# Introducción a las Instrucciones SIMD/Vectoriales

Diego Andrade Manuel Arenaz

Grupo de Arquitectura de Computadores
Universidade da Coruña

22 y 23 de Enero del 2009

1 Introducción

2 Las instrucciones SIMD/vectoriales

3 Guía de empleo de *SSE Intrinsics*

4 Un ejemplo completo

5 Prácticas a realizar

# Taxonomía de Flynn (1966)

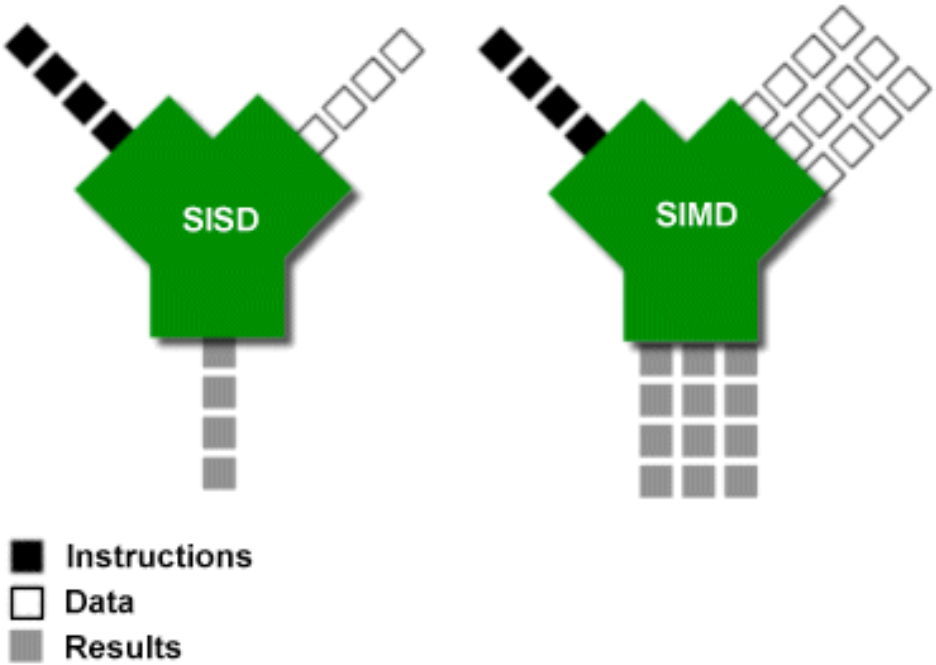
Basada en el flujo se instrucciones y datos

- SISD: Single Instruction Single Data
  - Uniprocesadores
- MISD: Multiple Instruction Single Data
  - Múltiples procesadores sobre un único flujo de datos
- SIMD: Single Instruction Multiple Data
  - Modelo de programación sencillo (SPMD)
  - Ejecución síncrona de la misma instrucción sobre datos diferentes
- MIMD: Multiple Instruction Multiple Data
  - Ejecución asíncrona de múltiples segmentos de código diferentes sobre datos diferentes

# Procesamiento en modo SIMD

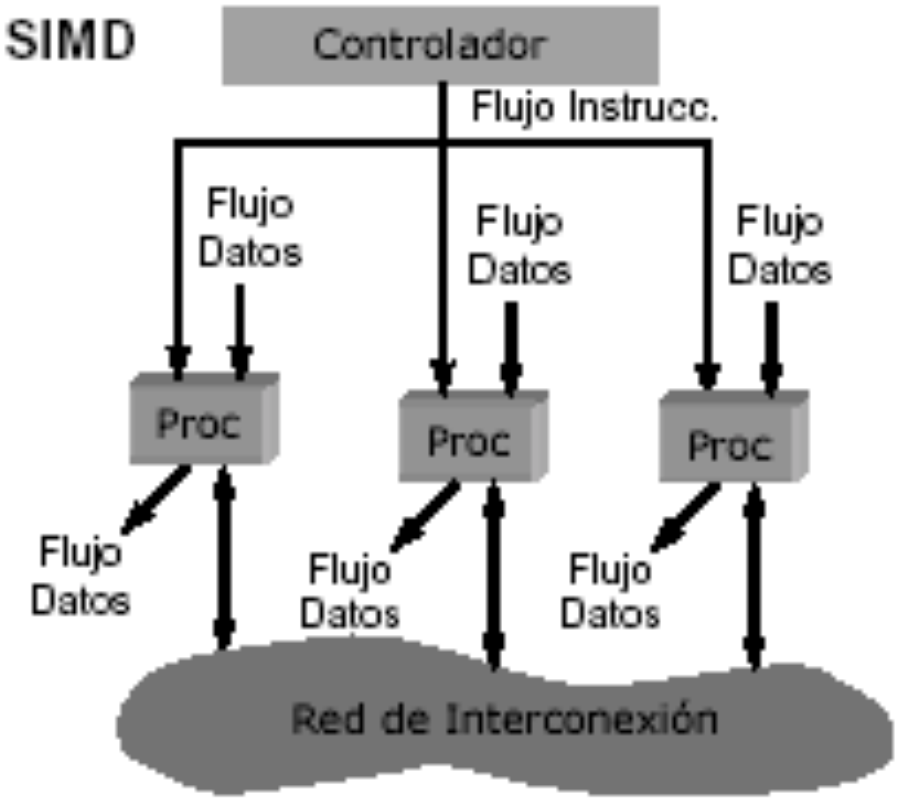
- Extensiones multimedia de los repertorios de instrucciones de los microprocesadores actuales (instrucciones SIMD/vectoriales): SSE de Intel, Altivec para el PowerPC (IBM), 3DNow para AMD.
- GPUs (Graphic Processing Unit): tarjetas gráficas como GeForce 8 Series, Quadro FX 5600/4600
- Cell processor: es el procesador de la Playstation3. Dentro de cada unidad unidad SPE (tiene 8 en el mismo procesador) utiliza instrucciones vectoriales.
  - Nota: El conjunto de 8 SPEs + la unidad principal PPE funcionan en modo asíncrono (MIMD).
- Un sistema MIMD puede implementar una ejecución paralela en modo SIMD (SPMD).

# Concepto SIMD




Concepto abstracto: SISD vs. SIMD

# Concepto SIMD




Organización SIMD

# Evolución histórica de las instrucciones SIMD

- Año 1997: MMX
- Año 1998: 3dNow
- Año 1999: SSE
- Año 2001: SSE2
- Año 2004: SSE3
- Año 2006: SSSE3
- Año 2007: SSE4
- Año 2007: SSE5
- Año 2008: AVX

- Aparecen en el Pentium.
- Usan 8 registros vectoriales (mm0 . . . mm7) de 64 bits. Registros compartidos con la FPU.
- Solo operaciones sobre enteros.

