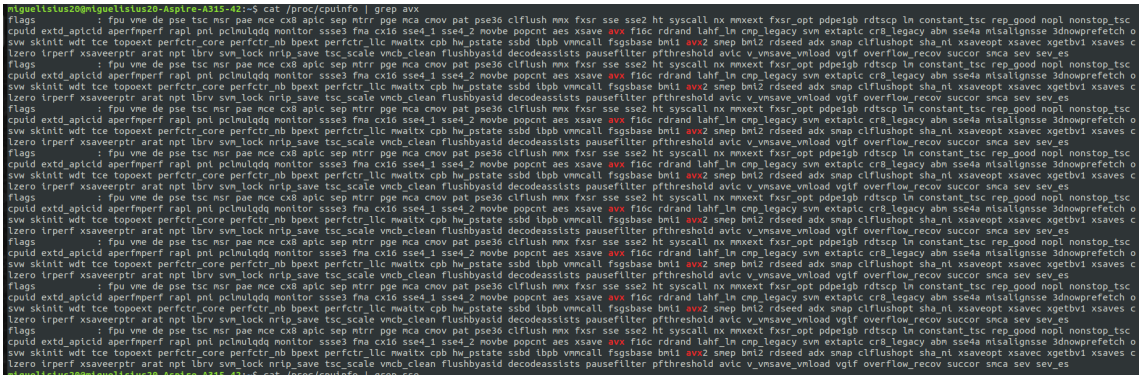
ASIGNATURA Computación de altas prestaciones

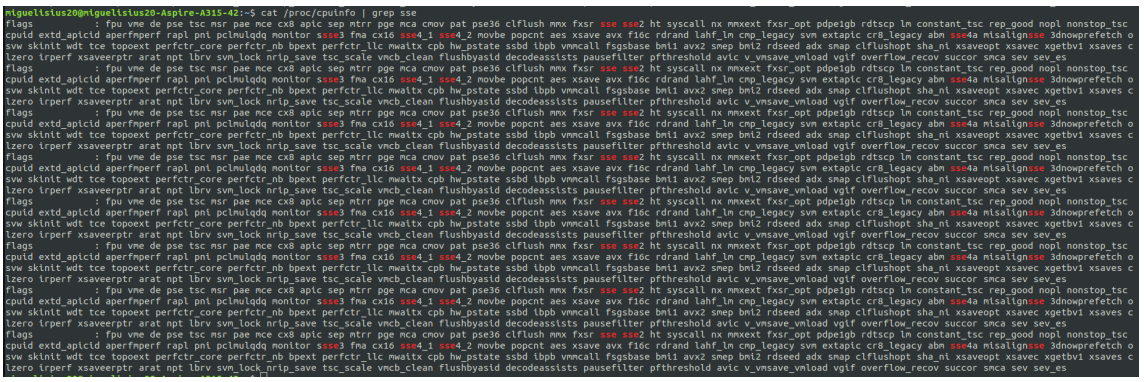
Práctica1
Vector processing and SIMD

You must write a report answering the questions proposed in each exercise, plus the requested files. Submit a zip file through Moodle. Check submission date in Moodle (deadline is until 11:59 pm of that date).

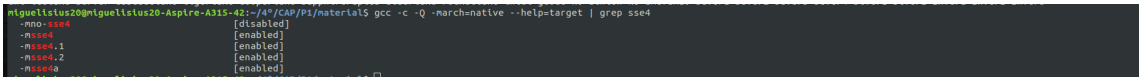
- Exercise 1:
  - Identify your CPU model and list the supported SIMD instructions.

Tras ejecutar el comando cat /proc/cpuinfo | grep sse y cat /proc/cpuinfo | grep avx podemos concretar que el ordenador soporta las instrucciones de avx y avx2 y las del sse4, sse4_2, etc.. como mostramos en las siguiente imágenes:



