

# TRABAJO DE ARQUITECTURA DE COMPUTADORES

Curso 2019-2020

## OBJETIVOS DEL TRABAJO

El trabajo persigue varios objetivos:

- Ilustrar las mejoras de rendimiento que pueden obtenerse en una máquina real mediante la programación de aplicaciones multihilo, aprovechando el grado de concurrencia del sistema.
- Programar las instrucciones SIMD que proporcionan algunas arquitecturas y valorar las mejoras de rendimiento que pueden obtenerse.
- Desarrollar habilidades de trabajo en grupo y documentación, básicas para un ingeniero en Informática.

## PLATAFORMA DE DESARROLLO

El trabajo se realizará sobre una máquina virtual con el software necesario preinstalado para la realización de las tareas. El usuario de esta máquina es “student” y la clave de acceso “workgroup”.

El número de CPUs con el que debe configurarse el hipervisor de la máquina virtual será el máximo posible del sistema sobre el que se instale, con un mínimo de dos. Para justificarlo, se debe adjuntar en la documentación final el contenido del fichero `/proc/cpuinfo` de la máquina virtual mostrando el número de procesadores y sus características.

En cuanto a la memoria, la máquina virtual debe configurarse con un mínimo de 3 GB, salvo que la máquina física tenga instalados menos de 6 GB, en cuyo caso debe reducirse a la mitad de la memoria disponible.

Los archivos necesarios para el desarrollo del trabajo en grupo se encuentran en un repositorio Git público alojado en Bitbucket. Antes de comenzar a trabajar es necesario que **uno y sólo uno de los miembros del grupo** haga un *fork* de dicho repositorio en un repositorio privado de Bitbucket, al que accederán todos los miembros del grupo. Este miembro pasará a ser el administrador del repositorio. La secuencia de operaciones a realizar por el administrador del repositorio es la siguiente:

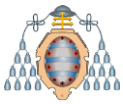
1. Entrar en la cuenta de Bitbucket usando un cliente web.
2. En la barra de navegación introducir la dirección <https://bitbucket.org/2acuniovi/2ac-teamwork/src/master/>, correspondiente al repositorio público. Aparecerá su contenido.
3. En la esquina superior izquierda aparece un signo +. Al hacer clic sobre él aparecen varias opciones para la creación de un repositorio. Debe elegirse la opción *Fork this repository*.
4. Elegir como nombre de repositorio el que aparece por defecto, añadiendo el nombre del grupo de trabajo. Por ejemplo, si el grupo de trabajo es el PL4-B, el nombre del repositorio debe ser `2ac-teamwork-pl4-b`. **Debe marcarse la casilla de verificación** para que el repositorio sea privado. En caso contrario el contenido del repositorio será público.
5. Dar al profesor de prácticas y a los miembros del grupo acceso de escritura al repositorio en la sección de *Settings > User and group access*. El profesor debe proporcionar su nombre de usuario. A partir de ese momento el repositorio estará disponible para todos los miembros del grupo y el profesor.

A continuación, todos los miembros del grupo deben clonar el repositorio creado dentro del directorio Documents de la máquina virtual, de forma análoga a como se explica en la sesión 0 de prácticas.

Se puede trabajar con el repositorio desde la interfaz de comandos tal como se explicó en la sesión 0, o empleando el sistema de control de versiones Git integrado en Visual Studio Code.

Se recomienda el empleo del repositorio para coordinar el desarrollo del trabajo en grupo.





## DESARROLLO DEL TRABAJO

El trabajo se desarrollará en dos fases, durante las cuales será necesario la realización de programas y la documentación en una memoria a entregar.

### FASE 1 (2 PUNTOS)

Durante esta fase, cada grupo debe realizar un programa monohilo secuencial en C/C++ que implemente operaciones sobre las componentes de imágenes digitales. Las características de los datos de las imágenes y el tipo de operaciones a realizar serán proporcionadas por el profesor a cada grupo, como complemento a estas instrucciones.

