Diseño Basado en Microprocesadores
Práctica 6
SIMD mediante funciones intrínsecas

Índice

---

1. Objetivos

En esta práctica se usarán funciones intrínsecas para programar ejemplos que aprovechen las instrucciones SIMD.

Puedes consultar los detalles del funcionamiento de estas funciones en `https://software.intel.com/sites/landingpage/IntrinsicsGuide`. También es útil la página `https://www.felixcloutier.com/x86/index.html` que incluye diagramas de funcionamiento de algunas instrucciones.

2. Ajustes del proyecto para usar funciones SIMD intrínsecas

Para usar las funciones SIMD intrínsecas será necesario realizar algunos ajustes en el proyecto de eclipse. **Es conveniente crear un proyecto de eclipse específico para esta práctica en lugar de aprovechar uno ya existente**. El proyecto será de 64 bits, así que no es necesario añadir los flags `-m32` que usamos al crear proyectos de 32 bits. En esta práctica no usaremos el ensamblador, así que no es necesario cambiar el nombre del ensamblador ni ajustar sus opciones.

2.1. Selección de la configuración *Release*

Para que el compilador traduzca de forma eficiente las funciones SIMD intrínsecas es necesario activar un nivel de optimización al menos **`-O1`**. Para conseguirlo, podemos seleccionar la configuración *Release* del proyecto (en lugar de la *Debug* que hemos estado usando), que ajustará el nivel de optimización **`-O3`**. Para ello, una vez que hayas creado el proyecto para la práctica, selecciona *Proyect* →*Build Configurations* →*Set Active* →*Release* en el menú principal.

2.2. Flags para el compilador

Para que el compilador GCC pueda generar instrucciones SSE/AVX/AVX2 añadiremos también al compilador el flag **`-mavx2`**. Para ello, en el apartado de opciones *Miscellaneous* del compilador, añade **`-mavx2`** a los flags que ya existan (no quites las que ya estén).

2.3. Depuración de la versión *Release*

El inconveniente de seleccionar la configuración *Release* es que ésta desactiva por defecto la incorporación de información de depuración en el ejecutable. Aunque se añada información de depuración, usar el depurador con un programa que ha sido compilado con alguna opción de optimización puede ser algo difícil y confuso, ya que las transformaciones que la optimización aplica pueden hacer que el flujo del código máquina generado se aleje del correspondiente al fichero fuente original. En ocasiones, esto puede hacer que la ejecución paso a paso en el depurador parezca no seguir un orden lógico.

Por este motivo, si necesitas depurar el programa puede ser conveniente elegir la configuración *Debug* seleccionando *Proyect* →*Build Configurations* →*Set Active* →*Debug* en el menú principal. Debes añadir también el flag **`-mavx2`** para el compilador en el apartado *Miscellaneous* de la configuración *Debug*. A continuación, vuelve a compilar.

Al ejecutar o depurar, asegúrate de que el ejecutable que se usa es el correspondiente a la configuración *Debug*.

Una vez que el programa funcione correctamente, vuelve a seleccionar la configuración *Release*.

3. Fichero de cabecera para funciones SIMD intrínsecas

Para usar funciones intrínsecas, incluye el fichero de cabecera **`immintrin.h`** en los ficheros fuente `.c` o `.cpp` desde los que las llames:

```
#include <immintrin.h>
```

4. Ejercicios

4.1. Ejercicio 1

Crea una función en C que emplee funciones AVX intrínsecas para sumar dos matrices 4×4 formadas por elementos de tipo `double`. El prototipo de la función es:

```
int fi_sumar_matrices_4x4_double(const double *ptr_matriz_1,
                                 const double *ptr_matriz_2,
                                 double *ptr_matriz_suma)
```

donde

`ptr_matriz_1` es un puntero al primer elemento de una de las matrices a sumar.

`ptr_matriz_2` es un puntero al primer elemento de la otra matriz a sumar.

`ptr_matriz_suma` es un puntero a la posición de memoria a partir de la cual hay que almacenar la matriz suma.

