reducir este número en ~50 G instrucciones. Esto se debe a:

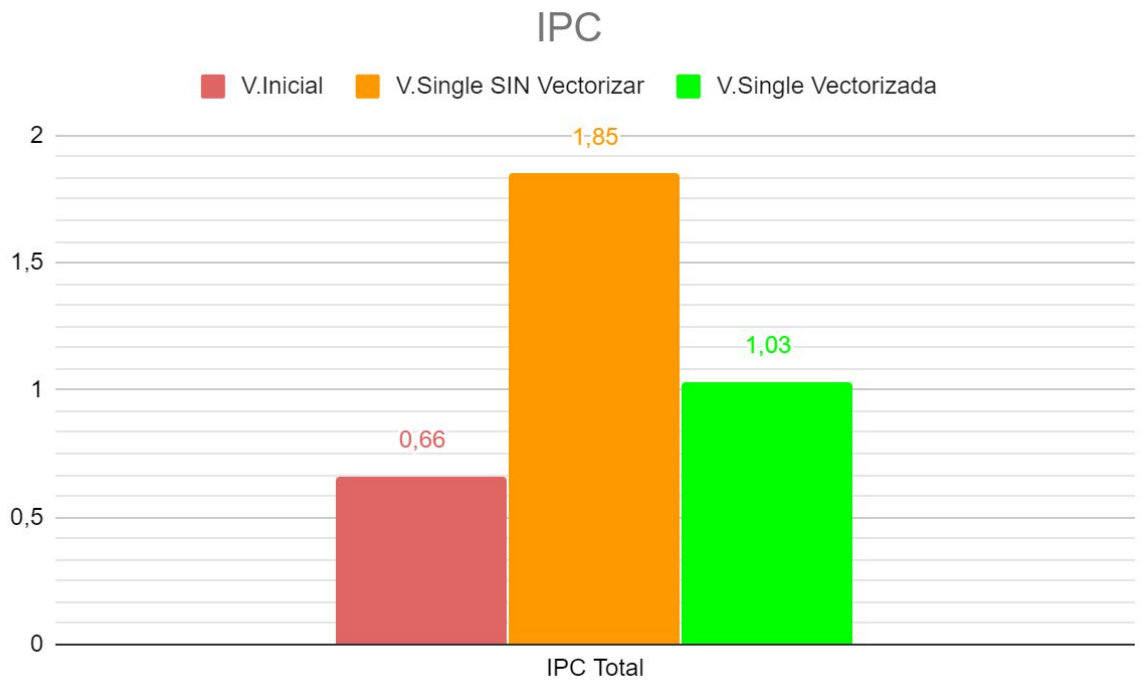
1) Los accesos a la memoria de la matriz son más óptimos (Aunque este cambio afecta mayormente a los fallos en caché).
2) En menor medida, la eliminación del condicional en el primer doble bucle for que elimina el condicional que hace que el predictor de saltos falle a mitad de la ejecución del bucle. Este cambio afecta a una complejidad menor del programa ya que solo dispone de una complejidad de N^2 (Respecto a la N^3 de la complejidad del programa, apenas se consiguen grandes cambios).

→ Posteriormente pasamos a la versión MultiThead SIN Vectorizar, se puede observar que la media de la ejecución con 2,4,6 y 12 Threads es un poco mayor a la ejecución Single Thread. Esto se debe a que el coste de creación, destrucción y coordinación entre Threads generan instrucciones extras al programa.
→ La versión Single Thread Vectorizada consigue reducir muy considerablemente las instrucciones ejecutadas. Básicamente, al conseguir vectorizar el Triple Bucle for, conseguimos reducir el número de instrucciones hasta 4 veces el número que conseguimos

con la Optimización SIN Vectorizar (Ya que los datos con los que trabajamos son de tipo Float y la Vectorización puede almacenar 4 valores y operar con 4 valores a la vez en el bucle).
→ Así mismo, la versión MultiThread Vectorizada, obtiene el mismo resultado con la penalización de la creación, destrucción y coordinación entre Threads (Tal y como pasaba en la ejecución MultiThead SIN Vectorizar).