# Evolución histórica de las instrucciones SIMD

- Año 1997: MMX
- Año 1998: 3dNow
- Año 1999: SSE
- Año 2001: SSE2
- Año 2004: SSE3
- Año 2006: SSSE3
- Año 2007: SSE4
- Año 2007: SSE5
- Año 2008: AVX

- Aparecen en el AMD K6-2.
- Operan tanto sobre enteros como sobre punto flotante.
- Extendida posteriormente en *Enhaced 3dNow* y *3dNow Professional*

# Evolución histórica de las instrucciones SIMD

- Año 1997: MMX
- Año 1998: 3dNow
- Año 1999: SSE
- Año 2001: SSE2
- Año 2004: SSE3
- Año 2006: SSSE3
- Año 2007: SSE4
- Año 2007: SSE5
- Año 2008: AVX

- Acrónimo de *Streaming SIMD Extensions.*
- *Usadas en el Pentium III y el AMD Athlon.*
- *8 nuevos registros (xmm0 . . . xmm7) de 128 bits.*
- *70 nuevas instrucciones vectoriales en punto flotante.*

# Evolución histórica de las instrucciones SIMD

- Año 1997: MMX
- Año 1998: 3dNow
- Año 1999: SSE
- Año 2001: SSE2
- Año 2004: SSE3
- Año 2006: SSSE3
- Año 2007: SSE4
- Año 2007: SSE5
- Año 2008: AVX

- Presentes en el Pentium IV y el AMD Athlon 64.
- Nuevas instrucciones vectoriales para enteros.
- Ya no es necesario el uso de las instrucciones MMX.

# Evolución histórica de las instrucciones SIMD

- Año 1997: MMX
- Año 1998: 3dNow
- Año 1999: SSE
- Año 2001: SSE2
- Año 2004: SSE3
- Año 2006: SSSE3
- Año 2007: SSE4
- Año 2007: SSE5
- Año 2008: AVX

- Se usan en el Pentium IV Prescott y en los últimos Athlon 64 y Athlon 64 X2.
- 13 nuevas instrucciones.
- Mejora la capacidad para trabajar (horizontalmente) sobre datos guardados en el mismo registro.

# Evolución histórica de las instrucciones SIMD

- Año 1997: MMX
- Año 1998: 3dNow
- Año 1999: SSE
- Año 2001: SSE2
- Año 2004: SSE3
- Año 2006: SSSE3
- Año 2007: SSE4
- Año 2007: SSE5
- Año 2008: AVX

- Acrónimo de *Suplementary SSE3.*
- *Usadas solo en el Intel Core 2 Duo.*
- *Son 16 nuevas instrucciones.*

# Evolución histórica de las instrucciones SIMD

- Año 1997: MMX
- Año 1998: 3dNow
- Año 1999: SSE
- Año 2001: SSE2
- Año 2004: SSE3
- Año 2006: SSSE3
- Año 2007: SSE4
- Año 2007: SSE5
- Año 2008: AVX

- Un parte serán implementadas en el Intel Core 2 Duo Penryn.
- Son 54 nuevas instrucciones.

# Evolución histórica de las instrucciones SIMD

- Año 1997: MMX
- Año 1998: 3dNow
- Año 1999: SSE
- Año 2001: SSE2
- Año 2004: SSE3
- Año 2006: SSSE3
- Año 2007: SSE4
- Año 2007: SSE5
- Año 2008: AVX

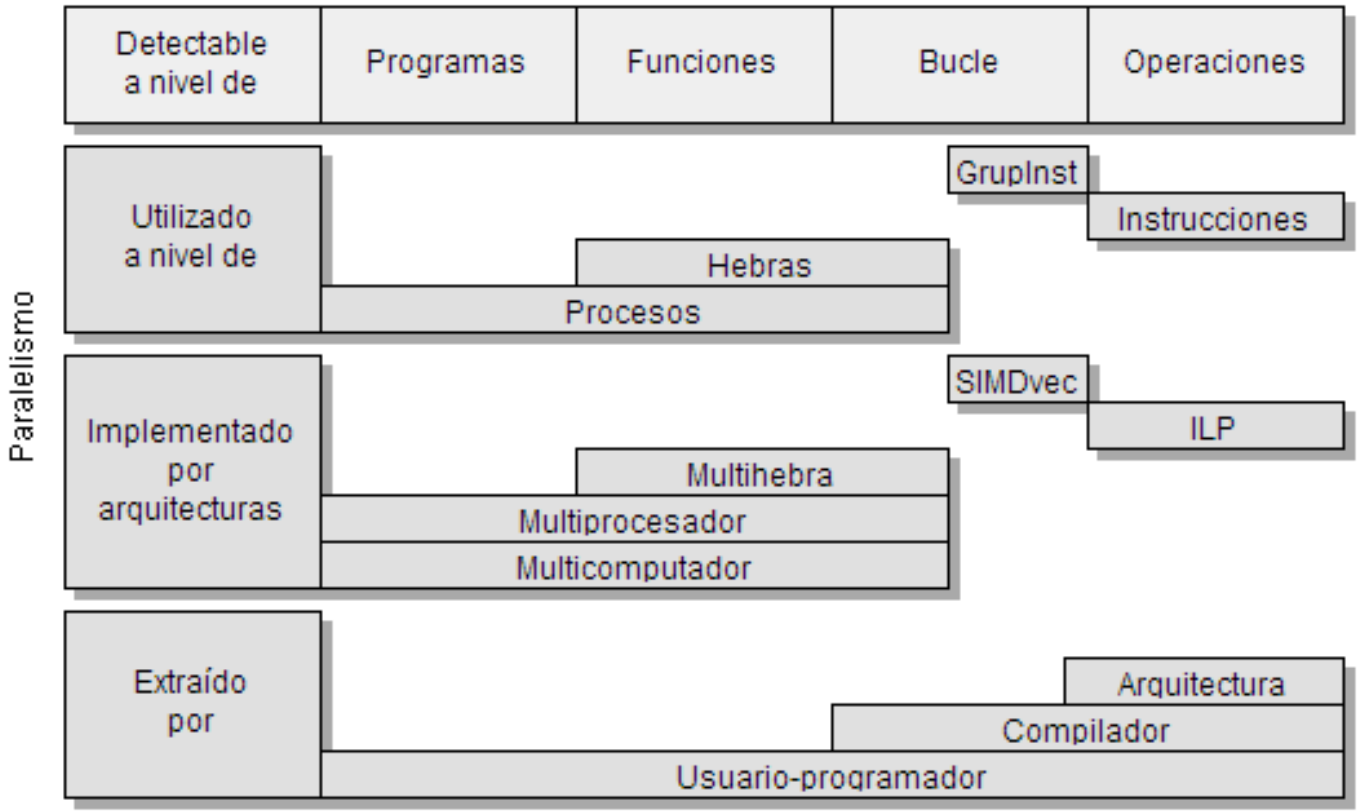
- Implementadas en el procesador AMD Bulldozer (2011)
- Son 170 nuevas instrucciones.

