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.

Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz

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 pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_single pti ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid mpx rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln pts hwp hwp_notify hwp_act_window hwp_epp md_clear flush_l1d arch_capabilities

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

❯ diff --color simple2_o3.s simple2_o3_native.s

Bloque 1:

```
9a10
>	xorl	%eax, %eax
12,17d12
<	movdqa		.LC0(%rip), %xmm2
<	movdqa		.LC1(%rip), %xmm4
<	movdqa		.LC2(%rip), %xmm3
<	movq	%rcx, %rax
<	movq	%rdx, %rsi
<	leaq	16384(%rcx), %rdi
```

Bloque 2:

```
19,34c14,22
<    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
```

Bloque 3:

```
37c25
<    movapd          .LC3(%rip), %xmm3
---
>    movsd   .LC0(%rip), %xmm2
```

Bloque 4:

```
44,49c32,35
<    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
```

Bloque 5:

```
67,68c53,54
<    .section .rodata.cst16,"aM",@progbits,16
<    .align 16
---
>    .section .rodata.cst8,"aM",@progbits,8
>    .align 8
```

Bloque 6:

```
70,89d55
<    .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
```

- 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.

```
❯ ./simple
4194513.817600
❯ ./intrinsics
4194513.817600
```

El primer bucle, queda de la siguiente forma:

```
__m256d vb = {0, 1, 2, 3};
__m256d va = {1, 2, 3, 4};

__m256d cons = {4, 4, 4, 4};

/* Populate A and B arrays */
for (i = 0; i < ARRAY_SIZE; i += 4)
{
    _mm256_store_pd(&b[i], vb);
    vb = _mm256_add_pd(vb, cons);

    _mm256_store_pd(&a[i], va);
    va = _mm256_add_pd(va, cons);

}
```

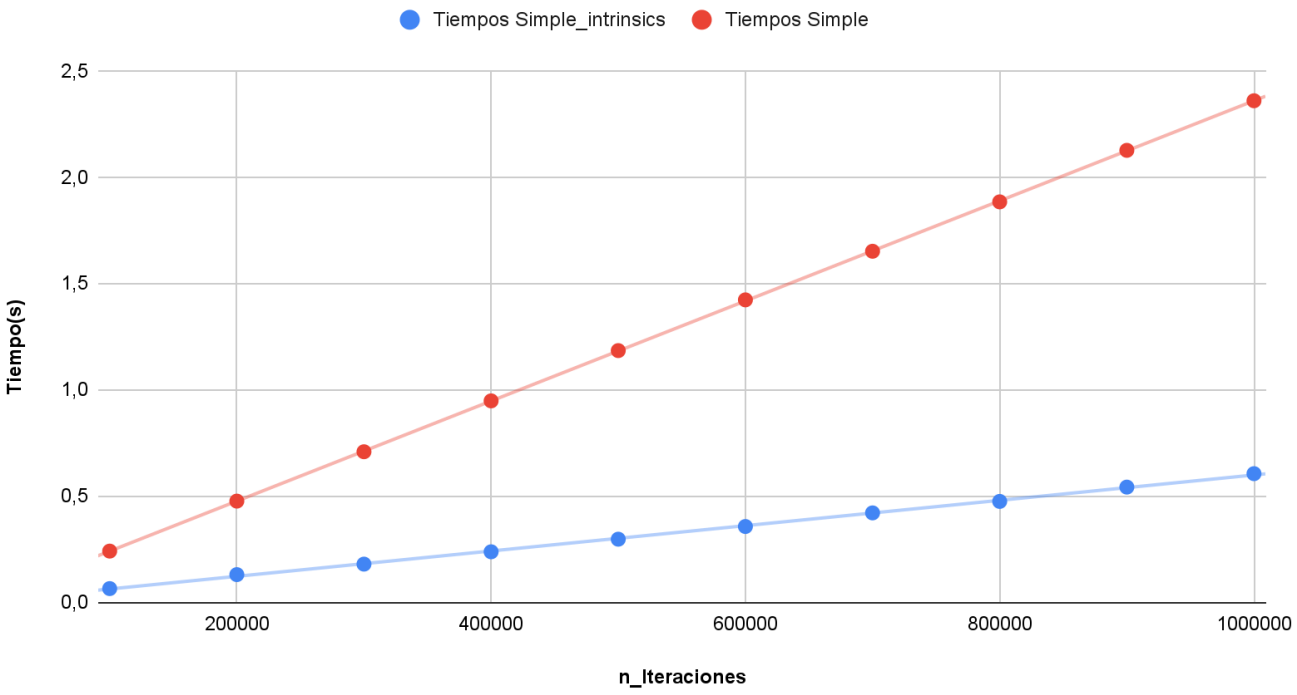
El segundo bloque queda de la siguiente forma:

```
__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 */
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);

        // c += m*a[i] + b[i];
    }
}

// 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];
```

- 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.

Tiempos Simple_intrinsics	Tiempos Simple
0,06320	0,239581
0,128949	0,474872
0,178317	0,707135
0,236446	0,946552
0,295623	1,183117
0,355387	1,421803
0,419435	1,650559
0,473498	1,882969
0,540748	2,125369
0,604040	2,358790

Tal y como observamos, los tiempos con intrínsecos son claramente menores.