El programa monohilo a desarrollar consiste en la implementación básica de las operaciones y se tomará como referencia para el cálculo de las aceleraciones obtenidas en las implementaciones posteriores. Debe emplearse el esqueleto contenido en el proyecto **SingleThread**.

Este programa es análogo al desarrollado durante la sesión práctica 1.2. No obstante son diferentes las operaciones a realizar por el algoritmo. El programa tiene que medir únicamente el tiempo que tarda en ejecutarse la parte de procesamiento de imágenes del algoritmo y obviar tanto los tiempos de carga e inicialización de las imágenes como el tiempo necesario para salvar las imágenes en disco.

Para mejorar la resolución de la medida de intervalos de tiempo, el procesamiento de imágenes debería tener una duración entre 5 y 10 segundos, por lo que si una sola ejecución del algoritmo no dura lo suficiente **se repetirá (en bucle) el número de veces necesario para alcanzar esos tiempos**.

Una vez realizado el programa, éste debe ejecutarse 10 veces en modo **Release** desde la línea de comandos y calcular la media y la desviación típica del tiempo de respuesta, reflejando todas las medidas obtenidas y las estadísticas en una tabla de la documentación.

### Fase 2 (8 puntos)

Durante esta fase, cada grupo debe realizar una versión monohilo con extensiones multimedia y una versión multihilo del programa anterior para aprovechar las capacidades de paralelismo y concurrencia del sistema y así obtener una ganancia de rendimiento.

#### **Fase 2 - Primera parte (5 puntos)**

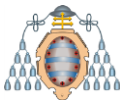
El segundo programa monohilo a desarrollar consiste en el empleo de las extensiones SIMD de la arquitectura para la mejora del rendimiento, para lo cual en esta parte se empleará el proyecto **SingleThread-SIMD**. Las extensiones SIMD se usarán para la implementación de las operaciones repetitivas que se llevan a cabo durante el procesamiento de las imágenes indicado por tu profesor. A estas extensiones también se las conoce como extensiones multimedia o vectoriales, pues en este campo de aplicación es habitual la realización de operaciones repetitivas muy simples sobre grandes vectores o matrices que codifican imágenes, audio y vídeo.

Las extensiones multimedia son opcionales y se agrupan en conjuntos de instrucciones. Dependiendo del procesador que incorpore el equipo pueden estar disponibles algunas y otras no. Por esta razón, el trabajo a realizar en primer lugar es identificar la lista de extensiones SIMD soportadas. Tenemos la lista completa en el archivo `cpuinfo`.

Debe comprobarse si el procesador soporta las siguientes extensiones SIMD, ordenadas de más antiguas a más modernas: MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AVX2 y AVX-512F. Las extensiones restantes no es necesario comprobarlas. Debe tenerse en cuenta que habitualmente el soporte de las extensiones más modernas incluye el soporte de las anteriores

El empleo de las extensiones SIMD se lleva a cabo usando lo que se denomina funciones intrínsecas (*intrinsics*). En apariencia son funciones de C, pero en realidad no lo son pues no son llamadas como las funciones habituales. Cada referencia a una función intrínseca en el código se traduce directamente en una instrucción ensamblador.



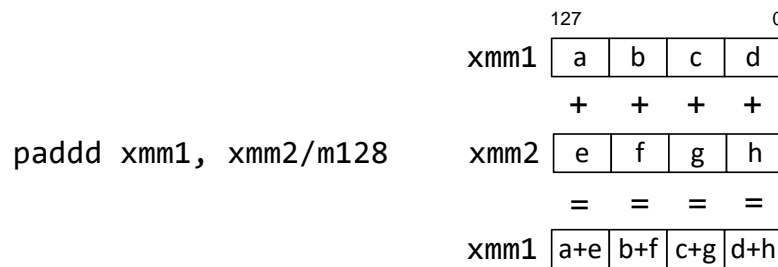


Por ejemplo, la función intrínseca `__m128i _mm_add_epi32 (__m128i a, __m128i b)` suma dos vectores (paquetes) `a` y `b` con 4 enteros de 32 bits y el resultado lo lleva a otro vector (paquete) de 4 enteros. Esta instrucción se traduce básicamente en la instrucción ensamblador

`paddq xmm1, xmm2/m128`

El registro `xmm1` es un registro de 128 bits usado como fuente y destino. El otro operando, `xmm2/m128`, es un registro fuente de 128 bits o un operando fuente de memoria de 128 bits. Los registros `xmm` fueron introducidos con las extensiones SSE2.