# Evolución histórica de las instrucciones SIMD

- Año 1997: MMX
- Año 1998: 3dNow
- Año 1999: SSE
- Año 2001: SSE2
- Año 2004: SSE3
- Año 2006: SSSE3
- Año 2007: SSE4
- Año 2007: SSE5
- Año 2008: AVX

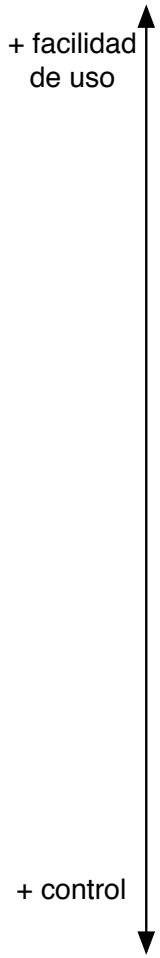
- Implementadas en el procesador Intel Sand Bridge (2010)

# Utilización de la tecnología SIMD




Contextualización del paralelismo SIMD

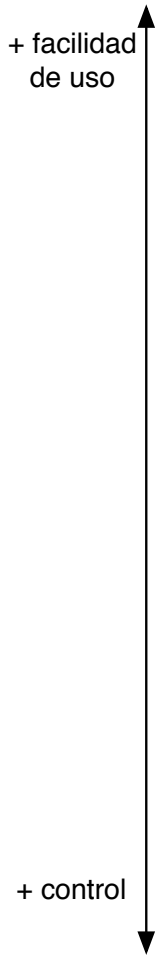
# Métodos para usar la tecnología SIMD




- Vectorización automática.
- Clases C++ especiales.
- Librería *Intrinsics*.
- *Lenguaje ensamblador embebido.*

- En gcc (version >=4.2)
  - -O2 -ftree-vectorize -ftree-vectorizer-verbose=*nivel* -*march*=arquitectura
- En icc
  - -xW -O2 -vec-report*nivel*

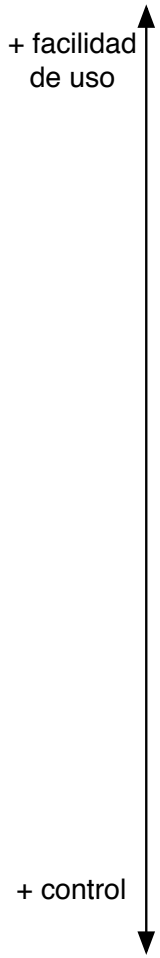
# Métodos para usar la tecnología SIMD




- Vectorización automática.
- Clases C++ especiales.
- Librería *Intrinsics*.
- *Lenguaje ensamblador embebido.*

- Cabeceras.
  - *ivec.h para MMX*
  - fvec.h para SSE e incluye *ivec.h*
  - dvec.h para SSE2 e incluye *fvec.h*
- Clases especiales.
  - *I8vec8, (ocho enteros de 8 bits)* I8vec16, (dieciseis enteros de 8 bits) *I16vec4, I16vec8, I32vec2, I32vec4, I64vec1, I64vec2, I128vec1*
  - *Isec8, (entero con signo)* Iuvec8 (entero sin signo)
  - *F32vec4 cuatro números flotante en precisión simple de 32 bits*
  - *F64vec4 cuatro números flotante en precisión simple de 32 bits*

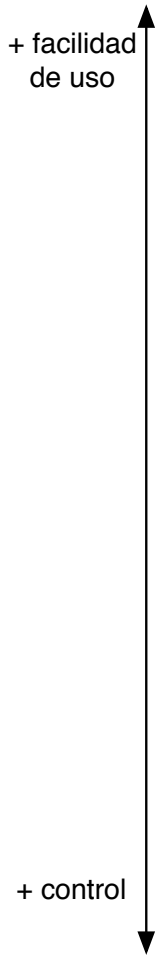
# Métodos para usar la tecnología SIMD




- Vectorización automática.
- Clases C++ especiales.
- Librería *Intrinsics.*
- *Lenguaje ensamblador embebido.*

- Cabeceras.
  - *mmintrin.h para MMX*
  - xmmintrin.h para SSE e incluye *mmintrin.h*
  - emmintrin.h para SSE2 e incluye *xmmintrin.h*
  - pmmintrin.h para SSE3 e incluye *emmintrin.h*
- Tipos de datos especiales.
  - *__m128,__m128d,__m128i*
- Instrucciones especiales.
  - *_mm_loadu_si128(), _mm_storeu_si128(), . . .*

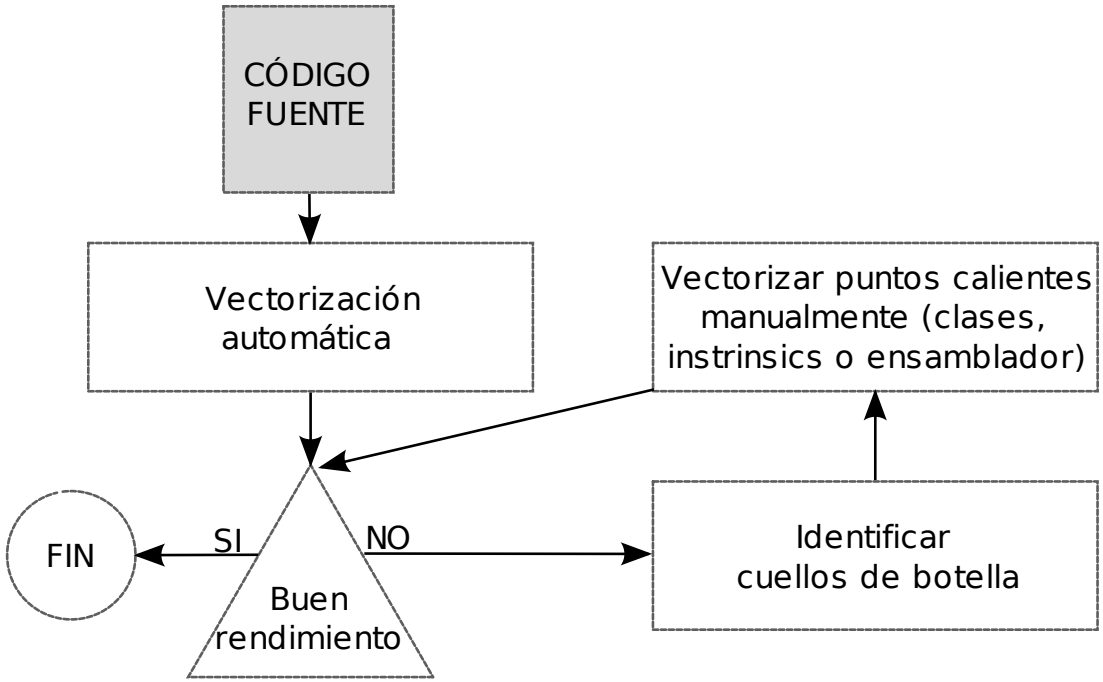
# Métodos para usar la tecnología SIMD




- Vectorización automática.
- Clases C++ especiales.
- Librería *Intrinsics*.
- *Lenguaje ensamblador embebido.*

```
_asm {
// Instrucciones en ensamblador
}
```

# Optimización usando instrucciones SSE




Flujo de trabajo para extraer paralelismo SIMD

# Usando SSE Intrinsics

Niveles de almacenamiento de los datos:

- Organización lógica de las estructuras de datos
- Organización física en RAM
- Organización física en los registros vectoriales

# Usando SSE intrinsics

Dos elecciones clave:

- Instrucciones vectoriales a utilizar
- Mapeado de los datos físicos en RAM a registros vectoriales: Facilitar la operación usando instrucciones vectoriales.

Factores de implementación a considerar:

- Alineado de memoria.
- Distribución de los datos.
- Tamaño de los datos.

Ambas elecciones están relacionadas

# Usando SSE intrinsics

Dos elecciones clave:

- Instrucciones vectoriales a utilizar
- Mapeado de los datos físicos en RAM a registros vectoriales: Facilitar la operación usando instrucciones vectoriales.

  Factores de implementación a considerar:
  - Alineado de memoria.
  - Distribución de los datos.
  - Tamaño de los datos.

Ambas elecciones están relacionadas

# Mapeo de datos: Alineado de memoria

- Los registros SSE tienen 128 bits (16 bytes).
- Las instrucciones vectoriales deben operar sobre datos alineados.
- Si operamos vectorialmente datos no alineados obtenemos una excepción. Existen instrucciones específicas para operar datos no alineados (- rendimiento).