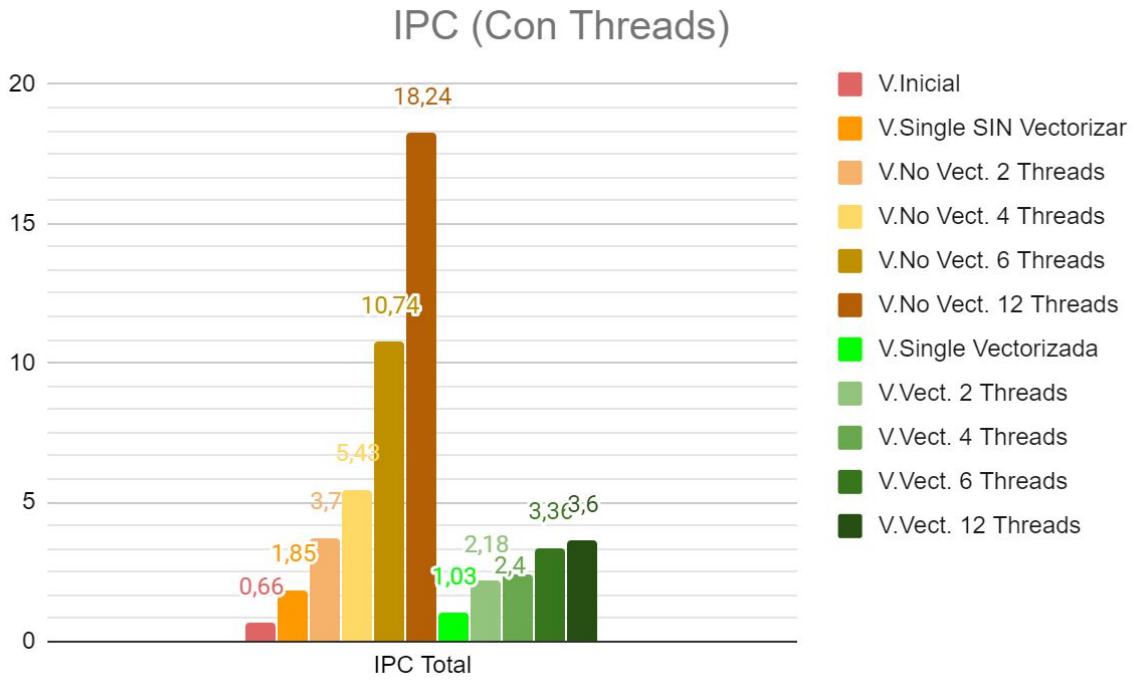
Cabe destacar que la versión Vectorizada, disminuye el número de instrucciones que ejecuta el programa **5 veces** respecto a la Versión Original.

II-Instrucciones Por Ciclo:

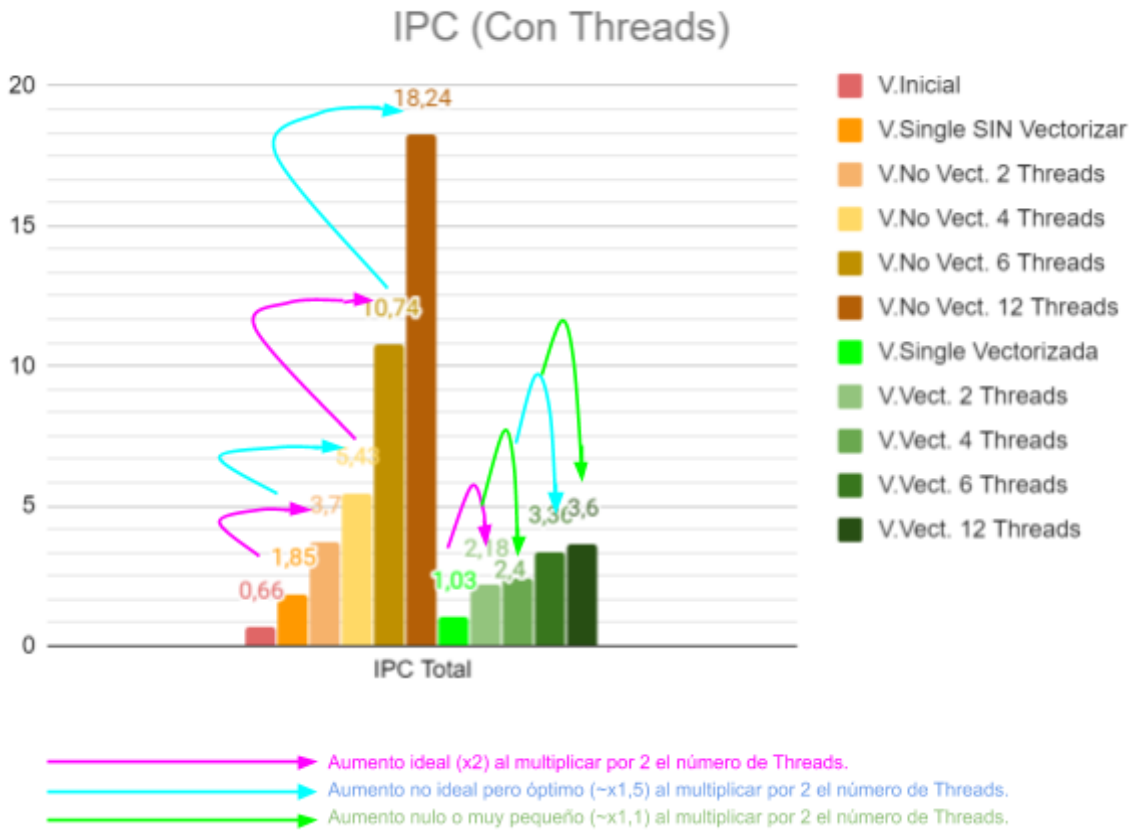


Aquí podemos ver cómo las versiones (Single Thread) de los diferentes códigos de nuestro programa van aumentando su IPC respecto la Versión Inicial.
Los cambios conseguidos en la Versión Sin Vectorizar permiten acceder a memória de forma más óptima y de esa manera, permitimos al programa seguir su ejecución sin depender tanto de esperar los datos cargados de memória.
Los cambios conseguidos en la Versión Vectorizada consiguen un IPC menor que la anterior, lo cual es “raro” desde el punto de vista de que la Versión Vectorizada consigue un Speed Up mayor que la Versión Sin Vectorizar con un IPC menor que el de esta. Pero no es tan raro si analizamos y explicamos que junto a este IPC menor, la Versión Vectorizada ejecuta hasta **4 veces menos instrucciones que la** Versión Sin Vectorizar.
Se podría traducir de forma que la Versión Vectorizada ejecuta instrucciones x0,55 veces más lenta que la Versión Sin Vectorizar **pero ejecuta 4 instrucciones a la vez** ⇒ x0,55 * 4 = x2,20 (Este valor es una **combinación** del cambio del IPC y del Nº de Instrucciones ejecutado en esta versión).

III-Instrucciones Por Ciclo (Con Threads):



Observamos la evolución de las Instrucciones Por Ciclo ejecutadas en las diferentes versiones. El aumento de IPC conforme se van añadiendo Threads a las versiones es:



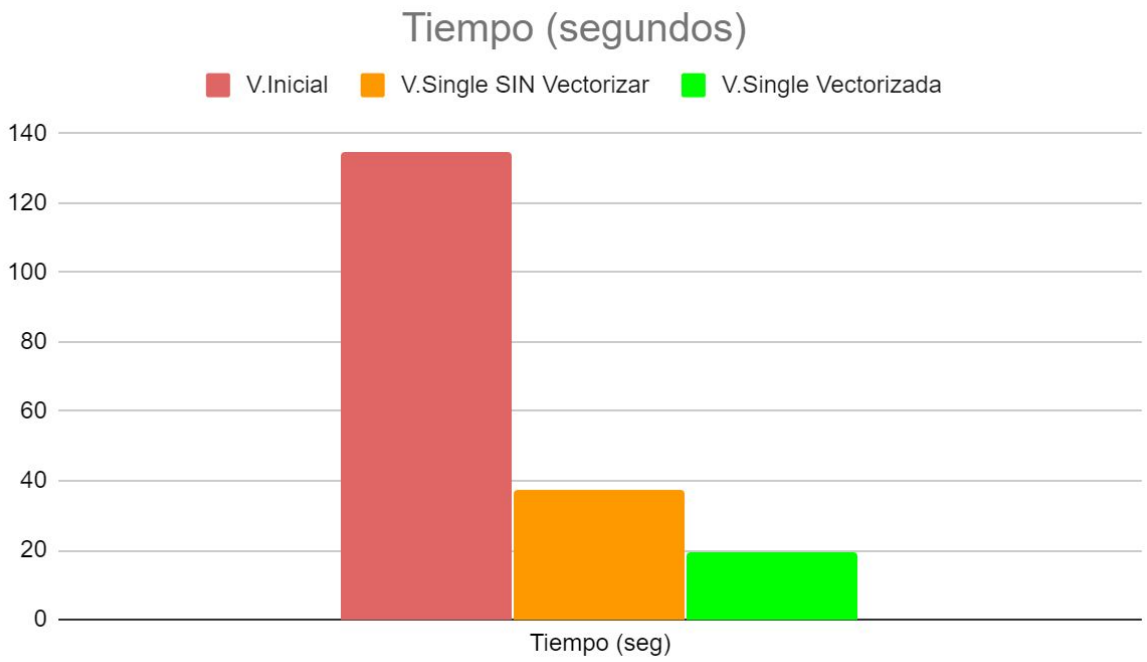
Cabe destacar que el IPC Total con Threads es calculado de la siguiente manera:
$IPC_{Total} = IPC_{De\ cada\ Thread} * \#Threads$.
Ahora evaluamos el IPC al añadir Threads en las versiones:
Para la Versión Sin Vectorizar:

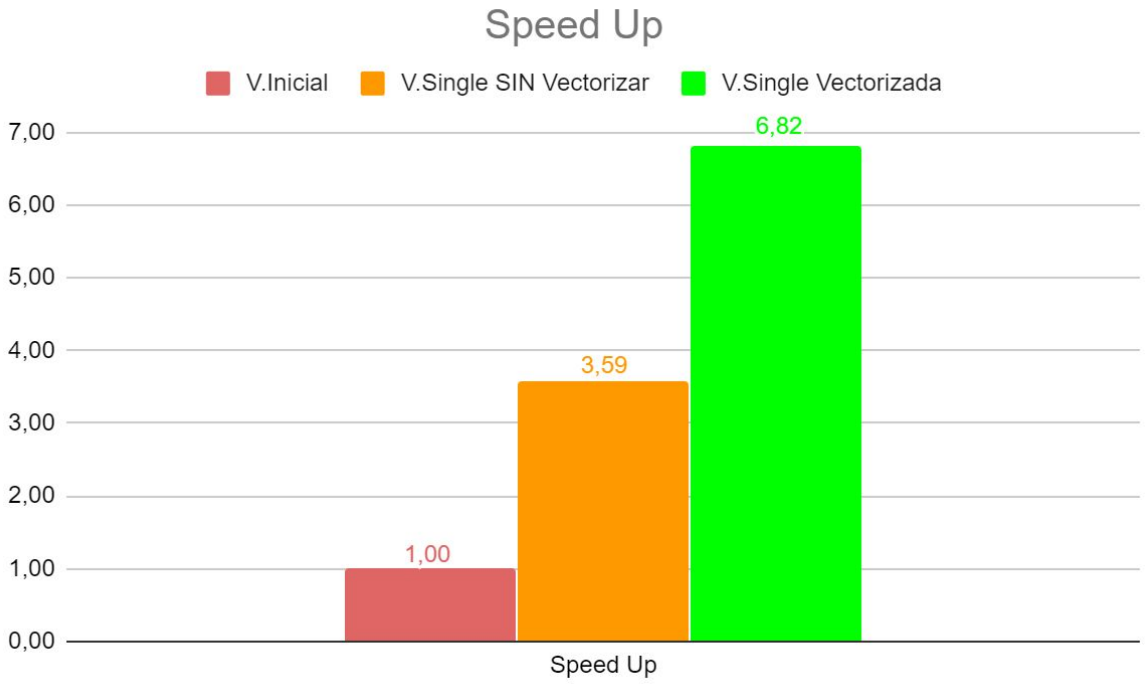
- Al pasar de: 1 Thread ⇒ 2 Threads y 4 Threads ⇒ 6 Threads conseguimos un aumento ideal del IPC, lo que indica que la repartición de memória entre Threads y la coordinación entre ellos se hace lo más óptimo posible.
- Al pasar de: 2 Threads ⇒ 4 Threads y 6 Threads ⇒ 12 Threads conseguimos un aumento óptimo del IPC, ello indica que:
    1) En el caso de 2 a 4 Threads, la repartición de memória entre los Threads no es equitativa y puede que haya algún Thread con más carga de trabajo que otro. Esto se arreglaría mirando más detalladamente qué ocurre en este caso y añadiendo algún pragma al paralelizar que repartiera mejor la memória entre los Threads.
    2) En el caso de 6 a 12 Threads, posiblemente ya no sea por la memória sinó por el hecho de que 2 Threads comparten una caché privada en la cual los Threads pueden sobreescribir el bloque de Datos que el otro Thread necesitará posteriormente y se molesten el uno al otro. Sin embargo, conseguir una mejora de (~x1,5) respecto la versión con 6 Threads es algo poco común en los programas (A no ser que dicho programa tenga un grado de paralelización MUY óptimo).