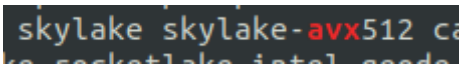


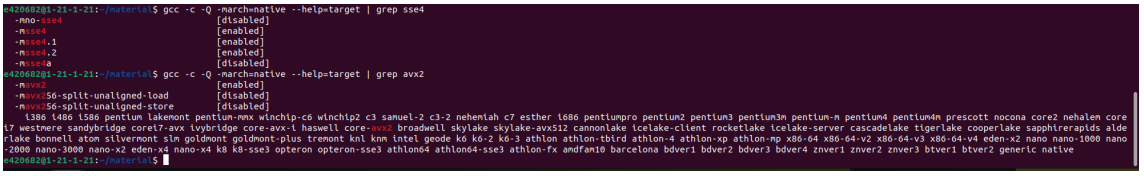
Como podemos ver en la siguiente imagen, en nuestro ordenador tenemos activadas las instrucciones de sse, es decir, que por parte de sse4.1 tendremos activadas un total de 47 instrucciones y las 7 restantes pertenecientes al sse4.2 Permitiendo realizar así operaciones en coma flotante a un mayor rendimiento.



En la siguiente imagen podemos ver que también están activados sse2 y sse3, que manejan datos de cuatro números en coma flotante de 32 bits o dos de 64 bits.

Como podemos ver en la siguiente imagen, nuestro procesador amd contiene skylake-avx512, permitiendo mejorar el rendimiento donde exista una gran carga de trabajo. Gracias al avx512 tendremos una aceleración en el rendimiento en aquellos trabajos que supongan una carga elevada. Pudiendo soportar así operaciones float de 32 a 64 bits.

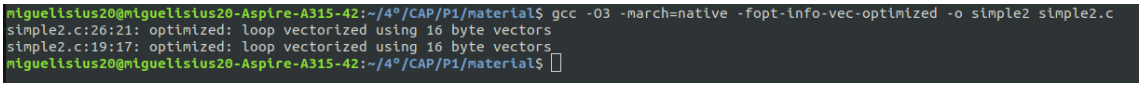




Tras ejecutar el comando:

```
gcc -O3 -march=native -fopt-info-vec-optimized -o simple2 simple2.c
```

Hemos obtenido que se ha vectorizado correctamente usando vectores de 16 Bytes como mostramos en la siguiente imagen:



Tras realizar los siguientes comandos con la vectorización totalmente desactivada y tras crear el archivo simple_o3_native.s (que está en ensamblador) y simple2_o3.s podemos comenzar a buscar las principales diferencias como mostramos en la siguiente imagen.

```
miguelisius20@miguelisius20-Aspire-A315-42:~/4º/CAP/P1/material$ gcc -O3 -march=native -fno-tree-vectorize -fopt-info-vec-optimized -o simple2_no_vec simple2.c
miguelisius20@miguelisius20-Aspire-A315-42:~/4º/CAP/P1/material$ gcc -S -O3 simple2.c
miguelisius20@miguelisius20-Aspire-A315-42:~/4º/CAP/P1/material$ mv simple2.s simple2_o3.s
miguelisius20@miguelisius20-Aspire-A315-42:~/4º/CAP/P1/material$ gcc -S -O3 -fno-tree-vectorize simple2.c
miguelisius20@miguelisius20-Aspire-A315-42:~/4º/CAP/P1/material$ mv simple2.s simple2_o3_native.s
miguelisius20@miguelisius20-Aspire-A315-42:~/4º/CAP/P1/material$ diff simple2_o3.s simple2_o3_native
diff: simple2_o3_native: No existe el archivo o el directorio
miguelisius20@miguelisius20-Aspire-A315-42:~/4º/CAP/P1/material$ diff simple2_o3.s simple2_o3_native.s
```

- Explain the main differences between both assembly codes (vectorized and non-vectorized) focused on the SIMD instructions generated by the compiler.

```
10a11
>       xorl    %eax, %eax
13,18d13
<       movdqa  .LC0(%rip), %xmm2
<       movdqa  .LC1(%rip), %xmm4
<       movdqa  .LC2(%rip), %xmm3
<       movq    %rcx, %rax
<       movq    %rdx, %rsi
<       leaq    16384(%rcx), %rdi
20,35c15,23
<       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
---
>       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
38c26
<       movapd  .LC3(%rip), %xmm3
---
>       movsd   .LC0(%rip), %xmm2
45,50c33,36
<       movapd  (%rdx,%rax), %xmm0
<       mulpd   %xmm3, %xmm0
<       addpd   (%rcx,%rax), %xmm0
<       addq    $16, %rax
<       addsd   %xmm0, %xmm1
<       unpckhpd        %xmm0, %xmm0
---
>       movsd   (%rdx,%rax), %xmm0
>       mulsd   %xmm2, %xmm0
>       addsd   (%rcx,%rax), %xmm0
>       addq    $8, %rax
68,69c54,55
<       .section        .rodata.cst16,"aM",@progbits,16
```

```
<       .align 16
---
>       .section        .rodata.cst8,"aM",@progbits,8
>       .align 8
71,90d56
<       .long   0
<       .long   1
<       .long   2
<       .long   3
<       .align 16
< .LC1:
<       .long   4
<       .long   4
<       .long   4
<       .long   4
<       .align 16
< .LC2:
<       .long   1
<       .long   1
<       .long   1
<       .long   1
<       .align 16
< .LC3:
<       .long   -611603343
<       .long   1072693352
```