La instrucción realiza la operación `xmm1+xmm2 → xmm1`, tal como se muestra a continuación.



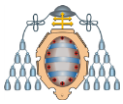
En el empleo de las funciones intrínsecas debe prestarse especial atención al alineamiento de memoria. Por ejemplo, cualquier acceso a operandos de tamaño 128 bits debe realizarse empleando direcciones de memoria múltiplos de 16 (para almacenar 128 bits hacen falta 16 bytes, es decir, 16 posiciones de memoria). El uso de accesos a memoria no alineados puede penalizar severamente el rendimiento de los programas y, en algunas arquitecturas de procesador, generar errores de ejecución ya que ese tipo de accesos están prohibidos.

Para ilustrar el empleo de las extensiones SIMD se presenta un programa que calcula la suma de 16 elementos de un vector de 18 reales de 32 bits usando paquetes de números reales de 256 bits de tamaño (8 flotantes en cada paquete). Para sumar los 16 elementos del array se realizan dos operaciones de suma, la primera sobre los 8 primeros elementos y la segunda sobre los 8 siguientes. En un caso real habría que repetir la operación iterando sobre el resto de elementos del array.

En el programa también se ilustra el protocolo más adecuado para trabajar con las extensiones multimedia y evitar al máximo los problemas de alineamiento de la memoria:

1. Se define una constante que es cuántos datos (`floats`) tenemos en cada paquete de 256 bits.
2. Declaramos variables del mismo tamaño que los paquetes a utilizar, en este caso `__m256`.
3. Se inicializan los arrays de datos `a` y `b`. Suponemos que son datos de entrada a nuestro programa.
4. Para almacenar el resultado del algoritmo se declara el array `c` alineado al tamaño de paquete (`__m256`) que vamos a utilizar en este caso. Su tamaño tiene que ser igual al número de paquetes necesarios para almacenar todos los datos del resultado. Se utilizarán instrucciones vectoriales para inicializarlo a -1.
5. Como los arrays de datos `a` y `b` no tienen por qué estar alojados en posiciones de memoria alineadas al tamaño de paquete usado (`__m256`), para acceder a su contenido es necesario utilizar una instrucción vectorial de carga desalineada. Utilizamos como destino una variable temporal por cada array.
6. Se realiza la suma vectorial de un paquete de datos con la instrucción SIMD adecuada.
7. Se repiten los pasos 5 y 6 con el siguiente paquete de datos.





```
// Example of use of intrinsic functions
// This example doesn't include any code about image processing.
// The image processing code must be added by the students and remove the unnecessary code.

#include <immintrin.h> // Required to use intrinsic functions
#include <malloc.h>
#include <stdio.h>

#define TAM 18 // Array size. Note: It is not a multiple of 8
#define ELEMENTSPERPACKET (sizeof(__m256)/sizeof(float))

int main()
{
    // Data arrays to sum. Might be or not memory aligned to __m256 size (32 bytes)
    float a[TAM], b[TAM];

    // Calculation of the size of the results array
    // How many 256 bit packets fit in the array?
    int NPACKets = (TAM * sizeof(float)/sizeof(__m256));
    // If is not an exact number we need to add one more packet
    if ((TAM * sizeof(float))%sizeof(__m256) != 0)
        NPACKets += 1;

    // Create an array aligned to 32 bytes (256 bits) memory boundaries to store the sum.
    // Aligned memory access improves performance
    float *c = (float *)__mm_malloc(sizeof(__m256) * NPACKets, sizeof(__m256));

    // 32 bytes (256 bits) packets. Used to stored aligned memory data
    __m256 va, vb;

    // Initialize data arrays
    for (int i = 0; i < TAM; i++) {
        *(a + i) = (float)i; // a = 0, 1, 2, 3, ...
        *(b + i) = (float)(2 * i); // b = 0, 2, 4, 6, ...
    }

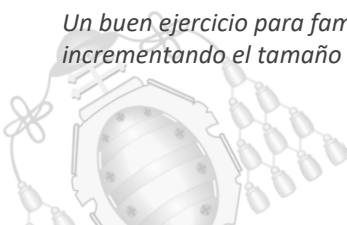
    // Set the initial c element's value to -1 using vector extensions
    *(__m256 *) c = __mm256_set1_ps(-1);
    *(__m256 *) (c + ELEMENTSPERPACKET) = __mm256_set1_ps(-1);
    *(__m256 *) (c + ELEMENTSPERPACKET * 2) = __mm256_set1_ps(-1);
    // Data arrays a and b do not have to be memory aligned to __m256 data (32 bytes)
    // so we use intermediate variables to avoid running errors.
    // We make an unaligned load of va and vb
    va = __mm256_loadu_ps(a); // va = a[0][1]...[7] = 0, 1, 2, 3, 4, 5, 6, 7
    vb = __mm256_loadu_ps(b); // vb = b[0][1]...[7] = 0, 2, 4, 6, 8, 10, 12, 14

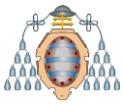
    // Performs the addition of two aligned vectors, each vector containing 8 floats
    *(__m256 *) c = __mm256_add_ps(va, vb); // c[0][1]...[7] = 0, 3, 6, 9, 12, 15, 18, 21

    // Next packet
    // va = a[8][9]...[15] = 8, 9,10,11,12,13,14,15
    // vb = b[8][9]...[15] = 16,18,20,22,24,26,28,30
    // c = c[8][9]...[15] = 24,27,30,33,36,39,42,45
    va = __mm256_loadu_ps(a + ELEMENTSPERPACKET);
    vb = __mm256_loadu_ps(b + ELEMENTSPERPACKET);
    *(__m256 *) (c + ELEMENTSPERPACKET) = __mm256_add_ps(va, vb);

    // Print resulting data from array addition
    for (int i = 0; i < TAM; i++)
        printf("\nc[%d]: %f", i, *(c + i));
    // Free memory allocated using __mm_malloc
    // It has to be freed with __mm_free
    __mm_free(c);
}
```

Un buen ejercicio para familiarizarse con el uso de las funciones intrínsecas sería modificar el ejemplo anterior incrementando el tamaño del array (TAM) y convertir el procesamiento en un bucle.





La lista de funciones intrínsecas y su descripción puede encontrarse en la página web:

<https://software.intel.com/sites/landingpage/IntrinsicsGuide/>

En la sección de bibliografía se suministran una serie de enlaces a documentos y páginas que explican gráficamente el funcionamiento de las instrucciones vectoriales y de algunas de las instrucciones con operaciones menos usuales.

Las funciones intrínsecas deben usarse en aquellas partes del programa en las cuales sea necesario realizar la misma operación sobre varios datos a la vez, tal como ocurre con las operaciones que deben realizarse sobre las imágenes. Para cada una de las operaciones sobre vectores del problema asignado debe indicarse de forma justificada las funciones intrínsecas que se usarán para reducir tanto como sea posible el tiempo de ejecución.

Si para mejorar la resolución en la toma de tiempos del algoritmo secuencial ha sido necesario repetirlo un determinado número de veces, para que las comparaciones de rendimiento sean válidas, será necesario hacer lo mismo y repetir el **mismo** número de veces la implementación del algoritmo usando las instrucciones SIMD.

Una vez realizado el programa, éste debe ejecutarse 10 veces en modo **Release** desde la línea de comandos para calcular la media y la desviación típica del tiempo de respuesta, reflejando en la memoria del trabajo todas las medidas obtenidas, estadísticas y aceleración respecto a la versión secuencial realizada en la Fase 1. Debe justificarse la aceleración obtenida.