Para la Versión Vectorizada:

- Al pasar de: 1 Thread ⇒ 2 Threads conseguimos un aumento ideal del IPC, lo que al aumentar a 2 Threads, el hecho de que cada Thread tenga permitido vectorizar su trabajo y este trabajo sea la mitad del trabajo que se hacía con la versión Single Thread indican que la cantidad de trabajo que hay (Tamaño de datos a procesar) no se vea limitado (aún) por el tiempo de acceso a memória.
- Al pasar de 4 Threads ⇒ 6 Threads conseguimos un aumento óptimo del IPC, lo que nos indica que para este caso, la repartición de datos sumada a la vectorización empieza a tener un cuello de botella en la memoria, ya que esta se ve sobreescrita por otros Threads que han de acceder a otro bloque diferente de datos.
- Al pasar de 2 Threads ⇒ 4 Threads y 6 Threads ⇒ 12 Threads conseguimos un aumento nulo o muy pequeño del IPC, esto indica que la creación de Threads y el trabajo en paralelo de ellos se ve MUY penalizada por las constantes sobreescrituras de caché que hacen los Threads y ello compensa negativamente toda lo que se puede mejorar añadiendo en el código más Threads.

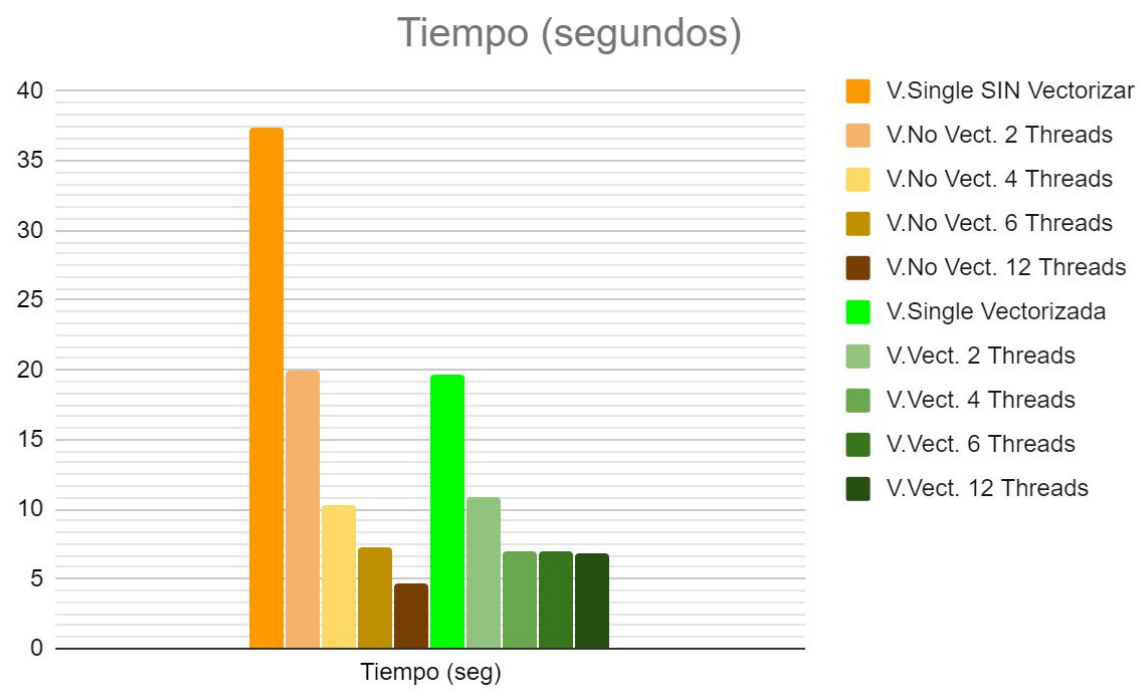
IV-Tiempo y Speed Up (Versiones Sin Threads):