Las dos últimas imágenes son las diferencias entre ambos códigos, el native y el que no es native. Lo que viene a decir prácticamente es que en las líneas:

- 10a11: se está añadiendo una nueva línea usando xorl que inicializa el registro.
- 13,18d13: son líneas que están siendo borradas realizando cálculos en memoria dando así un nuevo enfoque a los cálculos que se van a realizar y en la inicialización de los registros.
- 20,35c15,23: donde como se puede observar que hace unos cambios en las instrucciones que implican a los registros que están indicados por xmm como son %xmm2, %xmm3, %xmm1 y %xmm0, realizando así cálculos mediante las operaciones de SIMD que están relacionadas con aquellas que son de punto flotante y que como hemos dicho en apartados anteriores mejorarán la eficiencia y el rendimiento de aquellas operaciones que involucren un gran trabajo.
- 38c26: realiza un cambio de instrucción SIMD pasando de movapd, que mueve 2,4 u 8 valores en coma flotante de doble precisión; por movsd, que mueve un valor escalar de coma flotante de doble precisión desde el operando origen al operando destino.

- 45,50c33,36: De nuevo se vuelven a realizar sustituciones en operaciones SIMD que involucran a %xmm, de nuevo como hemos dicho en 20,35c15,23 son para operaciones de coma flotante y que aumentan el rendimiento y la eficiencia.
- 68,69c54,55: cambian los datos en la manera en la que se ejecuta lo que puede afectar en la manera a la que son accedidos.
- 71,91d56: que elimina constantes realizando una simplificación y reorganización de las constantes que son usadas.
- Por tanto estas diferencias expuestas indican cambios en la manera en la que se van a inicializar los registros, las optimizaciones en las operaciones de punto flotante mediante el conjunto de instrucciones que aporta SIMD y reorganizando la estructura de datos. Todos estos cambios tienen como objetivo proporcionar una mayor eficacia y un mayor rendimiento.

- Exercise 2:
    - Provide the source code of *simple2_intrinsics.c* after the vectorization of the loops. Explain how you have carried out the vectorization of the code.

Para la realización de este ejercicio hemos creado una copia del fichero simple2.c y lo hemos llamado simple2_intrisics.c.

En este nuevo fichero hemos tenido que vectorizar los bucles.

Lo primero de todo ha sido crear la variable diff es el tamaño del array módulo 4, ya que como hay que cargar de 4 en 4 será necesario usarlo para que llegue al múltiplo de 4 más cercano y así poder empezar a vectorizar correctamente.

En este primer bucle lo que hemos hecho ha sido mediante las variables _m256d y _mm256_store_pd “popular” ambos arrays de 4 en 4.

```
int diff = ARRAY_SIZE % 4;
/* Populate A and B arrays */
for (i = 0; i < (ARRAY_SIZE-diff); i+=4)
{
    __m256d vb = {i, i+1, i+2, i+3};
    __m256d va = {i+1, i+2, i+3, i+4};
    _mm256_store_pd(&a[i], va);
    _mm256_store_pd(&b[i], vb);
}
```

A continuación hemos tenido que crear otro bucle debido a que si por ejemplo el tamaño del array fuera 11, al hacer la diferencia e ir de 4 en 4, lo que pasa es que nos sobrarían 3 posiciones. Y este bucle lo que hace es rellenar esas tres posibles posiciones restantes.

```
//Array %4 !=0
for (i = (ARRAY_SIZE-diff); i < ARRAY_SIZE; i+=4)
{
    b[i] = i;
    a[i] = i + 1;
}
```

A continuación hemos creado dos variables mm y sum, la primera de ellas deberá almacenar 10001 y la segunda debe almacenar las sumas parciales.
Y de nuevo en el bucle más interno hemos cargado los arrays y ejecutado la operación m*a+b y a continuación la hemos acumulado en la variable sum.
Tras acumular en sum hemos guardado cada valor de la suma en una variable c, la cual será impresa por pantalla y nos mostrará el resultado final.