# Mapeado de datos: Alineado de memoria

- Memoria estática:
  - Alineamiento:

    ```
    _declspec(align(base,off)) float a[100];
    ```

    $a \ mod \ base = off$

- Memoria dinámica:
  - Modificación manual del puntero.
  - Instrucción específica de reserva de memoria:

    ```
    int *pBuf=(int *) _mm_malloc(128*sizeof(int),16);
    ...
    _mm_free(pBuf);
    ```

# Mapeado de datos: Distribución de los datos

Caso de estudio (problema):

- Producto de 2 vectores de 4 elementos.
- Se compone de 2 operaciones:
  - Producto
  - y Suma
- La distribución habitual de los datos dificulta la suma.

# Mapeado de datos: Distribución de los datos

$$\begin{bmatrix} x[0] & x[1] & x[2] & x[3] \end{bmatrix} \times \begin{bmatrix} y[0] \\ y[1] \\ y[2] \\ y[3] \end{bmatrix} = x[0]y[0] + x[1]y[1] + x[2]y[2] + x[3]y[3]$$

Operación a realizar: producto vector X vector

# Mapeado de datos: Distribución de los datos

$$\begin{bmatrix} x[0] & x[1] & x[2] & x[3] \end{bmatrix} \times \begin{bmatrix} y[0] \\ y[1] \\ y[2] \\ y[3] \end{bmatrix} = x[0]y[0] + x[1]y[1] + x[2]y[2] + x[3]y[3]$$

**Organización en RAM**

| x[0] |
|---|
| x[1] |
| x[2] |
| x[3] |
| y[0] |
| y[0] |
| y[2] |
| y[3] |

Ubicación de los datos en memoria

# Mapeado de datos: Distribución de los datos

$$\begin{bmatrix} x[0] & x[1] & x[2] & x[3] \end{bmatrix} \times \begin{bmatrix} y[0] \\ y[1] \\ y[2] \\ y[3] \end{bmatrix} = x[0]y[0] + x[1]y[1] + x[2]y[2] + x[3]y[3]$$

**Organización en RAM**

| x[0] |
|---|
| x[1] |
| x[2] |
| x[3] |
| y[0] |
| y[0] |
| y[2] |
| y[3] |

**Organización en registros vectoriales**

| x[0] | x[1] | x[2] | x[3] | Registro xmm0 |
|---|---|---|---|---|

| y[0] | y[1] | y[2] | y[3] | Registro xmm1 |
|---|---|---|---|---|

Ubicación de los datos en registros vectoriales

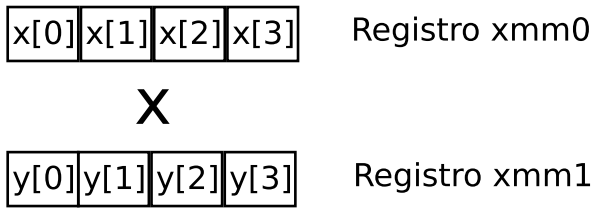
# Mapeado de datos: Distribución de los datos

$$\begin{bmatrix} x[0] & x[1] & x[2] & x[3] \end{bmatrix} \times \begin{bmatrix} y[0] \\ y[1] \\ y[2] \\ y[3] \end{bmatrix} = x[0]y[0] + x[1]y[1] + x[2]y[2] + x[3]y[3]$$

**Organización en RAM**

| |
|---|
| x[0] |
| x[1] |
| x[2] |
| x[3] |
| y[0] |
| y[0] |
| y[2] |
| y[3] |

**Organización en registros vectoriales**




Producto

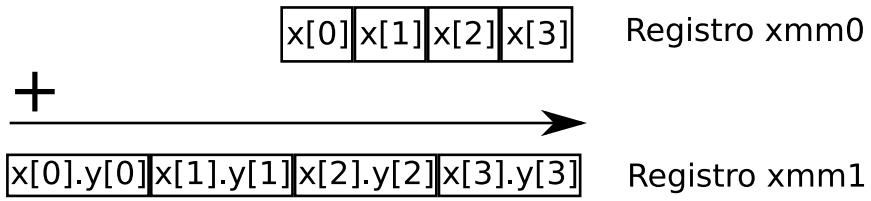
# Mapeado de datos: Distribución de los datos

$$\begin{bmatrix} x[0] & x[1] & x[2] & x[3] \end{bmatrix} \times \begin{bmatrix} y[0] \\ y[1] \\ y[2] \\ y[3] \end{bmatrix} = x[0]y[0] + x[1]y[1] + x[2]y[2] + x[3]y[3]$$

**Organización en RAM**

| x[0] |
|---|
| x[1] |
| x[2] |
| x[3] |
| y[0] |
| y[0] |
| y[2] |
| y[3] |

**Organización en registros vectoriales**




Suma. Problema: Es costoso sumar elementos ubicados en un mismo registro.

# Mapeado de datos: Distribución de los datos

Caso de estudio (solución):

- Si tenemos que hacer varios productos de vectores podemos agruparlos. (En este caso de 4 en 4).
- Nueva distribución de los datos: Entrelazamos elementos de diferentes vectores.
- Tanto la suma como los productos se producen entre datos almacenados en registros diferentes.

# Mapeado de datos: Distribución de los datos

AW, BX, CY y DZ

Hago 4 productos vector X vector simultáneamente

# Mapeado de datos: Distribución de los datos

AW, BX, CY y DZ

**Organización en RAM**

| |
|---|
| a[0] |
| b[0] |
| c[0] |
| d[0] |
| w[0] |
| x[0] |
| y[0] |
| z[0] |
| a[1] |
| b[1] |
| c[1] |
| .... |

Los elementos de los 4 vectors se encuentran entrelazados en memória

# Mapeado de datos: Distribución de los datos

## AW, BX, CY y DZ

**Organización en RAM**

| |
|---|
| a[0] |
| b[0] |
| c[0] |
| d[0] |
| w[0] |
| x[0] |
| y[0] |
| z[0] |
| a[1] |
| b[1] |
| c[1] |
| .... |

**Organización en registros vectoriales**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| a[0] | b[0] | c[0] | d[0] | x | w[0] | x[0] | y[0] | z[0] |
| a[1] | b[1] | c[1] | d[1] | x | w[1] | x[1] | y[1] | z[1] |
| a[2] | b[2] | c[2] | d[2] | x | w[2] | x[2] | y[2] | z[2] |
| a[3] | b[3] | c[3] | d[3] | x | w[3] | x[3] | y[3] | z[3] |

## Producto

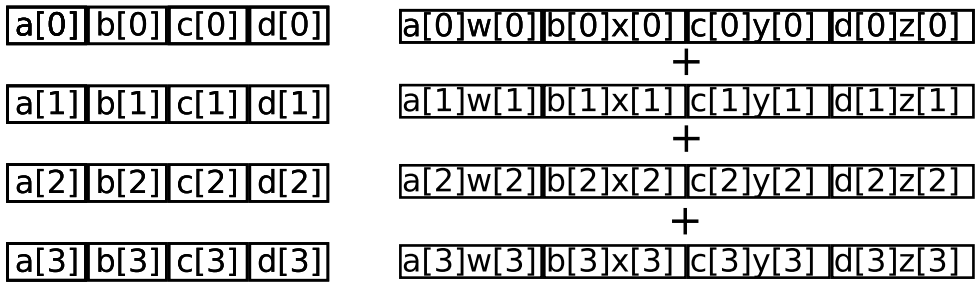
# Mapeado de datos: Distribución de los datos

## AW, BX, CY y DZ

**Organización en RAM**

| |
|---|
| a[0] |
| b[0] |
| c[0] |
| d[0] |
| w[0] |
| x[0] |
| y[0] |
| z[0] |
| a[1] |
| b[1] |
| c[1] |
| .... |

**Organización en registros vectoriales**




Suma. Ahora es más sencillos al operar sobre datos de diferentes registros.

