ARQO

Memoria Práctica 2

Roberto Martín Alonso

Diego Forte Jara

Pareja 11

ÍNDICE

Ejercicio 1

1.1-

- Procesador e instrucciones admitidas:

```
(base) tugfa123@AmogOS:/mnt/e/PracticasUamTercero/Practicas_ARQO/P2/Codigo$ lscpu
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         48 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  16
  On-line CPU(s) list:   0-15
Vendor ID:               AuthenticAMD
  Model name:            AMD Ryzen 7 3700X 8-Core Processor
    CPU family:          23
    Model:               113
    Thread(s) per core:  2
    Core(s) per socket:  8
    Socket(s):           1
    Stepping:            0
    BogoMIPS:            8400.04
    Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse3
                         6 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb rdts
                         cp lm constant_tsc rep_good nopl tsc_reliable nonstop_tsc cpuid extd_a
                         picid pni pclmulqdq ssse3 fma cx16 sse4_1 sse4_2 movbe popcnt aes xsav
                         e avx f16c rdrand hypervisor lahf_lm cmp_legacy svm cr8_legacy abm sse
                         4a misalignsse 3dnowprefetch osvw topoext perfctr_core ssbd ibpb stibp
                          vmmcall fsgsbase bmi1 avx2 smep bmi2 rdseed adx smap clflushopt clwb
                         sha_ni xsaveopt xsavec xgetbv1 clzero xsaveerptr arat npt nrip_save ts
                         c_scale vmcb_clean flushbyasid decodeassists pausefilter pfthreshold v
                         _vmsave_vmload umip rdpid
```

Tras analizar los resultados se comprueba que el procesador utilizado es un AMD Ryzen 3700X y las instrucciones admitidas son hasta AVX2

NOTA: Todas las pruebas realizadas y analizadas mostradas en este documento han sido realizadas en el mismo equipo y con las mismas herramientas de compilación, por lo que no se volverán a indicar en las siguientes pruebas para evitar redundancias.

- Versión del compilador:

```
(base) tugfa123@AmogOS:/mnt/e/PracticasUamTercero/Practicas_ARQO/P2/Codigo$ gcc --version
gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

La versión del compilador gcc utilizado es la 11.4.0

- Informe de vectorización:

```
(base) tugfa123@AmogOS:/mnt/e/PracticasUamTercero/Practicas_ARQO/P2/Codigo/material$ gcc -O3 -march=native -fopt-info-vec-optimized -o simple2 simple2.c
simple2.c:48:21: optimized: loop vectorized using 32 byte vectors
simple2.c:41:17: optimized: loop vectorized using 32 byte vectors
```

Se obtiene el mismo informe que en el ejemplo, vectorizándose ambos bucles del ejemplo. En este caso las líneas donde se localizan los bucles no coinciden con las del ejemplo ya que se utiliza una versión de simple2.c donde ya se calculan los tiempos de ejecución.

- Informe de no vectorización:

```
(base) tugfa123@AmogOS:/mnt/e/PracticasUamTercero/Practicas_ARQO/P2/Codigo/material$ gcc -O3 -march=native -fno-tree-vectorize -fopt-info-vec-optimized -o simple2_
no_vec simple2.c
(base) tugfa123@AmogOS:/mnt/e/PracticasUamTercero/Practicas_ARQO/P2/Codigo/material$
```

Tal y como se explica en el ejemplo de la práctica, no se obtiene informe alguno ya que no se vectoriza ningún bucle.

- Comparación ASM de vectorizado y no vectorizado:
  Siendo el primer fichero simple2_o3.s y el segundo simple2_o3_native.s percibimos las siguientes diferencias:

```
53,76c51,65
<       movdqa  .LC0(%rip), %xmm2
<       movdqa  .LC3(%rip), %xmm4
<       movdqa  .LC4(%rip), %xmm3
<       movq    %rcx, %rax
<       movq    %rdx, %rsi
<       leaq    16384(%rcx), %rdi
< .L6:
<       movdqa  %xmm2, %xmm0
<       addq    $32, %rax
<       paddd   %xmm4, %xmm2
<       addq    $32, %rsi
<       cvtdq2pd        %xmm0, %xmm1
<       movaps  %xmm1, -32(%rax)
<       pshufd  $238, %xmm0, %xmm1
<       paddd   %xmm3, %xmm0
<       cvtdq2pd        %xmm1, %xmm1
<       movaps  %xmm1, -16(%rax)
<       cvtdq2pd        %xmm0, %xmm1
<       pshufd  $238, %xmm0, %xmm0
<       cvtdq2pd        %xmm0, %xmm0
<       movaps  %xmm1, -32(%rsi)
<       movaps  %xmm0, -16(%rsi)
<       cmpq    %rdi, %rax
<       jne     .L6
---
>       movl    %eax, %r8d
>       xorl    %eax, %eax
>       testl   %r8d, %r8d
>       jne     .L21
> .L5:
>       pxor    %xmm0, %xmm0
>       leal    1(%rax), %esi
>       cvtsi2sdl       %eax, %xmm0
>       movsd   %xmm0, (%rcx,%rax,8)
>       pxor    %xmm0, %xmm0
>       cvtsi2sdl       %esi, %xmm0
>       movsd   %xmm0, (%rdx,%rax,8)
>       addq    $1, %rax
>       cmpq    $2048, %rax
>       jne     .L5
```

  Aquí destacamos instrucciones como movdqa, las cual mueve datos de 128 bits (cuatro valores de 32 bits o dos valores de 64 bits) de registros XMM a registros XMM o de registros XMM a memoria. Comparándola con movl, ésta solo mueve un valor de 32 bits entre registros (normalmente registros de propósito general, aunque también puede usar registros XMM) o entre registros y memoria.

En general las diferencias que se encuentran entre ambos códigos es el uso de instrucciones que trabajan con vectores de datos en el código con vectorización (instrucciones SIMD) y de registros que se usan para ello, como los XMM.

```
78c67
<       movapd  .LC5(%rip), %xmm3
---
>       movsd   .LC2(%rip), %xmm2
```

En este caso la instrucción movapd se utiliza para mover datos de 128 bits de precisión doble entre registros XMM o entre un registro XMM y la memoria. En cambio movsd realiza la misma tarea pero con un único dato.

```
84,90c73,77
< .L8:
<       movapd  (%rdx,%rax), %xmm0
<       mulpd   %xmm3, %xmm0
<       addpd   (%rcx,%rax), %xmm0
<       addq    $16, %rax
<       addsd   %xmm0, %xmm1
<       unpckhpd        %xmm0, %xmm0
---
> .L7:
>       movsd   (%rdx,%rax), %xmm0
>       mulsd   %xmm2, %xmm0
>       addsd   (%rcx,%rax), %xmm0
>       addq    $8, %rax
```

Este caso es parecido al anterior, pero usa además las instrucciones mulpd y addpd que operan con dos pares datos de doble precisión simultáneamente a diferencia de mulsd y addsd, que operan con un único par de valores de doble precisión.

1.2- Comparación de rendimientos:

-O3 con vectorización:

```
(base) tugfa123@AmogOS:/mnt/e/PracticasUamTercero/Practicas_ARQO/P2/Codigo$ gcc -O3 -o simple2 simple2.c
(base) tugfa123@AmogOS:/mnt/e/PracticasUamTercero/Practicas_ARQO/P2/Codigo$ ./simple2
Los microsegundos que ha tardado han sido: 1472414.000000
```

-O3 sin vectorización:

```
(base) tugfa123@AmogOS:/mnt/e/PracticasUamTercero/Practicas_ARQO/P2/Codigo$ gcc -O3 -fno-tree-vectorize -o simple2 simple2.c
(base) tugfa123@AmogOS:/mnt/e/PracticasUamTercero/Practicas_ARQO/P2/Codigo$ ./simple2
Los microsegundos que ha tardado han sido: 1473906.000000
```

-O2 con vectorización:

```
(base) tugfa123@AmogOS:/mnt/e/PracticasUamTercero/Practicas_ARQO/P2/Codigo$ gcc -O2 -o simple2 simple2.c
(base) tugfa123@AmogOS:/mnt/e/PracticasUamTercero/Practicas_ARQO/P2/Codigo$ ./simple2
Los microsegundos que ha tardado han sido: 1472628.000000
```

-O2 sin vectorización:

```
(base) tugfa123@AmogOS:/mnt/e/PracticasUamTercero/Practicas_ARQO/P2/Codigo$ gcc -O2 -fno-tree-vectorize -o simple2 simple2.c
(base) tugfa123@AmogOS:/mnt/e/PracticasUamTercero/Practicas_ARQO/P2/Codigo$ ./simple2
Los microsegundos que ha tardado han sido: 1472871.000000
```

-O1 con vectorización:

```
(base) tugfa123@AmogOS:/mnt/e/PracticasUamTercero/Practicas_ARQO/P2/Codigo$ gcc -O1 -o simple2 simple2.c
(base) tugfa123@AmogOS:/mnt/e/PracticasUamTercero/Practicas_ARQO/P2/Codigo$ ./simple2
Los microsegundos que ha tardado han sido: 1511397.000000
```

-O1 sin vectorización:

```
(base) tugfa123@AmogOS:/mnt/e/PracticasUamTercero/Practicas_ARQO/P2/Codigo$ gcc -O1 -fno-tree-vectorize -o simple2 simple2.c
(base) tugfa123@AmogOS:/mnt/e/PracticasUamTercero/Practicas_ARQO/P2/Codigo$ ./simple2
Los microsegundos que ha tardado han sido: 1521964.000000
```

	-O1	-O2	-O3
Tiempo con vectorización (microsegundos)	1511397	1472628	1472414
Tiempo sin vectorización (microsegundos)	1521964	1472871	1473906

Para calcular la aceleración se utiliza la fórmula:

$$\text{Aceleración}(A) = T_{original} / T_{optimizado}$$

Como Toriginal se tomará el tiempo de la ejecución mas lenta, en este caso -O1 sin vectorizar, obteniéndose los siguientes resultados:

	Sin vectorización	Con vectorización
Aceleración -O1	1521964/1521964= 1	1521964/1511397 = 1,007
Aceleración -O2	1521964/1472871 = 1,0333	1521964/1472628 = 1,0335
Aceleración -O3	1521964/1473906 = 1,0326	1521964/ 1472414 = 1,337

Como se puede observar en la tabla de resultados, el tiempo de ejecución ha ido en aumento desde -O1 sin vectorización hasta -O3 con vectorización. Como es un programa con un tiempo de ejecución muy corto las diferencias de tiempo son pequeñas y el resultado también depende de que tan ocupado este el procesador en el momento de ejecución, pudiendo darse que el código con vectorización tenga un tiempo de ejecución mayor que el código sin vectorizar.

1.3- Opciones posibles para el flag -march:

```
(base) tugfa123@AmogOS:/mnt/e/PracticasUamTercero/Practicas_ARQO/P2/Codigo$ gcc -O3 -S -march=xxx simple2.c
cc1: error: bad value ('xxx') for '-march=' switch
cc1: note: valid arguments to '-march=' switch are: nocona core2 nehalem corei7 westmere sandybridge corei7-avx ivybridge core-avx-i ha
swell core-avx2 broadwell skylake skylake-avx512 cannonlake icelake-client rocketlake icelake-server cascadelake tigerlake cooperlake s
apphirerapids alderlake bonnell atom silvermont slm goldmont goldmont-plus tremont knl knm x86-64 x86-64-v2 x86-64-v3 x86-64-v4 eden-x2
 nano nano-1000 nano-2000 nano-3000 nano-x2 eden-x4 nano-x4 k8 k8-sse3 opteron opteron-sse3 athlon64 athlon64-sse3 athlon-fx amdfam10 b
arcelona bdver1 bdver2 bdver3 bdver4 znver1 znver2 znver3 btver1 btver2 native
(base) tugfa123@AmogOS:/mnt/e/PracticasUamTercero/Practicas_ARQO/P2/Codigo$
```

Para el análisis de las diferencias entre arquitecturas para SSE se utiliza -march=athlon64sse3, para AVX se utiliza -march= corei7-avx y para AVX512 se utiliza -march= skylake-avx512

En lo referente a registros se comprueba que el ensamblador correspondiente a SSE utiliza solo registros XMM para la vectorización y tanto AVX como AVX512 usan registros XMM e YMM. Se observa que en el ensamblador de AVX512 no se usan registros ZMM, esto puede deberse a que como es un programa tan simple no sea necesario usar registros de 512 bits.

En lo referente a instrucciones se comparan los ficheros mediante el comando diff y se observa que se usan instrucciones similares, pero en AVX512 usa mas frecuentemente registros YMM que en AVX. También se encuentra que en AVX512 se usa la instrucción vextractf64x2 que se utiliza para extraer un vector de doble precisión de un registro de 256 o 512 bits y en AVX se utiliza la instrucción vextractf128 que extrae parte de un registro YMM, de 256 bits y lo copia en un registro XMM, de 128 bits

Por otra parte, comparando el ensamblado de SSE con AVX se observa que este último contiene instrucciones adicionales que no están en el ensamblado de SSE, como vmovdqa, que se usa para mover datos de 128 bits entre registros XMM y memoria.

Por último, se aprecia que en el ensamblado se usa la instrucción xorpd, que es parte del juego de instrucciones SSE y en el ensamblado se usa la instrucción vxorpd, que es parte del juego de instrucciones AVX

Ejercicio 2

Se crea el fichero simple2_intrinsics.c y se comprueba que el valor de la variable c no es alterado por la vectorización es el mismo