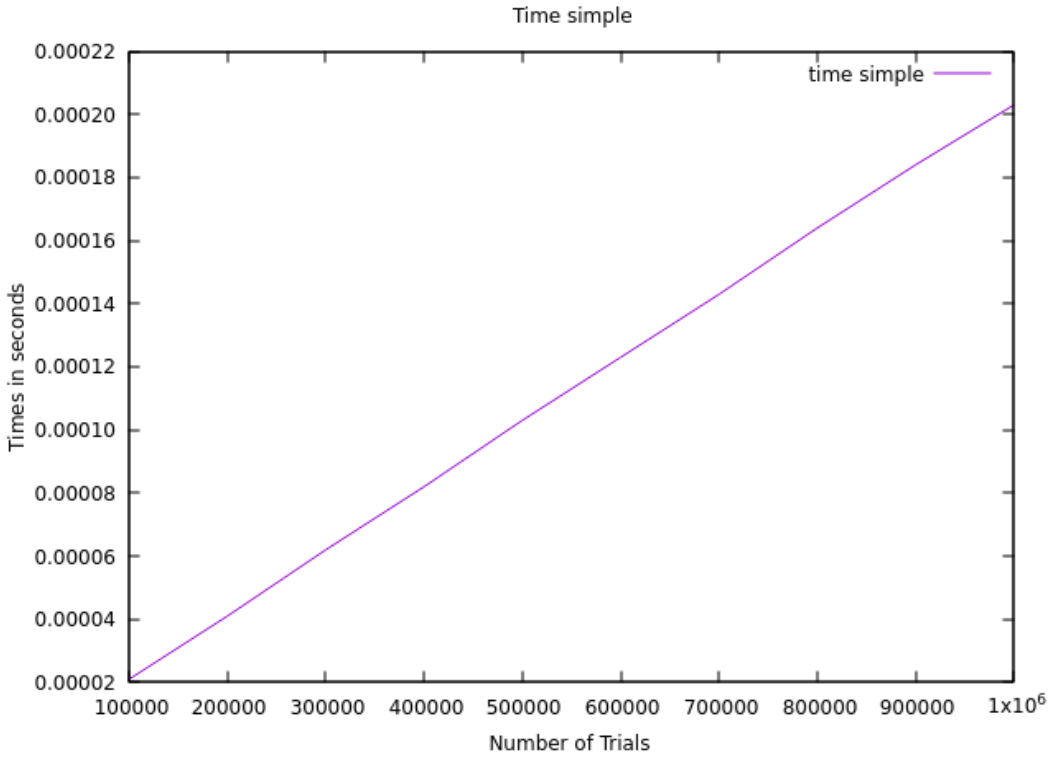
```
for (t = 0; t < NUMBER_OF_TRIALS; t++)
{
    for (i = 0; i < (ARRAY_SIZE-diff); i+=4)
    {
        __m256d sum = {0.0, 0.0, 0.0, 0.0};
        // Load arrays
        __m256d va = _mm256_load_pd(&a[i]);
        __m256d vb = _mm256_load_pd(&b[i]);
        // Compute m*a+b
        __m256d tmp = _mm256_fmadd_pd(mm, va, vb);
        // Accumulate results
        sum = _mm256_add_pd(tmp, sum);
        for (j = 0; j < 4; j++)
        {
            c += sum[j];
        }
    }

    for (i = (ARRAY_SIZE - diff); i < ARRAY_SIZE; i++)
    {
        c += m * a[i] + b[i];
    }
}
```