# Mapeado de datos: Tamaño de los datos

- La cantidad de paralelismo alcanzable utilizando instrucciones SIMD depende directamente del tamaño de los elementos individuales.
- Es necesario usar el tipo de datos más pequeño que podamos.

**Ejemplo**

Usar unsigned short en vez de *unsigned int si sabemos que el máximo entero almacenado nunca excederá del rango de un* unsigned short.

# Programación en SSE usando intrinsics

Cabeceras a utilizar:

- *mmintrin.h para instrucciones MMX*
- xmmintrin.h para instrucciones SSE. Incluye *mmintrin.h*
- emmintrin.h para instrucciones SSE2. Incluye *xmmintrin.h*
- pmmintrin.h para instrucciones SSE3. Incluye *emmintrin.h*

Proporcionan:

- Tipos de datos para manejo de los registros vectoriales
- Instrucciones vectoriales

En Linux la llamada *cat /proc/cpuinfo me dice qué extensiones están disponibles en mi máquina*

# Tipos de datos de intrinsics

| Tipo | Registro | Almacena |
|---|---|---|
| *__m64* | MMX | 8x8 bits, 4x16 bits, 2x32 bits o 1x64 bits |
| *__m128* | XMM | 4x32 bits |
| *__m128d* | XMM | 2x64 bits |
| *__m128i* | XMM | 16x8 bits, 8x16 bits, 4x32 bits, 2x64 bits |

# Instrucciones vectoriales de intrinsics

El nombre suele seguir la forma *_libreria_instruccion_sufijo*

---

- *Librería: Para SSE se usa siempre* mm. Para *Altivec se usa* vec.
- Instrucción: Por ejemplo la suma es *add.*
- *Sufijo:*
  - *Sobre registros MMX(64-bit):*
    - -pi# Contienen enteros de #-bit empaquetados.
    - -pu# Contienen enteros sin signo de #-bit empaquetados.
    - -si64 Contienen un solo entero de 64 bits.
  - *Sobre registros XMM(128-bit):*
    - -epi# Contienen enteros de #-bit empaquetados.
    - -epu# Contienen enteros sin signo de #-bit empaquetados.
    - -ps Contienen valores en punto flotante de precisión simple empaquetados.
    - -ss Contienen un número en punto flotante de precisión simple.
    - -pd Contienen valores en punto flotante de doble precisión empaquetados.
    - -sd Contienen un número en punto flotante de doble precisión.
    - -si128 Contienen un solo entero de 128 bits.

# Instrucciones vectoriales de intrinsics

El nombre suele seguir la forma *_libreria_instruccion_sufijo*

---

- *Librería: Para SSE se usa siempre* mm. Para *Altivec se usa* vec.
- Instrucción: Por ejemplo la suma es *add.*
- *Sufijo:*
  - *Sobre registros MMX(64-bit):*
    - -pi# Contienen enteros de #-bit empaquetados.
    - -pu# Contienen enteros sin signo de #-bit empaquetados.
    - -si64 Contienen un solo entero de 64 bits.
  - *Sobre registros XMM(128-bit):*
    - -epi# Contienen enteros de #-bit empaquetados.
    - -epu# Contienen enteros sin signo de #-bit empaquetados.
    - -ps Contienen valores en punto flotante de precisión simple empaquetados.
    - -ss Contienen un número en punto flotante de precisión simple.
    - -pd Contienen valores en punto flotante de doble precisión empaquetados.
    - -sd Contienen un número en punto flotante de doble precisión.
    - -si128 Contienen un solo entero de 128 bits.

# Instrucciones vectoriales de intrinsics

El nombre suele seguir la forma *_libreria_instruccion_sufijo*

---

- *Librería: Para SSE se usa siempre* mm. Para *Altivec se usa* vec.
- Instrucción: Por ejemplo la suma es *add.*
- *Sufijo:*
  - *Sobre registros MMX(64-bit):*
    - -pi# Contienen enteros de #-bit empaquetados.
    - -pu# Contienen enteros sin signo de #-bit empaquetados.
    - -si64 Contienen un solo entero de 64 bits.
  - *Sobre registros XMM(128-bit):*
    - -epi# Contienen enteros de #-bit empaquetados.
    - -epu# Contienen enteros sin signo de #-bit empaquetados.
    - -ps Contienen valores en punto flotante de precisión simple empaquetados.
    - -ss Contienen un número en punto flotante de precisión simple.
    - -pd Contienen valores en punto flotante de doble precisión empaquetados.
    - -sd Contienen un número en punto flotante de doble precisión.
    - -si128 Contienen un solo entero de 128 bits.

# Instrucciones vectoriales de intrinsics

El nombre suele seguir la forma *_libreria_instruccion_sufijo*

---

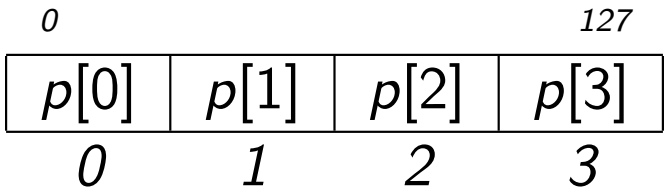
- *Librería: Para SSE se usa siempre* mm. Para *Altivec se usa* vec.
- Instrucción: Por ejemplo la suma es *add.*
- *Sufijo:*
  - *Sobre registros MMX(64-bit):*
    - -pi# Contienen enteros de #-bit empaquetados.
    - -pu# Contienen enteros sin signo de #-bit empaquetados.
    - -si64 Contienen un solo entero de 64 bits.
  - *Sobre registros XMM(128-bit):*
    - -epi# Contienen enteros de #-bit empaquetados.
    - -epu# Contienen enteros sin signo de #-bit empaquetados.
    - -ps Contienen valores en punto flotante de precisión simple empaquetados.
    - -ss Contienen un número en punto flotante de precisión simple.
    - -pd Contienen valores en punto flotante de doble precisión empaquetados.
    - -sd Contienen un número en punto flotante de doble precisión.
    - -si128 Contienen un solo entero de 128 bits.

# Tipos de instrucciones SSE

- Carga/Almacenamiento
- Inicialización
- Conversión
- Reordenamiento
- Comparación
- Aritméticas
- Lógicas

# Instrucciones: Operaciones de carga/almacenamiento

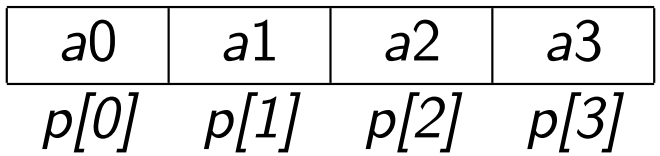
*__m128 _mm_load_ps(float *p)*




*Carga en un registro vectorial los 4 primeros flotantes en simple precisión almacenados a partir de la dirección señalada por* p. La dirección debe tener una alineamiento de 16 bytes.

