**ASIGNATURA Computación de altas prestaciones**

**Práctica1**

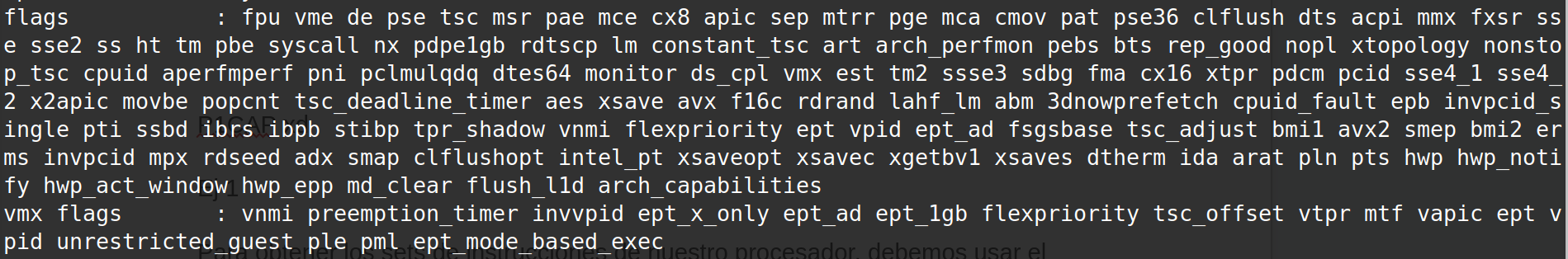
**Vector processing and SIMD**

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).

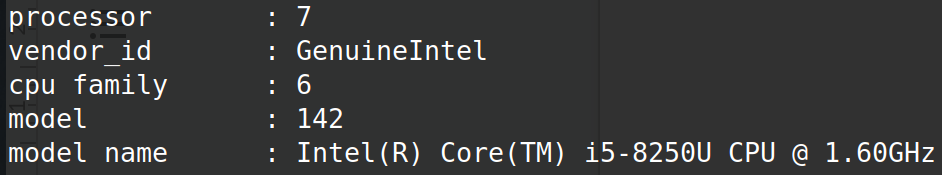
* Exercise 1:
  + Identify your CPU model and list the supported SIMD instructions.

Para obtener los sets de instrucciones de nuestro procesador, debemos usar el comando “cat /proc/cpuinfo“, nos fijamos en cualquiera de los cores (ya que todos son identicos en características) y observamos el campo de “flags”.

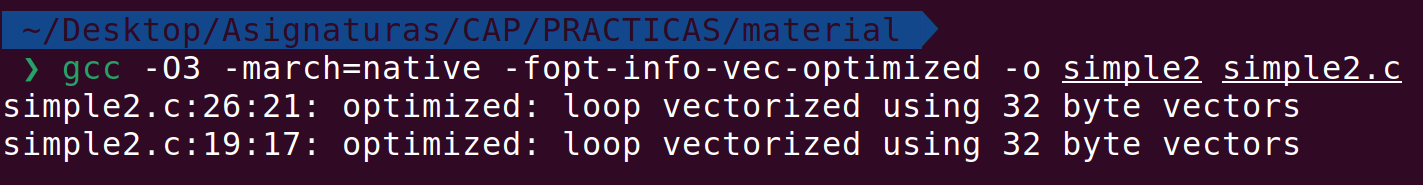
En nuestro caso, nuestro cpu tiene los siguientes sets de instrucciones, incluyendo **sse, sse2, avx**… entre otros:



Nuestro modelos de procesador se puede ver que cuenta con 8 nucleos logicos (son 4 fisicos pero aparecen 8 por el Hyper Threading), de fabricante Intel, un i5-8250U con frecuencia base a 1.60GHz.



Ejecutamos el comando “gcc -O3 -march=native -fopt-info-vec-optimized -o simple2 simple2.c” y obtenemos este resultado:



Nos optimiza dos bucles, uno en la línea 26 y otro en la linea 19, usando vectorización de 32 bytes.

Con este otro comando generamos un ejecutable no vectorizado: gcc -O3 -march=native -fno-tree-vectorize -fopt-info-vec-optimized -o simple2\_no\_vec simple2.c

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

Generamos el codigo de instrucciones con los siguientes comandos:

*gcc -S -O3 simple2.c*

*mv simple2.s simple2\_o3.s*

*gcc -S -O3 -fno-tree-vectorize simple2.c*

*mv simple2.s simple2\_o3\_native.s*

*diff simple2\_o3.s simple2\_o3\_native.s > diferencias.txt*

Con este último comando, guardamos en diferencias.txt las diferencia en el código de los dos ficheros:

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

Comparamos los dos códigos generados y observamos una clara diferencia en el código:

La diferencia más clara se encuentra a la hora de manejar los datos que van a ser partícipes en las operaciones del programa. En el archivo simple2\_o3.s, los datos a manejar son registros SIMD alineados en memoria. Esto se ve, ya que se usan instrucciones como: “movapd”. En cambio, en el archivo simple\_o3\_native.s, se utilizan registros normales, ya que no se aplica vectorización en el compilador. Esto se ve en instrucciones como “movsd” las cuales trabajan con registros estándar.