```
(base) tugfa123@AmogOS:/mnt/e/PracticasUamTercero/Practicas_ARQO/P2/Codigo$ gcc -mavx -mfma -o intrinsics simple2_intrinsics.c
(base) tugfa123@AmogOS:/mnt/e/PracticasUamTercero/Practicas_ARQO/P2/Codigo$ gcc -o no_intrinsics simple2.c
(base) tugfa123@AmogOS:/mnt/e/PracticasUamTercero/Practicas_ARQO/P2/Codigo$ ./intrinsics
Los microsegundos que ha tardado han sido: 2.000000
El resultado es: 4194513.817600
(base) tugfa123@AmogOS:/mnt/e/PracticasUamTercero/Practicas_ARQO/P2/Codigo$ ./no_intrinsics
Los microsegundos que ha tardado han sido: 5.000000
El resultado es: 4194513.817600
```

2.1- Vectorización del primer bucle:

Se vectoriza el primer bucle mediante el siguiente código:

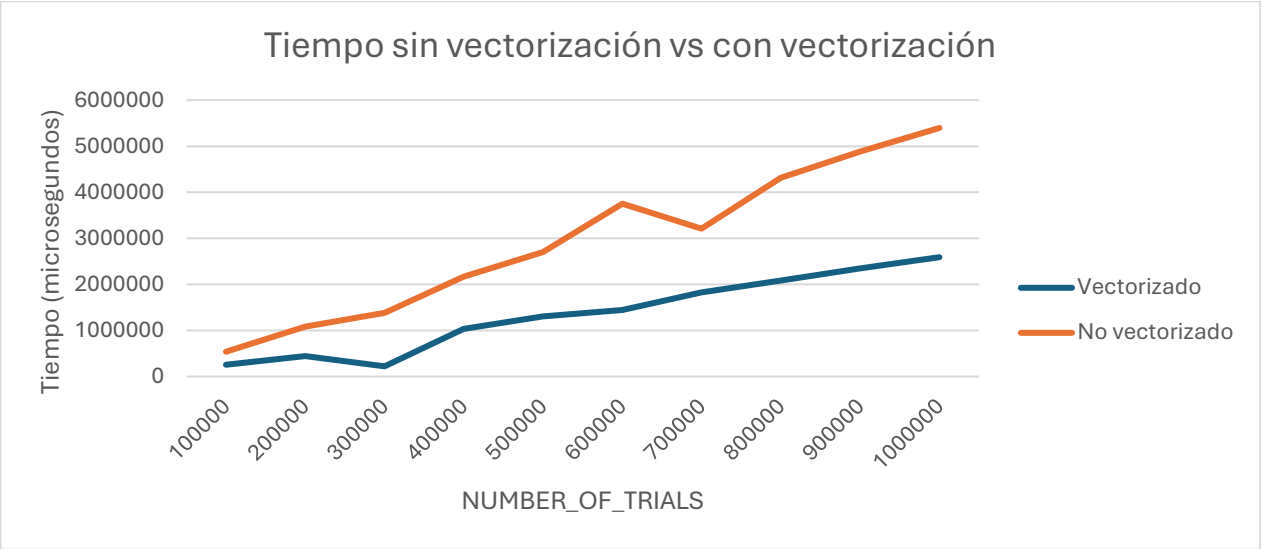
```
__m256d vib = {-4.0, -3.0, -2.0, -1.0};
__m256d via = {-3.0, -2.0, -1.0, 0.0};
__m256d index = {4.0, 4.0, 4.0, 4.0};
```