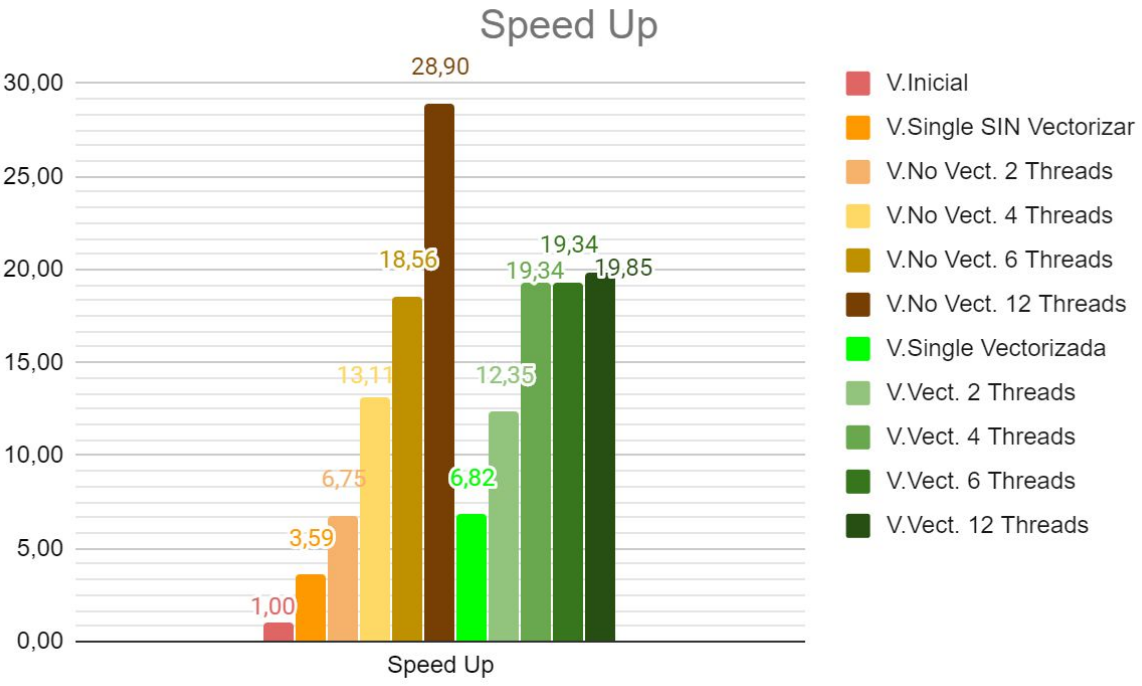


Por último, las combinaciones de IPC e Instrucciones junto a las explicaciones dadas anteriormente sobre el porqué cambian en cada caso, dá lugar a los siguientes tiempos y el siguiente Speed Up.
Destacamos que el mejor Speed Up para la versión Single Thread, lo conseguimos aplicando la vectorización. Ya que gracias a Vectorizar, podemos hacer 4 veces menos instrucciones y por ello aumentamos mucho nuestro rendimiento (Incluso aunque otros factores como el IPC disminuyan su valor).

V-Tiempos y Speed Up (Versiones Con Threads):



(Recordar que la Versión Inicial era de 134 seg.No está incluido en la gráfica ya que quitaría precisión a los demás datos).



Al aplicar Threads, observamos que la versión Vectorizada (que es la que mejor Speed Up conseguía a nivel Single Thread), consigue un buen rendimiento al duplicar los Threads pero a partir de ahí **se queda estancada**. Esto se debe a que la memoria compartida de Threads se empieza a solapar, haciendo que haya mayor espera por datos de memoria.

Por último destacar que aunque la versión SIN vectorizar no supera a nivel Single Thread la versión Vectorizada, esta lo hace al aplicar sobre ella 12 Threads, consiguiendo **el mejor Speed Up de todas las ejecuciones del código.** Esto se debe a que la Versión Sin Vectorizar, al añadir el nivel de MultiThread, "compensa" el hecho de que no se vectorize el código.

Que una versión con 12 Threads (Y 2 Threads por core) obtenga una mejora destacable en un código (cuyo Bottleneck absoluto y exclusivo NO sea la paralelización con Threads) nos indica que es muy probable que la versión Single Thread sobre la que basamos la paralelización NO sea lo más óptima posible en cuanto a no haber sido Vectorizada.

Conclusiones

En este proyecto hemos tratado de hacer las cosas paso a paso y tratando de ser precavidos. Sin embargo, nos hemos confiado en algunas ocasiones y hemos interpretado que un resultado ya era correcto por el hecho de obtener un tiempo mejor, lo cual ha llevado a muchas suposiciones erróneas que se habrían evitado si hubiéramos mirado el código ensamblador con más detalle y nos hubiéramos percatado de todo lo que el código nos ocultaba desde un inicio.

Somos optimistas en cuanto a saber que determinados cambios hemos sabido aplicarlos y han funcionado, pero cambios más complejos como los de la Vectorización del código y el entender qué partes el compilador no optimiza, no hemos sabido solucionarlas sin ayuda del profesor. Sin embargo, no son todo malas noticias ya que gracias a eso, hemos conseguido entender mejor nuestro código, aprender a no dejarnos engañar tan fácilmente por el resultado de las ejecuciones y empezar a dudar de lo que suponemos y dar el paso a mirar a fondo el código ensamblador hasta diferenciar completamente los pasos que nosotros hemos hecho en el código de los que el compilador ha hecho por su cuenta.

Vías de Continuación

Aunque nosotros hayamos conseguido optimizar bastante el código, el trabajo no ha de acabar ahí. Hay partes del código que podrían optimizarse mejor:

1. **Paralelizar de mejor forma con OpenMP**: Mejorar los pragmas y hacer que pequeños cambios en la forma de distribuir u organizar la memoria de cada Thread evite pérdidas de tiempo en la ejecución MultiThread.

2. **Paralelizar con OpenACC**: Nuestro código tiene el potencial necesario para llegar a poder conseguir un buen resultado ejecutando instrucciones de OpenACC y paralelizando masivamente las instrucciones a ejecutar.

Es posible que una vez conseguidos estos 2 puntos, se pueda llegar a optimizar aún más. Sin embargo, quizá haya que aumentar el tamaño del problema y dejar de trabajar con una matriz de 3000*3000.