# Instrucciones: Operaciones de carga/almacenamiento

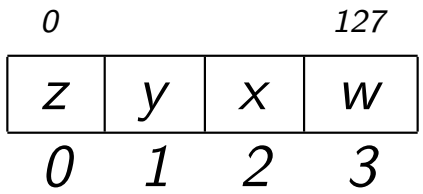
*void _mm_store_ps(float *p, __m128 a)*




*Almacena los 4 valores en punto flotante y simple precisin almacenados en el registro a en la dirección de memoria señalada por* p. La direccin debe tener una alineamiento de 16 bytes.

# Instrucciones: Operaciones de inicialización

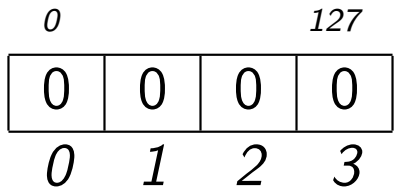
$$\_\_m128\ \_mm\_set\_ps(float\ z, float\ y, float\ x, float\ w)$$




*Inicializa el registro con los números en puntos flotante de precisión simple especificados en z,y,x y* w.

# Instrucciones: Operaciones de inicialización

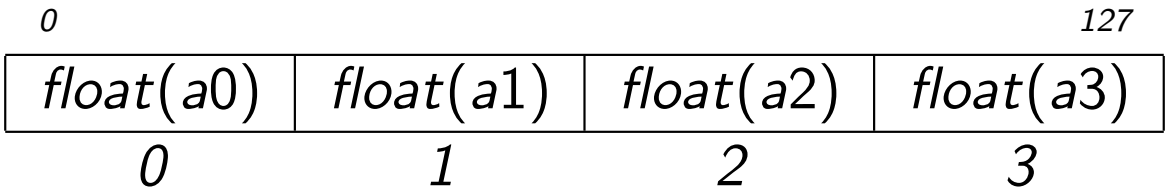
*__m128 _mm_setzero_ps(void)*




*Inicializa el registro con 4 ceros en punto flotante y simple precisión.*

# Instrucciones: Operaciones de conversión

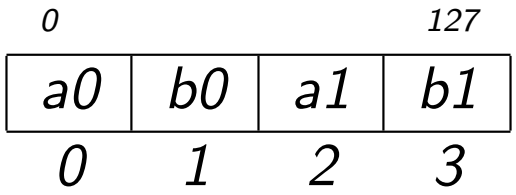
*__m128 _mm_cvtpi16_ps(__m64 a)*




*Convierte los 4 valores* signed integer de 16 bits almacenados en *a en 4 números en punto flotante con precisión simple. Existen conversiones disponibles entre otros tipos.*

# Instrucciones: Operaciones de reordenación

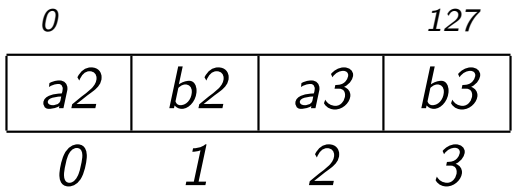
*__m128 _mm_unpacklo_ps(__m128 a, __m128 b)*




*Entrelaza los 2 valores en punto flotante y simple precisin de la parte inferior de* a con los 2 de la parte inferior de *b.*

# Instrucciones: Operaciones de reordenación

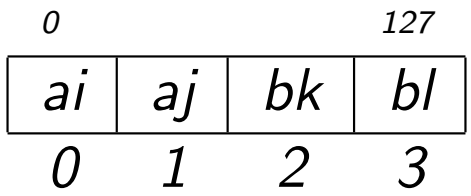
*__m128 _mm_unpackhi_ps(__m128 a, __m128 b)*




*Entrelaza los 2 valores en punto flotante de la parte superior de* a con los 2 de la parte superior de *b.*

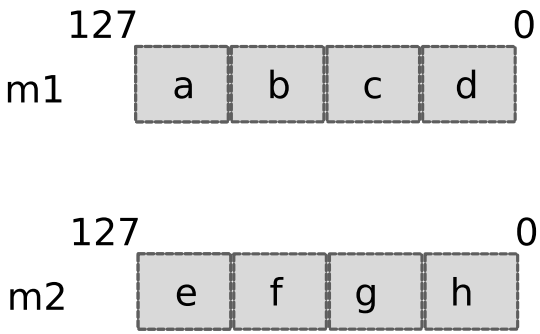
# Instrucciones: Operaciones de reordenación