* 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.

#include <stdlib.h>

#include <stdio.h>

#include <x86intrin.h>

#include <sys/time.h>

#define ARRAY\_SIZE 2048

#define NUMBER\_OF\_TRIALS 300000

/\*

\* Statically allocate our arrays. Compilers can

\* align them correctly.

\*/

static double a[ARRAY\_SIZE], b[ARRAY\_SIZE], c;

int main(int argc, char \*argv[]) {

int i,t;

struct timeval tv, tv2;

/\* Populate A and B arrays \*/

for (i=0; i < ARRAY\_SIZE; i+=4) {

/\*

Como queremos vectorizarlo, lo que hacemos es añadir el i+=4 en vez de i++,

de forma que ahora trabajamos con vectores de 4 posiciones. Para continuar haciendo las operaciones de forma correcta

el vector debe cubrir todos los valores anteriores. De ahí la suma "d".

\*/

\_\_m256d d = {i, i + 1.0, i + 2.0, i + 3.0};

\_\_m256d e = {i + 1.0, i + 2.0, i + 3.0, i + 4.0};

\_mm256\_store\_pd(&b[i], d);

\_mm256\_store\_pd(&a[i], e);

}

\_\_m256d mm = {1.0001, 1.0001, 1.0001, 1.0001};

\_\_m256d sum = {0.0, 0.0, 0.0, 0.0}; // to hold partial sums

/\* Perform an operation a number of times \*/

// Get the current time

if (gettimeofday(&tv, NULL) == -1) {

perror("gettimeofday1");

return 1;

}

for (t=0; t < NUMBER\_OF\_TRIALS; t++) {

for (i=0; i < ARRAY\_SIZE; i+=4) {

// 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);

}

}

if (gettimeofday(&tv2, NULL) == -1) {

perror("gettimeofday2");

return 1;

}

long microseconds = tv2.tv\_usec - tv.tv\_usec;

// Get sum[2], sum[3]

\_\_m128d xmm = \_mm256\_extractf128\_pd (sum, 1);

// Extend to 256 bits: sum[2], sum[3], 0, 0

\_\_m256d ymm = \_mm256\_castpd128\_pd256(xmm);

// Perform sum[0]+sum[1], sum[2]+sum[3], sum[2]+sum[3], 0+0

sum = \_mm256\_hadd\_pd (sum, ymm);

// Perform sum[0]+sum[1]+sum[2]+sum[3]…

sum = \_mm256\_hadd\_pd (sum, sum);

c = sum[0];

printf("%lf", c);

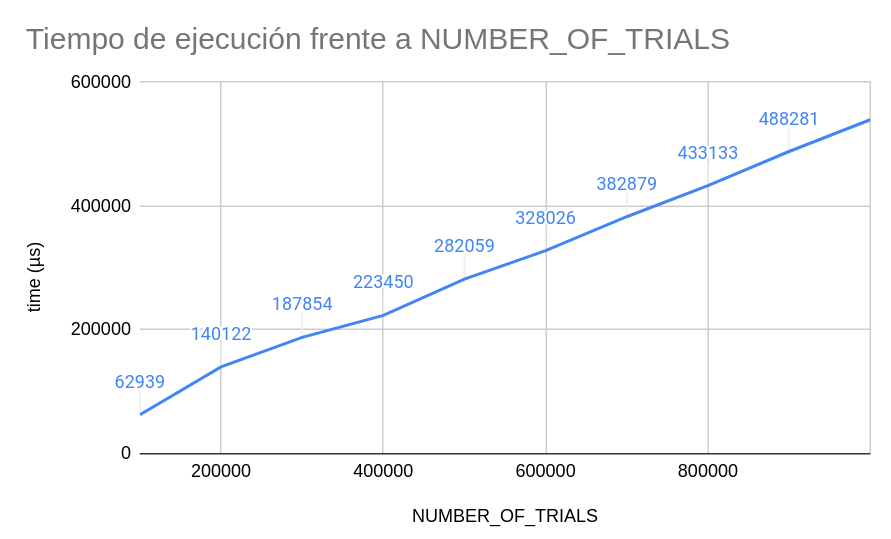
printf("Tiempo de ejecución para %d iteraciones: %ldms.",NUMBER\_OF\_TRIALS, microseconds);

return 0;

}

Como queremos vectorizar la suma de un array en un vector de 4 posiciones, lo que hacemos es añadir el i+=4 en vez de i++, de forma que ahora trabajamos con vectores de 4 posiciones. Para continuar haciendo las operaciones de forma correcta el vector debe cubrir todos los valores anteriores. De ahí la suma "d". De esta forma tenemos que realizar 4 veces menos de operaciones ya que se trabaja con conjuntos de 4 datos a la vez.

* + 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.



Como podemos observar, el incremento de tiempo es prácticamente lineal, esto se debe a que el coste de cómputo de la suma de elementos de un array vectorizado de 4 en 4 en un incremento de veces supone un aumento de coste en computación proporcional a la cantidad de iteraciones.