Los tres punteros deben apuntar a direcciones divisibles entre 32 para poder acceder a las filas de las matrices mediante funciones AVX de carga/almacenamiento alineadas.

La función retornará 1 si puede realizar correctamente su trabajo y 0 si alguno de los punteros es nulo o no está alineado correctamente.

Para realizar este ejercicio son útiles las siguientes funciones instrínsecas SIMD:

- `__m256d _mm256_add_pd(__m256d a, __m256d b)`
- `__m256d _mm256_load_pd(double const * mem_addr)`
- `void _mm256_store_pd(double * mem_addr, __m256d a)`

La primera función se corresponde con la instrucción `VADDPD` mientras que las dos últimas se corresponden con la instrucción `VMOVAPD`.

No hay inconveniente en usar el valor retornado por una función como argumento de otra.

4.2. Ejercicio 2

Escribe una función que use funciones intrínsecas AVX/AVX2 para calcular la media aritmética de los valores contenidos en un array de floats. El prototipo de la función es:

```
int fi_media_floats(float *ptr_datos,
                    unsigned int longitud_array,
                    float *ptr_resultado)
```

donde

`array_datos` es un puntero al primero de los datos del array. Debe estar alineado a una dirección divisible entre 32.

`longitud_array` indica el número de floats en el array.

ptr_resultado es un puntero a la variable donde guardar la media.

La función devuelve 1 si pudo trabajar correctamente y 0 si hubo algún error en los valores de los argumentos.

Procesa tantos bloques de ocho floats como sea posible usando funciones intrínsecas AVX. Para realizar este ejercicio son útiles las siguientes funciones:

- `__m256 _mm256_set1_ps(float a)`
- `__m256 _mm256_add_ps(__m256 a, __m256 b)`
- `__m256 _mm256_hadd_ps(__m256 a, __m256 b)`
- `__m256 _mm256_permute2f128_ps(__m256 a, __m256 b, int imm8)`
- `__m128 _mm256_castps256_ps128(__m256 a)`
- `void _mm_store_ss(float* mem_addr, __m128 a)`

Para la parte final de la función, que incluirá la suma de los elementos que no se hayan podido agrupar en bloques de ocho, la división de la suma entre el número de elementos y el almacenamiento en la variable apuntada por `ptr_resultado` no uses funciones intrínsecas escalares, sino líneas de código en C normales.

4.3. Ejercicio 3

Crea una función en C que emplee funciones intrínsecas AVX/AVX2 para seleccionar de una imagen RGB sólo las componentes de color deseadas. El prototipo de la función es

```
int fi_seleccionar_rgb(unsigned char *ptr_imagen,
                       unsigned int ancho,
                       unsigned int alto,
                       unsigned char seleccion_rgb)
```

donde

ptr_imagen es un puntero a los datos de la imagen. Cada pixel de la imagen está representado por tres bytes consecutivos. El primero representa la intensidad de azul, el segundo la intensidad de verde y el tercero la intensidad de rojo. Se exige que este puntero esté alineado a una posición divisible entre 32 para permitir acceder a los datos de la imagen mediante funciones AVX de carga/almacenamiento alineadas. La modificación de la imagen se realizará in situ.

ancho es el ancho de la imagen en pixels.

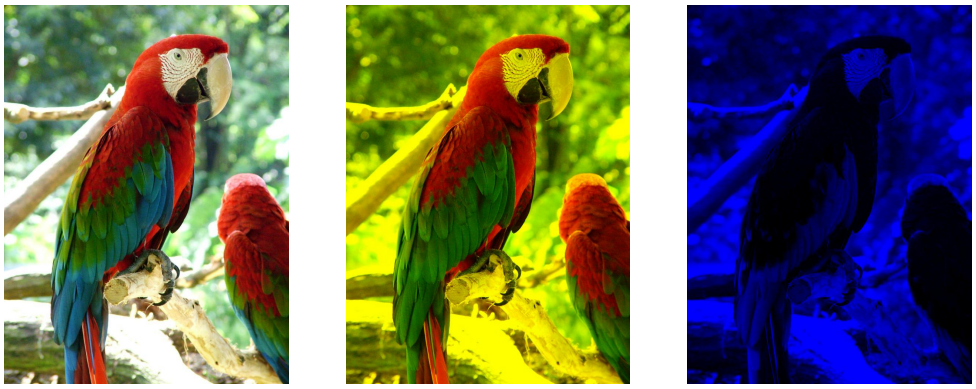
alto es el alto de la imagen en pixels.