```
/* Populate A and B arrays */
for (i=0; i < ARRAY_SIZE; i+= 4) {

    vib = _mm256_add_pd(vib, index);
    _mm256_store_pd(&b[i], vib);
    //b[i] = i;

    via = _mm256_add_pd(via, index);
    _mm256_store_pd(&a[i], via);
    //a[i] = i+1;
}
```

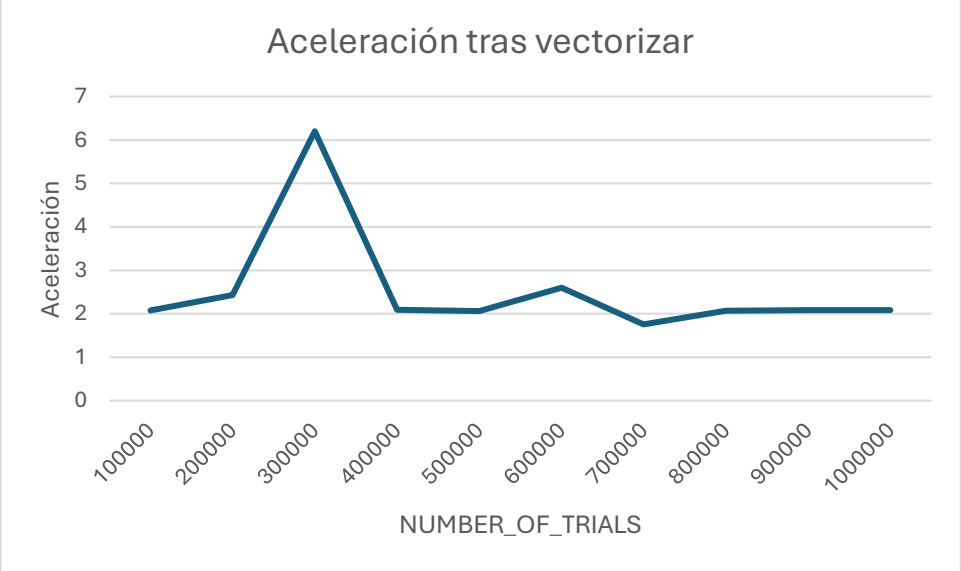
Explicación: Se crean tres nuevos vectores, uno para el array a, otro para el array b y por último un array que hará de sumador. En cada iteración del bucle a cada vector de cada array se le sumará el vector index y se guardará en el array, es decir, comenzara el vector vib en {-4.0, -3.0, -2.0, -1.0}, se le suma el vector index {4.0, 4.0, 4.0, 4.0}, quedando {0.0, 1.0, 2.0, 3.0} y guardándose cada dato en b[0], b[1], b[2], b[3] respectivamente. Con el array “a” el funcionamiento es el mismo solo que el valor de los datos éste empiezan desplazados en una unidad.

2.2- Tiempos de ejecución y gráfica de comparación:

NUMBER_OF_TRIALS	100000	200000	300000	400000	500000	600000	700000	800000	900000	1000000
Tiempo con vectorización (microsegundos)	260225	446827	222639	1036964	1308879	1444550	1831096	2085820	2349028	2591588
Tiempo sin vectorización (microsegundos)	539588	1084668	1380524	2167518	2701849	3756009	3213601	4317640	4886486	5397369
Aceleración	2,073544	2,4274898	6,2007285	2,09025386	2,0642466	2,60012391	1,755015	2,06999645	2,08021616	2,08264933



2.3- Gráfica de vectorización:



Ejercicio 3

* E1.1 Considere la función _mm256_srlv_epi64. Incluso si no sabe lo que significa srlv, ¿Qué se puede decir de la salida y de los argumentos de entrada de la función?

La salida es un vector de 256 bits y los argumentos de entrada será un vector de enteros con signo de 8, 16, 32, o 64 bits

* E1.2 Considere la función _mm_testnzc_ps. Incluso si no sabe lo que significa testnzc, ¿Qué se puede decir de la salida y de los argumentos de entrada de la función?

Como no aparece número después de mm, el tamaño del vector devuelto es de 128 bits y los argumentos de entrada son vectores que contienen floats

* E1.3 En un registro SIMD de 128 bit, ¿Cuántos enteros se pueden almacenar?

Depende del tamaño del entero, si es de 32 bits entonces se podrían almacenar 4 enteros