*__m128 _mm_shuffle_ps(__m128 a, __m128 b, int imm)*



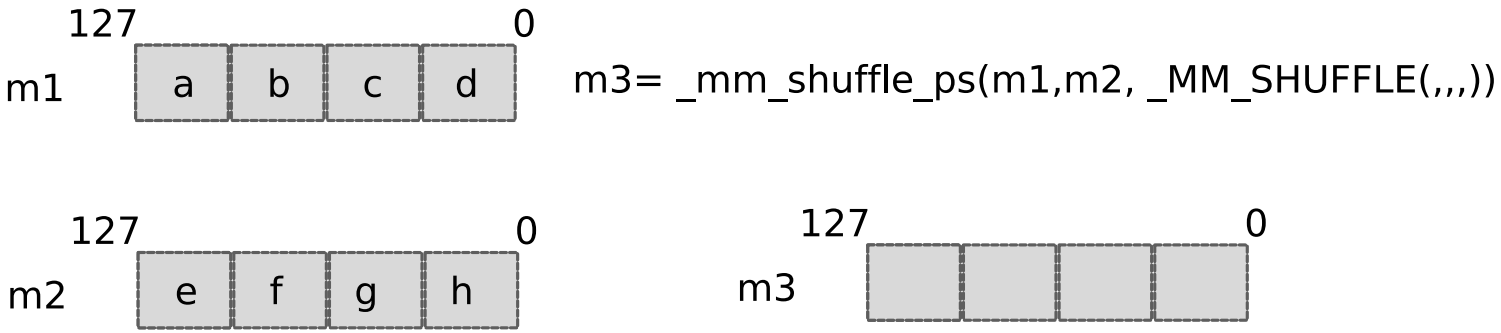

*Selecciona 4 valores de a y b basándose en el contenido de la máscara imm.*

# Operación de shuffle: Uso de la máscara



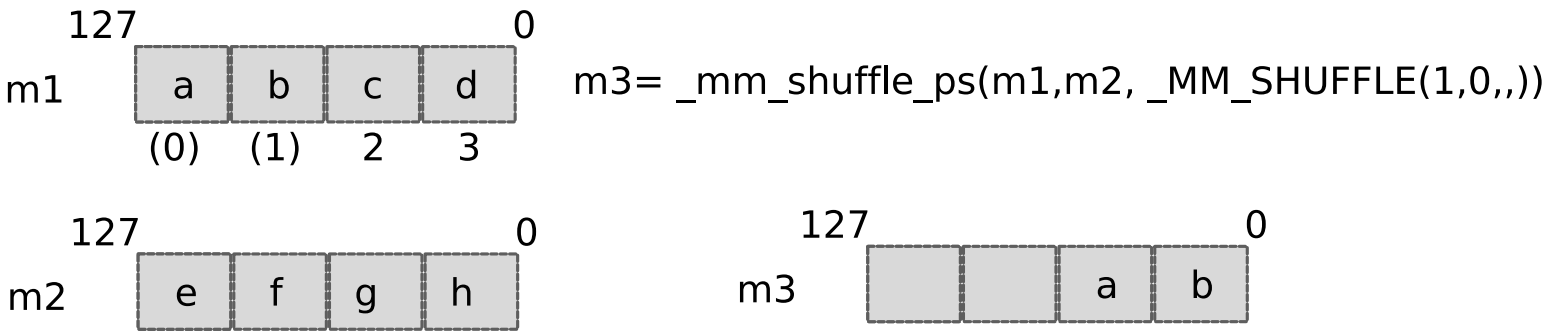

Dados dos registros vectoriales m1 y m2

# Operación de shuffle: Uso de la máscara



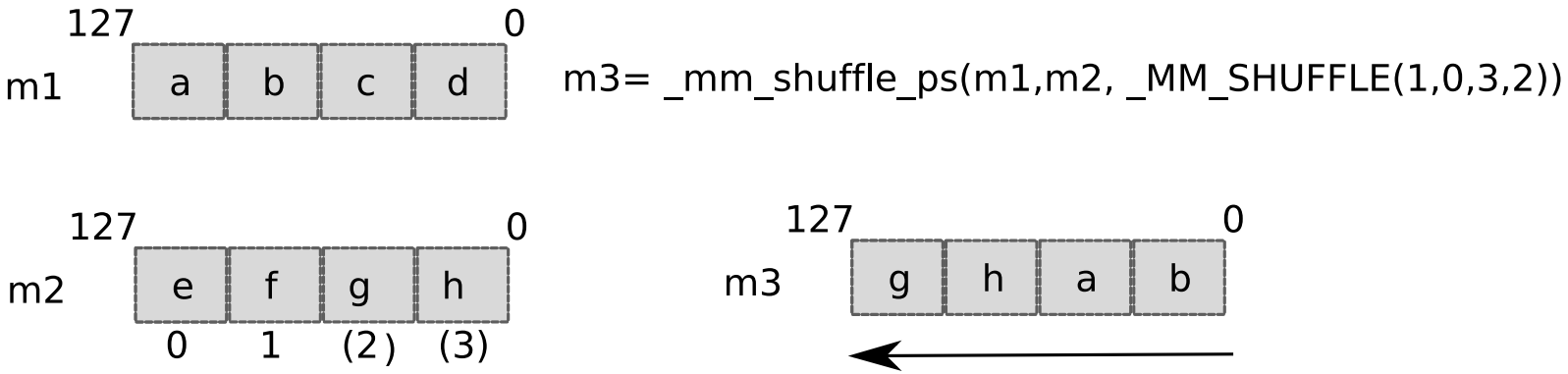

Los mezclo (shuffle) en un tercer registro m3, usando una máscara

# Operación de shuffle: Uso de la máscara




Los 2 primeros argumentos de la máscara seleccionan qué elementos de m1 serán almacenados en m3

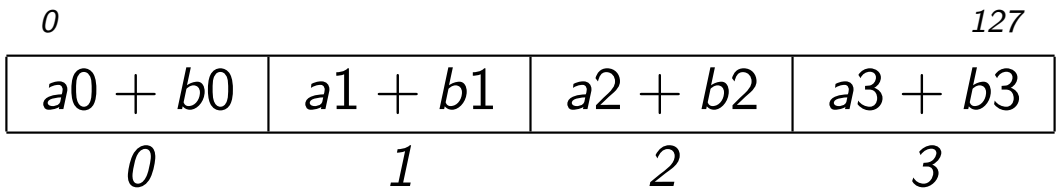
# Operación de shuffle: Uso de la máscara




Los 2 últimos argumentos de la máscara seleccionan qué elementos de m2 serán almacenados en m3

# Instrucciones: Operaciones aritméticas

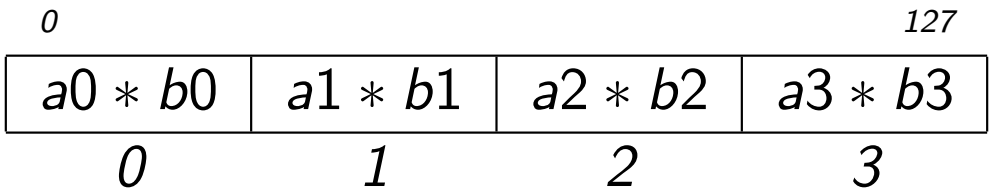
*__m128 _mm_add_ps(__m128 a, __m128 b)*




*Suma los 4 flotantes en simple precisión en* a y los 4 de *b.*

# Instrucciones: Operaciones aritméticas

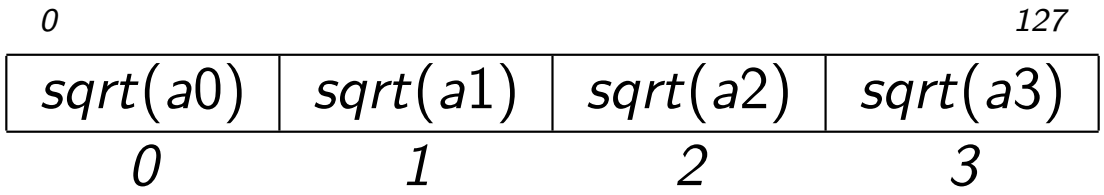
*__m128 _mm_mul_ps(__m128 a, __m128 b)*




*Multiplica los 4 flotantes en simple precisión en* a y los 4 de *b.*

# Instrucciones: Operaciones aritméticas

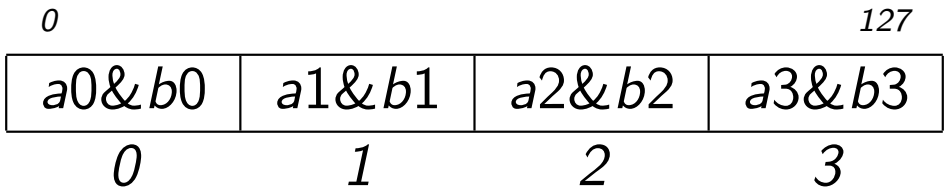
*__m128 _mm_sqrt_ps(__m128 a)*




*Calcula la raíz cuadrada de los 4 números en punto flotante y simple precisión almacenados en* a.

# Instrucciones: Operaciones lógicas

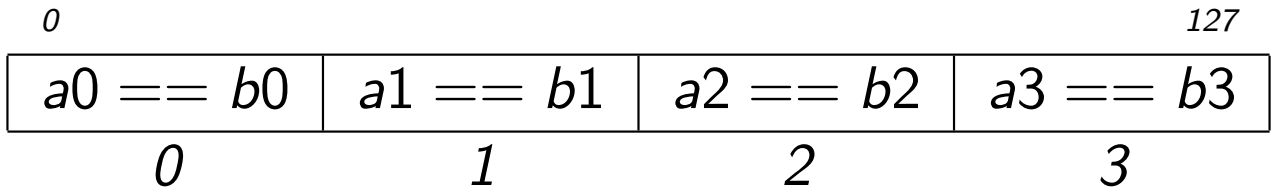
*__m128 _mm_and_ps(__m128 a, __m128 b)*




*Realiza un* and bit a bit entre los 4 valores en punto flotante almacenados en *a y los 4 almacenados en* b.