seleccion_rgb indica las componentes de color a preservar y las componentes de color a eliminar:

- Si el bit 0 de `seleccion_rgb` es 1 se preservará la componente azul de la imagen. Si el bit 0 de `seleccion_rgb` es 0 se eliminará la componente azul de la imagen.
- Si el bit 1 de `seleccion_rgb` es 1 se preservará la componente verde de la imagen. Si el bit 1 de `seleccion_rgb` es 0 se eliminará la componente verde de la imagen.

- Si el bit 2 de `seleccion_rgb` es 1 se preservará la componente roja de la imagen. Si el bit 2 de `seleccion_rgb` es 0 se eliminará la componente roja de la imagen.

La función devuelve 1 si pudo trabajar correctamente y 0 si hubo algún error en los valores de los argumentos.



(a) (b) (c)

Figura 1: Imagen original (a) y resultado de seleccionar las componentes roja y verde, llamando a la función con `seleccion_rgb` = 6 (b), y de seleccionar solo la componente azul, llamando a la función con `seleccion_rgb` = 1 (c).

En la página de la asignatura hay disponibles ficheros fuente de partida y una imagen de prueba.

Las siguientes funciones intrínsecas SIMD son suficientes para hacer el ejercicio (pero puedes usar cualquier otra si lo estimas oportuno):

- `__m256i _mm256_setzero_si256(void)`
- `__m256i _mm256_or_si256 (__m256i a, __m256i b)`
- `__m256i _mm256_and_si256 (__m256i a, __m256i b)`
- `__m256i _mm256_load_si256 (__m256i const * mem_addr)`
- `void _mm256_store_si256 (__m256i * mem_addr, __m256i a)`

La primera función usa una instrucción `VPXOR` para generar un vector de 256 bits a cero. Las dos siguiente se corresponden, respectivamente, con las instrucciones `VPOR` y `VPAND`. La dos últimas usan la instrucción `VMOVDQA`.

La imagen en color que se va a procesar procederá de un fichero BMP con formato de 24 bits por pixel, que almacena la información de color de cada pixel mediante tres bytes consecutivos que representan la intensidad de luz **azul, verde y roja, en ese orden**. Por tanto, es mejor procesar la imagen en bloques de 96 bytes porque de esa forma un bloque con los datos de 32 pixels encaja perfectamente en tres registros YMM (de 256 = 32×8 bits cada uno).

Para conservar las componentes de color seleccionadas y eliminar el resto podemos aplicar a cada bloque de 32 pixels (96 bytes) una operación AND con la máscara adecuada. A su vez, esta máscara se puede obtener combinando mediante la operación OR las máscaras que permiten seleccionar las componentes azul, verde y roja, según indique el argumento

`seleccion_rgb`. Por ejemplo, para seleccionar la componente de azul de los 32 pixels, sería necesaria la siguiente máscara de 96 bytes:

```
{0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,
    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,
    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0,
 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,
    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,
    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0}
```

Como las instrucciones y funciones intrínsecas AVX/AVX2 nos permitirán manejar a la vez un máximo de 32 bytes, está máscara tendrá que ser separada en tres partes:

Máscara azul 1:

```
{0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,
    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0}
```

Máscara azul 2:

```
{   0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0,
 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF}
```

Máscara azul 3:

```
{   0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,
    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0, 0xFF,    0,    0}
```

Las máscaras para seleccionar las otras dos componentes se obtienen colocando el valor 0xFF en las posiciones correspondientes y rellenando con ceros las demás.

Los pixels que no puedan agruparse en bloques de 32 se procesarán de uno en uno.