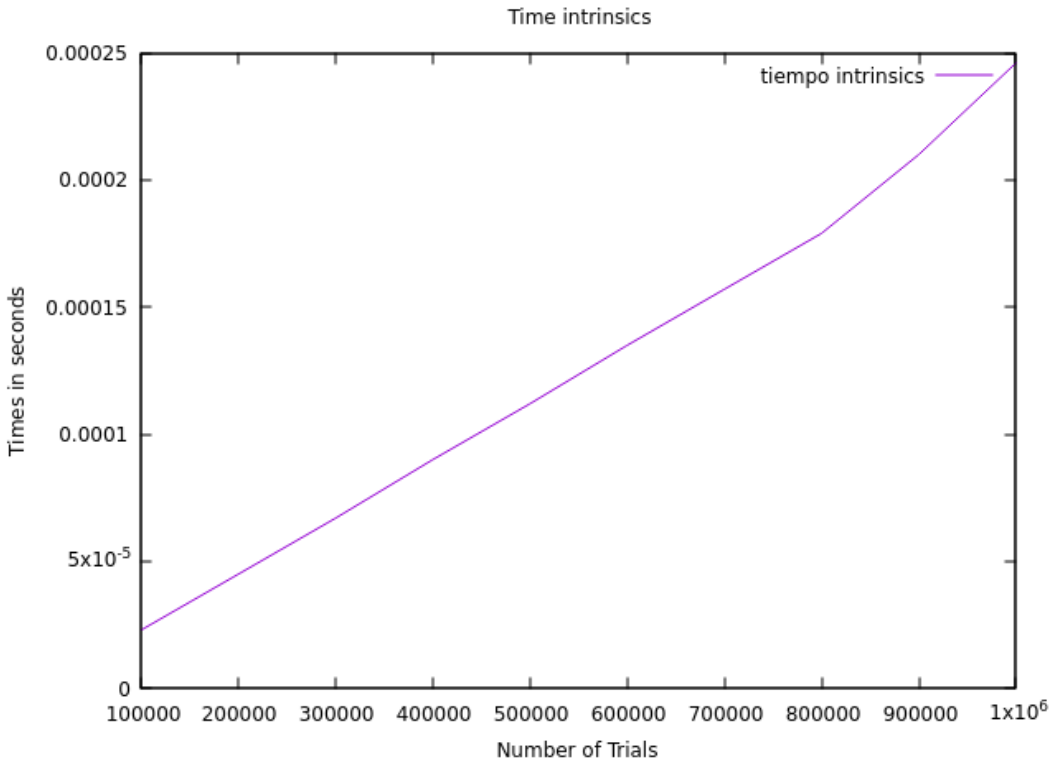
Mediante la vectorización de los bucles, que ahora cada bucle va a recorrer en vez de N elementos que hay en el array recorrerá N/4, esto se debe a que un double ocupa 8B, lo que implica 64bits = 8*8. Si empleamos vectores de 256bits/64 = 4. Gracias a esto podremos reducir el número de iteraciones que se llevan a cabo en el bucle.

- Compare the execution time for different values of NUMBER_OF_TRIALS: from 100.000 to 1.000.000 in steps of 100.000. Plot the results in a graph. Discuss the results.

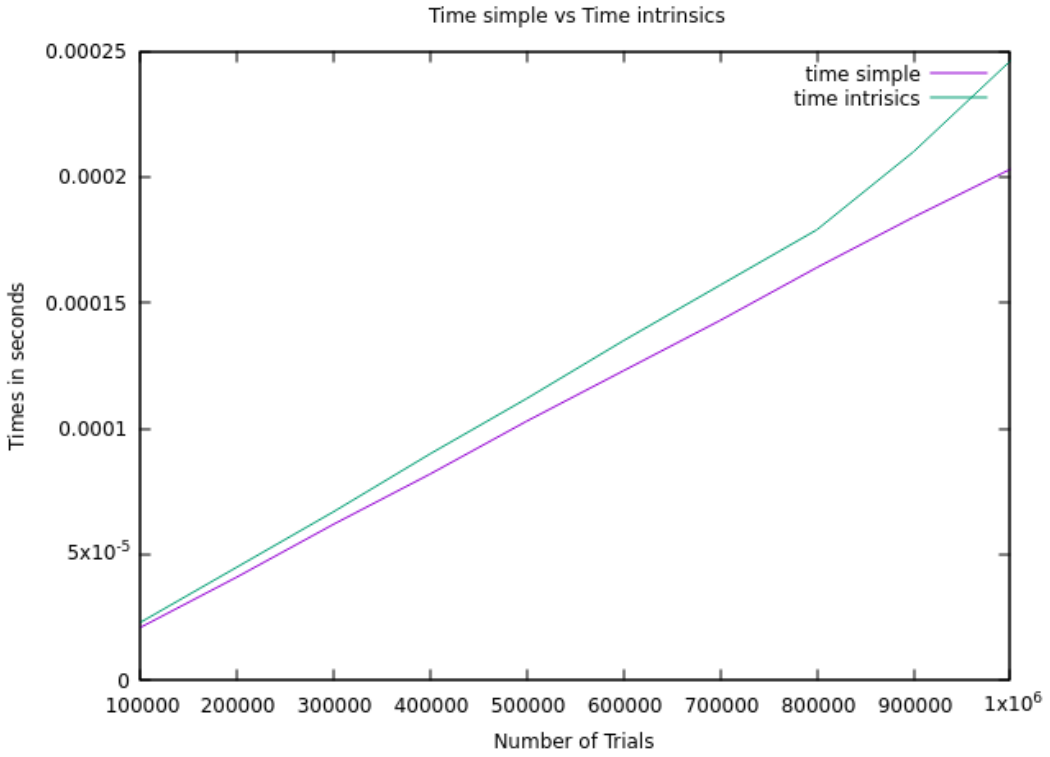
A continuación vamos a poner unas gráficas explicativas en cuanto al rendimiento entre ambos programas.
Esta primera gráfica se corresponde con el simple2.c, que tras ejecutarlo de 100000 en 100000 hasta un tamaño de 1000000, hemos obtenido los siguientes resultados.