# Instrucciones: Operaciones de comparación

*__m128 _mm_cmpeq_ps(__m128 a, __m128 b)*




*Compara 2 a 2 los 4 valores en punto flotante y precisión simple almacenados en* a y los 4 almacenados en *b. Para cada comparación devuelve un valor de 32 bits.* 0x*ffffffff (true)* 0x0 *(false).*

# Sumar los elementos de dos vectores

```
...

float *a,*b,*c;
a=(float *) malloc(size*sizeof(float));
b=(float *) malloc(size*sizeof(float));
c=(float *) malloc(size*sizeof(float));

/*... Inicializacion de los array ...*/

for (i=0; i<size; i++) {
 a[i]=b[i]+c[i];
}

...
```

# Sumar los elementos de dos vectores

Alineamiento de datos

```
...

float *a,*b,*c;
a=(float *) _mm_malloc(size*sizeof(float),16);
b=(float *) _mm_malloc(size*sizeof(float),16);
c=(float *) _mm_malloc(size*sizeof(float),16);

...
```

# Sumar los elementos de dos vectores

Declarar registros vectoriales

```
...
__m128 rega,regb,regc;

for (i=0; i<size; i++) {
 a[i]=b[i]+c[i];
}

...
```

# Sumar los elementos de dos vectores

Cambiar el paso del bucle

```
...

for (i=0; i<size; i=i+4) {
 a[i]=b[i]+c[i];
}

...
```

# Sumar los elementos de dos vectores

Procesar los elementos vectorialmente

```
...
for (i=0; i<size; i=i+4) {

  // Carga
regb =_mm_load_ps(&(b[i]));
regc=_mm_load_ps(&(c[i]));

 // Suma
 rega=_mm_add_ps(regb,regc);

 // Almacenamiento
 _mm_store_ps(&(c[i]),rega);
}
```

# Apartado 1. Cálculo de la norma de cada fila de una matriz almacenada en formato comprimido

Vectorizar la operación producto (*) considerando:

1. Cada elemento es de tipo double.
2. Cada elemento es de tipo float.

```
for (i=0; i<nfilas; i++) {
   normafila = 0;
      for (j=0; j<dimfilas[i]; j++) {
         normafila += valores[i][j] * valores[i][j];
      }
   norma[i] = sqrt(normafila);
}
```

# Apartado 1. Cálculo de la norma de cada fila de una matriz almacenada en formato comprimido

Problemas a resolver:

1. Alineamiento de los datos.
2. El límite del bucle puede no ser múltiplo del número de elementos que van a ser procesados simultáneamente en cada iteración del bucle vectorizado.

```
for (i=0; i<nfilas; i++) {
   normafila = 0;
      for (j=0; j<dimfilas[i]; j++) {
         normafila += valores[i][j] * valores[i][j];
      }
   norma[i] = sqrt(normafila);
}
```

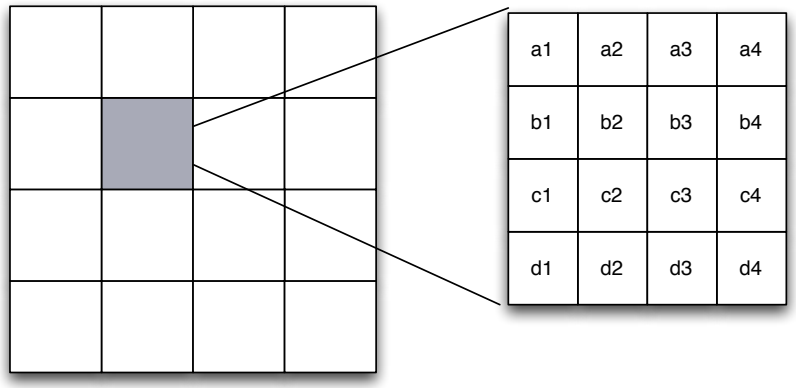
# Apartado 2. Cálculo de la norma de cada fila de una matriz densa

Vectorizar todas las operaciones (*,+,sqrt) que aparecen en el bucle.

```
for (i=0; i<nfilas; i++) {
  normafila = 0;
    for (j=0; j<ncols; j++) {
      normafila += valores[i*ncols+j]*valores[i*ncols+j];
     }
  norma[i] = sqrt(normafila);
}
```

# Apartado 2. Cálculo de la norma de cada fila de una matriz densa

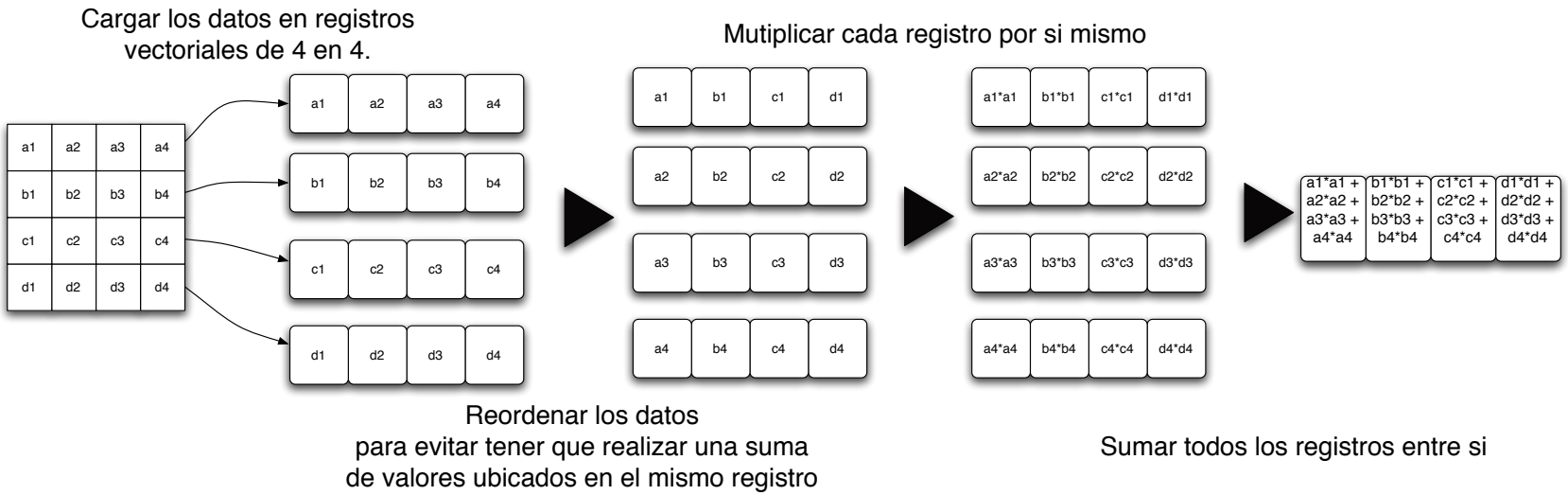
La matriz se procesa en subloques del tamaño apropiado. En el caso de datos tipo float serán bloques de tamaño 4x4.




## Operación a realizar

a1*a1 + a2*a2 + a3*a3 + a4*a4
b1*b1 + b2*b2 + b3*b3 + b4*b4
c1*c1 + c2*c2 + c3*c3 + c4*c4
d1*d1 + d2*d2 + d3*d3 + d4*d4

# Apartado 2. Cálculo de la norma de cada fila de una matriz densa




## Operación a realizar

a1*a1 + a2*a2 + a3*a3 + a4*a4
b1*b1 + b2*b2 + b3*b3 + b4*b4
c1*c1 + c2*c2 + c3*c3 + c4*c4
d1*d1 + d2*d2 + d3*d3 + d4*d4