* E1.4 En un registro SIMD de 128 bit, ¿Cuántos números reales de precisión simple (float) se pueden almacenar?

Se pueden almacenar 4 floats

* E1.5 En un registro SIMD de 128 bit, ¿Cuántos números reales de doble precisión (double) se pueden almacenar?

Se pueden almacenar dos doubles

En un procesador con extensiones SIMD se utilizan registros de 2048 bits.

* E1.6 ¿Cuántos números reales de doble precisión pueden almacenar?

Se pueden almacenar 32 doubles

* E1.7 ¿Cuántos carácteres se pueden almacenar?

Como un char ocupa 8 bits se pueden almacenar 256 chars

* E1.8 ¿Cuántos **pixeles** se pueden almacenar? (un pixel son 3 bytes para almacenar rojo-verde y azul)

Se pueden almacenar 85 pixeles.

* E1.9 ¿Cuantos enteros se pueden almacenar en un registro YMM?

Depende del tamaño del entero, para enteros de 32 bits se pueden almacenar 8 enteros

* E1.10 Suponga un vector de 32 doubles, (double vest[32]), ¿ Cuantos registros se necesitan para procesarlo utilizando instrucciones SIMD?

Para registros ZMM se necesitan 4 registros, para YMM se necesitan 8 registros y para XMM se necesitan 16 registros.

* E2.1 ¿Coinciden siempre los resultados SIMD con el secuencial?

No, no coinciden en todos los casos.

* E2.2 ¿Coincide los denominados secuencial 1 y secuencial 2?

Tampoco coinciden los datos de secuencial 1 y secuencial 2.

* E2.3 ¿Cómo se justifican estos resultados?

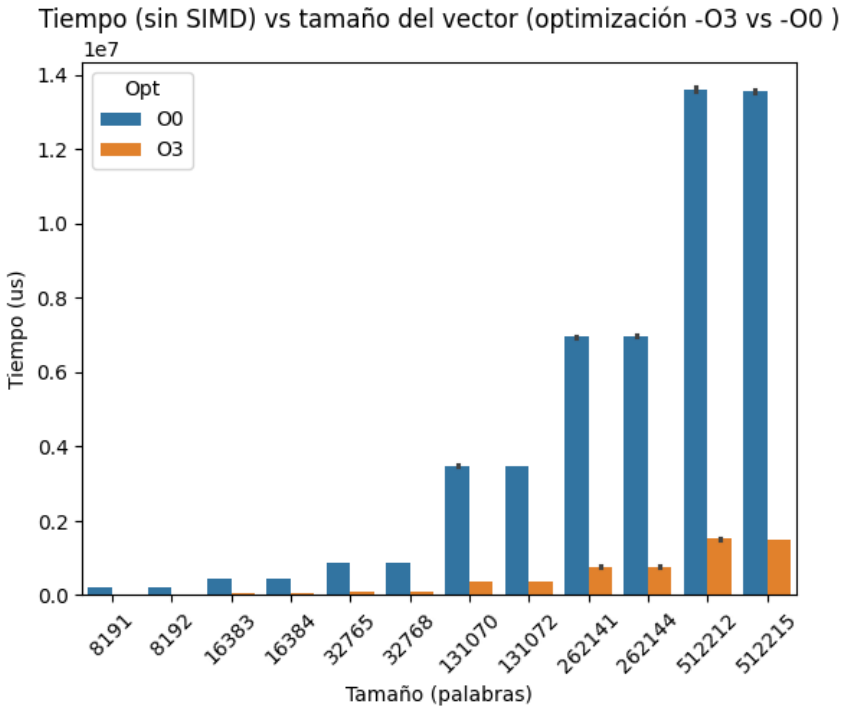
Dependiendo del orden en el que se operen los datos estos son truncados en órdenes distintos, por lo que se producen distintos resultados.

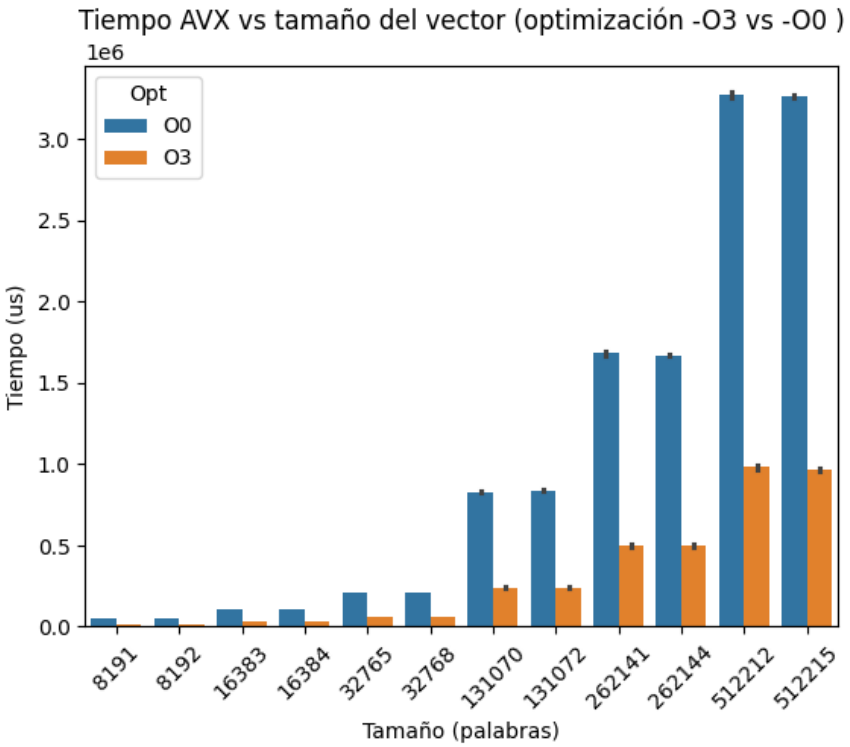
* E2.4 Si son diferentes ¿Cuál es el resultado correcto?

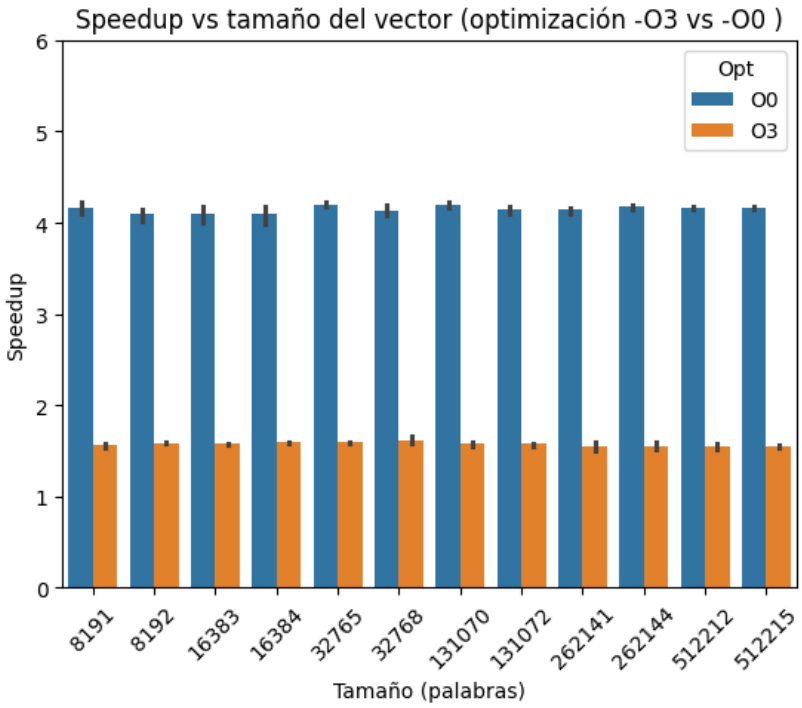
Estrictamente ninguno, ya que el resultado correcto es el que se obtiene tras operar con todas las cifras de los operandos y no truncando los valores. Como esto no es posible, todos los valores son correctos.

* E3

Partiendo del código del análisis del producto escalar de dos vectores con doubles se modifica para usarlo en valores de tipo float y analizar los resultados, obteniéndose las siguientes gráficas:

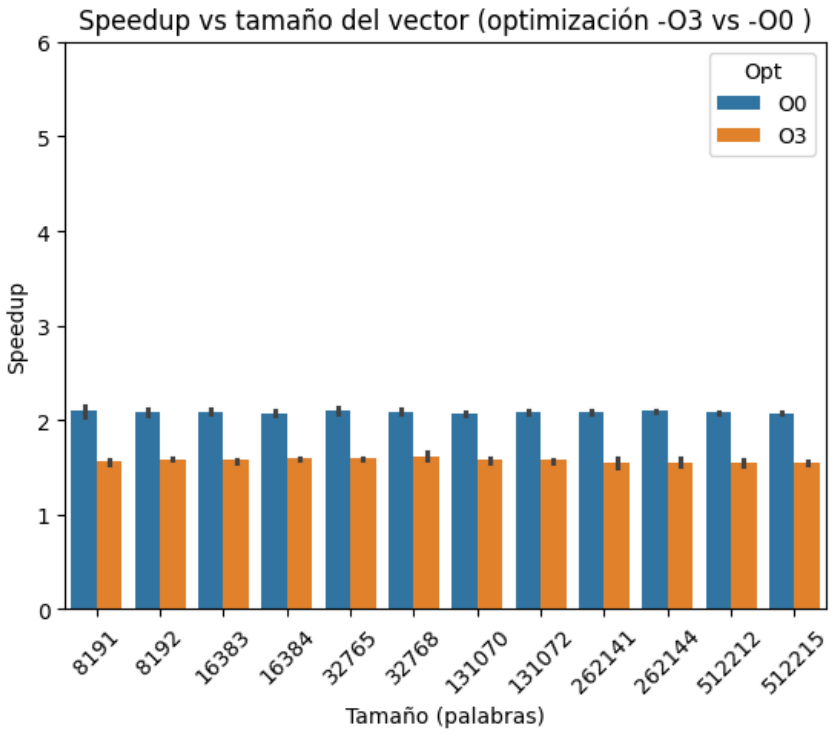






Como puede observarse en la primera y segunda gráfica el tiempo de ejecución es mucho menor en el código compilado con el flag -O3 que el compilado con el -O0, lo cuál es esperable ya que al compilar con -O3 se aplican mas optimizaciones que con -O0.

En lo que respecta a la gráfica de la aceleración (speedup) se observa que la aceleración del código menos optimizado (-O0) es mayor que la del código optimizado (-O3), no se ha encontrado explicación de a que se debe esto. Por otro lado, se ha comparado dicha gráfica con la generada por el producto escalar de dos vectores con elementos double, la cual es la siguiente:



Como se puede apreciar en el caso de O0 en vectores con datos float la aceleración es algo mas de 4 y en vectores con datos double con el mismo flag de compilación la aceleración es algo mayor a 2, por lo que la aceleración de ha duplicado prácticamente. Reflexionando, esto puede deberse a que en el mismo registro AVX de 256 bit caben 4 doubles u 8 floats, por lo que se pueden operar el doble de elementos por instrucción al usar floats en lugar de doubles.

Ejercicio 4

0. Compila y ejecuta el programa utilizando algunas de las imágenes proporcionadas como argumentos. Examina los resultados que se generaron y analiza brevemente el programa proporcionado.
   Se observa que el programa proporcionado convierte una imagen en color a escala de grises, cargando la imagen como un array de bits, tomando el valor RGB de cada uno de ellos, modificándolo para pasarlo a gris y guardando dicha modificación en una nueva imagen, que será la final.

1. El programa incluye dos bucles. El primer bucle (indicado como Loop 0) itera sobre los argumentos aplicando el algoritmo a cada uno de ellos. El segundo bucle (indicado como Loop 1) computa el algoritmo de escala de grises. ¿Es este bucle óptimo para ser vectorizado? ¿Por qué? Consejo: recomendamos utilizar Compiler Explorer (https://godbolt.org/).
   El segundo bucle es óptimo para ser optimizado ya que pueden cargarse 4 píxeles al mismo tiempo y operar sobre sus valores rgb para obtener 4 píxeles en escala de grises.

2. Vectorización manual del bucle:

```
// RGB to grey scale
int r, g, b, i, j;
float r_coef = 0.2989, green_coef = 0.5870, blue_coef = 0.1140, grey_val;

#pragma GCC ivdep
for (int idx = 0; idx < width * height; idx++)
{
    i = idx % width; //Indice de las columnas
    j = idx / width; //Indice de las filas

    getRGB(rgb_image, width, height, 4, i, j, &r, &g, &b);
    grey_val = r_coef * r + green_coef * g + blue_coef * b;
    grey_image[idx] = (int)(grey_val);

}
```

Se modifica el código original por el de la imagen, reduciendo el número de bucles a uno y haciendo que "i" y "j" se calculen dentro de éste para ayudar al compilador a vectorizar el bucle. Adicionalmente se incluye la directiva "GCC ivdep" para sugerir la vectorización del bucle.
Aun con estos cambios en el código se han revisado los informes de vectorización tras compilar el código con el flag "-fopt-info-vec-optimized" y no aparece como vectorizado, también se han revisado los reportes de bucles no vectorizados con el flag de compilación "-fopt-info-vec-missed" y tampoco aparece como no vectorizado. Se han realizado varias pruebas de ejecución del programa y se han observado mejoras en los tiempos del código mejorado respecto al original, pero no se puede asegurar que sea porque el código haya sido vectorizado.