Gráfica sobre simple2_intrinsics.c:



Gráfica comparativa entre simple2.c y simple2_instrinsics.c:



Podemos apreciar como las gráficas son lineales.Como se puede ver claramente la vectorización logra unos resultados bastante superiores en comparación con los que no están vectorizados siendo este mucho más rápido y eficiente que la versión no vectorizada (simple2.c).
En las tres gráficas propuestas el eje X se corresponde con el número de pruebas realizadas y el eje Y se corresponde con el número de segundos que tarda en realizar la iteración.
Como podemos apreciar la versión vectorizada comienza en 0.00003 segundos, esto indica que va a ir más rápido que la versión no vectorizada.

- Exercise 3:
    - The program includes two loops. The first loop (indicated as Loop 0) iterates over the arguments applying the algorithm to each of them. The second loop (indicated as Loop 1) computes the grey scale algorithm. Is this loop optimal to be vectorized? Why?

No es óptimo vectorizar el segundo loop, esto se debe a que a pesar de que se almacena todo en un array unidimensional, tras cada iteración se va a acceder a una nueva posición de memoria a la que están separados por una distancia.

- Provide the source code of the auto-vectorized version of the code. Explain the changes in the code to help the compiler to vectorize the loop.

En este caso en el programa autovectorized el programa va a contener un array el cual contendrá una imagen que va a ser unidimensional. Tras hacer los respectivos cálculos el programa irá dando saltos por el array utilizando el offset para poder moverse por la memoria que contiene la imagen que queremos.
La principal diferencia es que dichos saltos que hace por el array que comentamos se hacen a través de la memoria en vez de recorrer el array de seguido como hace el programa greySimple.c.

```
        continue;
    }

    gettimeofday(&ini,NULL);
    // RGB to grey scale
    int r, g, b;
    for (int j = 0; j < height; j++)
    {
        for (int i = 0; i < width; i++)
        {
            getRGB(rgb_image, width, height, 4, j, i, &r, &g, &b);
            grey_image[i * width + j] = (int)(0.2989 * r + 0.5870 * g + 0.1140 * b);
        }
    }

    stbi_write_jpg(grey_image_filename, width, height, 1, grey_image, 10);
```

- Provide the source code after manually vectorizing the code. Explain your solution.