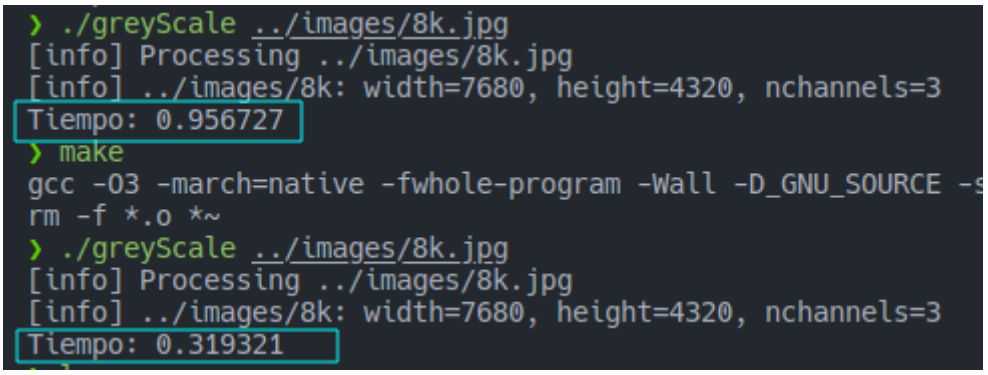


- Exercise 3:
    - The program includes two loops. The first loop (indicated as Loop 0) iterates over the arguments applying the algorithm to each of them. The second loop (indicated as Loop 1) computes the grayscale algorithm. Is this loop optimal to be vectorized? Why?

Consideramos que no es óptimo vectorizar el segundo bucle, ya que el segundo trata pixel a pixel las imágenes y tiene un tamaño demasiado grande como para que podamos vectorizarlo de manera óptima.

    - 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.

Debido a que el orden de acceso a los datos es importante para la velocidad a la hora de recorrer matrices, hemos cambiado el orden de acceso para que recorra las columnas primero para maximizar la eficiencia de acceso a la memoria. Antes estaba al reves, con el acceso a filas primero, y gracias a eso podemos observar como el algoritmo mejora su tiempo de ejecución:



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

```
for(int i = 0, j = 0; j < imageSize; i += 16, j += 4) {

    __m128i* data_ptr_one = (__m128i*)(rgb_image + i);
    __m128i* data_ptr_two = (__m128i*)(rgb_image + i + 8);
    __m128i filas = _mm_loadl_epi64(data_ptr_one);
    __m128i columnas = _mm_loadl_epi64(data_ptr_two);

    // Extendemos los vectores y los convertimos a floats
    __m256i extendedFilasInt = _mm256_cvtepu8_epi32(filas);
    __m256 extendedFilasFloat = _mm256_cvtepi32_ps(extendedFilasInt);
    __m256i extendedColumnasInt = _mm256_cvtepu8_epi32(columnas);
    __m256 extendedColumnasFloat = _mm256_cvtepi32_ps(extendedColumnasInt);

    // generamos el vector de coeficientes y lo usamos para multiplicar los vectores por pares de pixeles.
    __m256 coeficientes = _mm256_set_ps(0.0, 0.1140, 0.5870, 0.2989, 0.0, 0.1140, 0.5870, 0.2989);
    __m256 par1 = _mm256_mul_ps(extendedFilasFloat, coeficientes);
    __m256 par2 = _mm256_mul_ps(extendedColumnasFloat, coeficientes);

    // hacemos un horizontal add, dos  veces porque no se completa en una sola.
    __m256 h_add = _mm256_hadd_ps(par1, par2);
    h_add = _mm256_hadd_ps(h_add, h_add);

    // permutamos el vector final y lo extraemos para el outcome.
    __m256 permutado = _mm256_permutevar8x32_ps(h_add, _mm256_set_epi32(0, 0, 0, 0, 5, 1, 4, 0));
    __m128 outcome = _mm256_extractf128_ps(permutado, 0);

    for (int k = 0; k < 4; k++) {
        grey_image[j + k] = (int) outcome[k];
    }
}
```

Tal y como hemos indicado en el código con los comentarios, separamos los pixeles de dos en dos, extendemos los vectores y los convertimos a floats, después generamos el vector de coeficientes y lo usamos para multiplicar los vectores pòr pares de pixeles, siendo par1, el pixel 1 y 2, y par2 el 3 y 4. Después haremos horizontal add, dos veces debido a que no acababa de hacerlo de una sola, y permutamos el vector y lo extraemos para sacarlo en la imagen.

- 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.

	greyscale auto (s)	greyscale manual (s)
8k	0,385949	0,356616
4k	0,124572	0,108843
FHD	0,055421	0,040849
HD	0,015936	0,016629
SD	0,005566	0,005998
	greyscale auto (FPS)	greyscale manual (FPS)
8k	3	3
4k	8	9
FHD	18	24
HD	63	60
SD	180	167

Debido a el tiempo que tardan en procesar la imagen, podemos observar cómo sería inviable tener este proceso para tamaños de 4k y 8k, con el manual podríamos acercarnos a una experiencia más cinematográfica con 24 fps, y si quisiéramos mantener ese procesamiento para ordenadores o móviles nos iríamos a un mínimo de HD para los 60 fps que vendrían bien debido a la habitual velocidad de refresco de 60hz.