### ***Fase 2 – segunda parte (3 puntos)***

Debe convertirse la solución básica (sin las instrucciones SIMD) en multihilo para aprovechar las capacidades de concurrencia del sistema. Para facilitar la tarea se suministra el proyecto **MultiThread** correctamente configurado, aunque vacío. El número de hilos a emplear coincidirá con el nivel de concurrencia del sistema. Por ejemplo, si el sistema dispone de 1 procesador de 4 núcleos, éste dispone de 4 CPU. Si las CPU disponen de capacidad de *Hyperthreading*, el máximo grado de concurrencia será  $1 \times 4 \times 2 = 8$ , pues el sistema podrá ejecutar concurrentemente 8 hilos (sin emplear tiempo compartido).

La programación con hilos en Linux se ha realizado en la sesión 1.2 de prácticas.

Es necesaria la paralelización del algoritmo, es decir, la descomposición del algoritmo global en algoritmos más simples que puedan asignarse a hilos. Por ejemplo, la suma de los elementos de un vector de dimensión  $16 \times N$  puede descomponerse en 16 sumas de vectores de dimensión  $N$ , cada una de las cuales se puede llevar a cabo en un hilo diferente. Finalmente, el hilo principal recoge la suma parcial de cada uno de los 16 hilos y genera la suma total.

Como en los dos casos anteriores, si ha sido necesario repetir en bucle el algoritmo secuencial para obtener medidas adecuadas de tiempo, habrá que realizar el mismo número de repeticiones en la ejecución del algoritmo en su versión multihilo. Dentro del algoritmo multihilo hay que incluir el tiempo de creación y destrucción de los hilos.

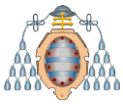
Al igual que con los programas anteriores, una vez realizado el programa, éste debe ejecutarse 10 veces en modo **Release** desde la línea de comandos para calcular la media y la desviación típica del tiempo de respuesta, reflejando en la memoria todas las medidas obtenidas, estadísticas y justificando la aceleración obtenida respecto a la implementación básica.

## **ENTREGAS**

### ***Fase 1***

La entrega consistirá en una pequeña memoria dónde se incluirá la información solicitada en la descripción de esta fase. En el documento se detallará el trabajo de cada uno de los miembros del grupo y su porcentaje de participación. En la entrega también se incluirá el código fuente del programa. La entrega será única y





será subida al Campus Virtual en un único archivo comprimido, al foro del grupo, por uno de los miembros del mismo. El plazo máximo de entrega será el viernes 25 de octubre.

## Fase 2

La entrega consistirá en una memoria dónde se incluirá la información solicitada en la descripción de esta fase. En el documento se detallará el trabajo de cada uno de los miembros del grupo y su porcentaje de participación. En la entrega también se incluirá el código fuente de los programas correspondientes a la fase 2. La entrega será única y será subida al Campus Virtual en un único archivo comprimido, al foro del grupo, por uno de los miembros del mismo. El plazo máximo de entrega será el viernes 13 de diciembre a las 20h.

## REFERENCIAS


<https://software.intel.com/sites/landingpage/IntrinsicsGuide/> .- Referencia principal de las instrucciones intrínsecas de Intel. Muestra el nombre de la función intrínseca, los parámetros que recibe y el resultado que devuelve. También muestra las instrucciones máquinas a las que equivale dicha función y una descripción de la operación a nivel de bits.

<http://tommesani.com/index.php/simd/34-mmx-primer.html> .- Dedicado a las instrucciones MMX y SSE. Tiene unos esquemas gráficos que explican bastante bien el funcionamiento de algunas instrucciones especiales.

<https://www.codeproject.com/Articles/874396/Crunching-Numbers-with-AVX-and-AVX> .- Artículo que explica el uso de AVX para el cálculo numérico. Muy interesantes las secciones 3.1 *Data Types* y 3.2 *Function Naming Conventions* que explican los tipos de datos y mecanismo de nombrado de las funciones intrínsecas.