* 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?

Ambos bucles se prestan para ser vectorizados, debido a que las operaciones con los datos son independientes e idénticas para cada conjunto de los mismos. Lo que nos permite vectorizar el proceso.

Hay que anotar que es necesario acceder a los datos de forma “horizontal” y el offset  *“offset = im + (x + width \* y) \* nchannels;”* calcularlo de esta forma, al igual que en el segundo bucle que guardamos los valores de esta forma: “*grey\_image[i \* height + j] = (int)(0.2989 \* r + 0.5870 \* g + 0.1140 \* b);”*

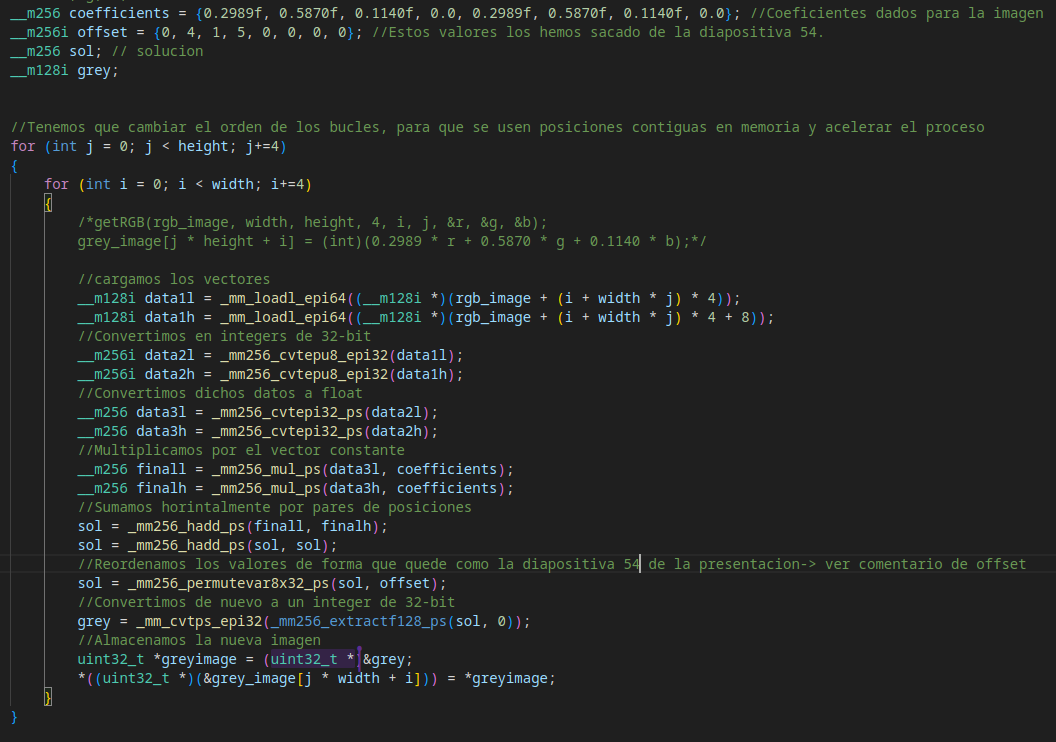
* + Provide the source code of the auto-vectorized version of the code. Explain the changes in the code to help the compiler to vectorize the loop.

linea 13: unsigned char \*offset = im + (x + height \* y) \* nchannels;

linea 83: grey\_image[j \* height + i] = (int)(0.2989 \* r + 0.5870 \* g + 0.1140 \* b);

Este cambio ha sido realizado para poder recorrer las posiciones de forma más óptima. Esto se debe a que ahora, los datos, se encuentran y recorren en posiciones contiguas de memoria, mejorando el rendimiento global del programa.

* + Provide the source code after manually vectorizing the code. Explain your solution.



Como se puede ver en la imagen anterior, hemos hecho varios cambios al código solicitado. Hemos seguido la guía indicada para la práctica 1.

* + 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.

| Calidad | Version Original | | Version Auto-Vec | | SpeedUp |
| --- | --- | --- | --- | --- | --- |
| Tiempo | FPS | Tiempo | FPS |
| SD | 0.005344 | 187.126 | 0.003838 | 260.552 | 1.3924 |
| HD | 0.019294 | 51.829 | 0.014655 | 68.236 | 1.3165 |
| FHD | 0.042520 | 23.518 | 0.031745 | 31.501 | 1.3394 |
| 4k | 0.173026 | 5.779 | 0.143119 | 6.987 | 1.2089 |
| 8k | 0.921894 | 1.085 | 0.653223 | 1.531 | 1.4113 |

# Como se puede observar, la versión auto-vectorizada del código tarda, por regla general, menos tiempo en ejecutar que la versión original. Otra cosa a remarcar es que a medida que mejora la calidad de imagen, los valores obtenidos muestran mayores diferencias. Esto se puede ver mediante el SpeedUp. Ya que el aumento de carga se realiza en la parte paralelizable(Los bucles) por lo que tiene sentido que cuanto mayor sea la carga, más se note esa diferencia.