Para la realización del manual_vectorized vamos a usar **AVX2**.

```
__m256 value = _mm256_setr_ps(0.2989f, 0.5870f, 0.1140f, 0.0f, 0.2989f, 0.5870f, 0.1140f, 0.0f);
__m256 tot;
__m256 tot1;
__m128i grey;
__m256i of = _mm256_setr_epi32(0, 4, 1, 5,0,0,0,0);


gettimeofday(&ini,NULL);
// RGB to grey scale
int r, g, b;
for (int j = 0; j <  height; j++)
{
    for (int i = 0; i<width; i+=4)
    {
        __m128i datal = _mm_loadl_epi64((__m128i *)(rgb_image + (i + width * j) * 4));
        __m128i datah = _mm_loadl_epi64((__m128i *)(rgb_image + (i + width * j) * 4 + 8));
        // Convertimos a entero de 32 bits
        __m256i data32bI = _mm256_cvtepu8_epi32(datal);
        __m256i data32bH = _mm256_cvtepu8_epi32(datah);
        // Convertimos a float (single precision)
        __m256 datafloatI = _mm256_cvtepi32_ps(data32bI);
        __m256 datafloatH = _mm256_cvtepi32_ps(data32bH);
        // Multiplicamos por el vector de pesos
        __m256 mulI = _mm256_mul_ps(datafloatI, value);
        __m256 mulH = _mm256_mul_ps(datafloatH, value);
        // Sumamos los elementos
        tot1 = _mm256_hadd_ps(mulI, mulH);
        // Redondeamos los valores del vector de su valor a float a integer
        tot = _mm256_hadd_ps(tot1, tot1); // tras segundo hadd hemos añadido 4byte por cada pixel

        // permutamos los valores para que queden en el orden correcto
        tot = _mm256_permutevar8x32_ps(tot, of);
        // Convertimos a entero de 32 bits
        grey = _mm_cvtps_epi32(_mm256_extractf128_ps(tot, 0));
        // Guardamos los valores en el vector de salida
        //__m128i arr = _mm_setr_epi8(0, 1 * 4, 2 * 4, 3 * 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);

        /*grey_image[(j * width) + i] = grey[0];
        grey_image[(j * width) + i + 1] = grey[1];
        grey_image[(j * width) + i + 2] = grey[2];
        grey_image[(j * width) + i + 3] = grey[3];*/

        uint32_t *ptr = (__m128i *)&grey;

        grey_image[(j * width) + i] = ptr[0];
        grey_image[(j * width) + i + 1] = ptr[1];
        grey_image[(j * width) + i + 2] = ptr[2];
        grey_image[(j * width) + i + 3] = ptr[3];
```

Este programa básicamente lo que hace modificar las tonalidades de rojo, azul y verde de una imagen que se recibe por argumento. Para ello mediante la sentencia : “uint8_t *grey_image = malloc(width * height);”, lo que va hacer va a ser reservar memoria para un array unidimensional de un tamaño de ancho por largo (de la imagen que le pasamos). Como está dentro de un bucle esta sentencia lo que va hacer es recorrer cada píxel de la imagen y modificará el color multiplicando cada color por unos coeficientes específicos según el color que sea, y una vez se haya terminado de compilar y ejecutar el programa nos guardará el resultado en un fichero .jpg .

Lo primero que hacemos en el bucle es cargar un vector de 64 bits de enteros en memoria para que se puedan acceder a dichas posiciones. Tras esto realizamos una conversión cambiándolos a enteros de 32 bits y luego a float. Con esta última conversión de enteros a float podremos pasar de blanco a negro con la fórmula proporcionada.
Tras esto multiplicamos los colores de cada píxel por su coeficiente correspondiente que hará un split en el vector de 512 bits, obteniendo uno nuevo de 256 bits con los que procederemos a realizar las sumas dos horizontales para obtener así el tripo de gris que le corresponderá a cada píxel. Estas operaciones nos obligan a realizar una serie de permutaciones, asi que mediante una variable que hace de offset (en nuestro caso es of) y junto con los cuatro valores

superiores realizaremos la permutación.Y tras esto podremos devolverlos a sus posiciones correspondientes.

- Fill in a table with time and speedup results compared to the original version and auto-vectorized version for images of different resolutions (SD, HD, FHD, UHD-4k, UHD-8k). You must include a column with the fps at which the program would process. Discuss the results.

Para este ejercicio hemos compilado con la flag -O3 el programa autovectorizado y sin la flag el fichero greyScale.c

Imagen	Normal		Autovectorized		SpeedUp
	Tiempo(s)	FPS	Tiempo(s)	FPS	
4k	0.674501	1.4826	0.193833	5.1591	3,47980478
8k	4.888153	0.2046	1.931836	0.5176	2,530314685
SD	0.020591	48.5649	0.008142	122.82	2,528985507
HD	0.068695	14.5571	0.031287	31.9622	2,195640362
FHD	0.160257	6.24	0.051464	19.4311	3,113963159

Como podemos ver, claramente la nueva versión autovectorizada proporciona tiempos menores a la versión normal. Debido a esto la calidad de la imagen también se verá afectada aumentado notoriamente, ya que como tiene que recorrer más píxeles el trabajo será mayor al igual que la calidad.