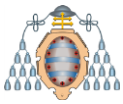
## APÉNDICE A.- EL ENTORNO VISUAL STUDIO CODE

Para el desarrollo de las aplicaciones en la máquina Linux se va a utilizar el IDE multiplataforma Visual Studio Code de Microsoft. El uso básico del entorno es muy intuitivo aunque aquí se incluye una pequeña guía de uso.

- Para abrir el IDE.- Pulsar sobre el icono  situado en la barra lateral de accesos directos del escritorio.
- Abrir un WorkSpace.- Ir al menú "File→Open Workspace". Los proyectos están en la carpeta Documentos. Abrir el documento *NombreProyecto.code-workspace*.
- Editar.- Seleccionar el archivo *.cpp* que se desea editar. Para crear archivos/directorios nuevos pulsar el icono correspondiente situado al lado del nombre del workspace.
- Depurar.- Con el archivo deseado seleccionado en el editor ir al menú a "Debug→Start Debugging" o pulsar F5. El ejecutable se creará en la carpeta *Debug* del proyecto.
  - Para añadir un breakpoint.- Pulsar con el botón en el lateral de la línea en la que se desea que se detenga el programa hasta que se muestre un punto rojo.
  - Inspeccionar variables.- Por defecto, el depurador muestra el estado de todas las variables bajo la pestaña lateral "Variables-Locals"
  - Añadir una variable a inspección.- Para poder ver la evolución del valor de una variable, seleccionar la variable en el editor y "Botón derecho→Debug:Add to Watch". La variable aparecerá en la pestaña lateral "Watch".
  - Volver al editor.- Una vez ha finalizado el programa, para volver al editor, seleccionar en el menú "View→Explorer" o "Ctrl+Shift+E".
- Generar versión **Release**.- Para generar la versión **Release** del programa seleccionar la opción del menú "Terminal→Run Build Task" o pulsar "Ctrl+Shift+B". El ejecutable se creará en la carpeta principal del proyecto.
- Compilar el programa sin generar el ejecutable.- Para compilar en busca de fallos de sintaxis sin necesidad de crear el ejecutable seleccionar "Terminal→Run Task..." y escoger/buscar la tarea de nombre "g++ only compile active file".







- Visual Code trae soporte integrado para Git como gestor de control de versiones. En el siguiente enlace ([https://code.visualstudio.com/docs/editor/versioncontrol#\\_git-support](https://code.visualstudio.com/docs/editor/versioncontrol#_git-support)) se puede encontrar ayuda sobre su uso.

## APÉNDICE B.- LAS IMÁGENES

En la carpeta “Documents” está disponible un programa de prueba denominado `TestImages`. El programa se invoca desde la línea de comandos y hay que pasarle dos imágenes como parámetros. El programa muestra simultáneamente ambas imágenes junto con su diferencia de tal manera que, cuando se pasan dos imágenes iguales, el resultado será una imagen negra. También muestra unas pequeñas estadísticas de la imagen resultado. Cuanto más próximos a cero sean los resultados, más semejantes serán las imágenes de entrada. Se puede utilizar para comprobar que las distintas implementaciones del algoritmo generan los mismos resultados.

En la carpeta “Pictures” del directorio de trabajo se encuentra un conjunto de imágenes disponibles para la realización de los algoritmos propuestos. Las distintas carpetas contienen los siguientes tipos de imágenes:

**Normal.-** Esta carpeta contiene imágenes tomadas con valores correctos de exposición.

**Contrast.-** En esta carpeta se encuentran imágenes tomadas con valores bajos de exposición (-2) o altos (+2). Son imágenes aptas para los algoritmos de alteración mejora del contraste/exposición.

**WhiteBalance.-** Esta carpeta contiene imágenes tomadas con alteraciones en su balance de blancos. Son imágenes adecuadas para utilizar con los algoritmos de corrección del balance de blancos.

**Backgrounds.-** Esta carpeta contiene unos fondos degradados del mismo tamaño que las imágenes anteriores y que se pueden usar como segunda imagen en los modos de fusión.

