Introducción a SIMD

Organización del Computador II

(Slides de Gonza)
Presenta Mariano

Departamento de Computación

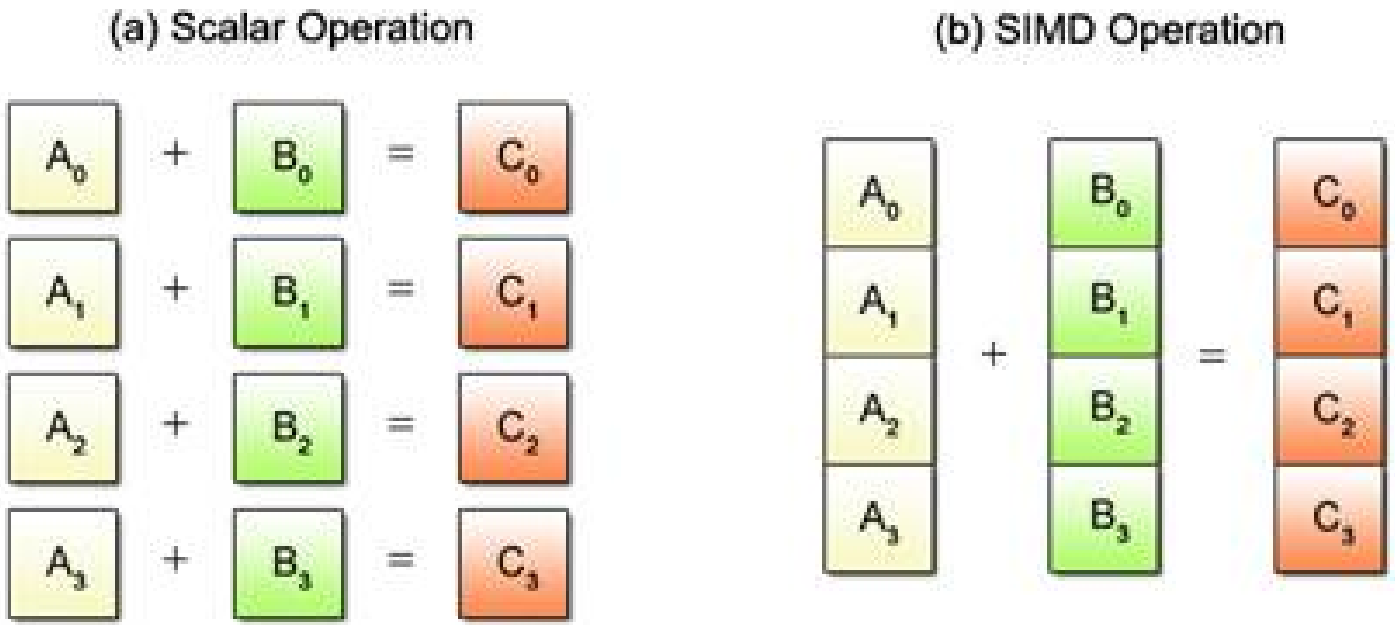
2018-05-10

Procesamiento vectorial, ¿qué vamos a ver?

Single **I**nstrucction, **S**ingle **D**ata

vs

Single **I**nstruction, **M**ultiple **D**ata



Procesamiento vectorial, ¿para qué sirve?

- ¿Siempre podemos usar **SIMD**?

  No. Hay muchos algoritmos que no se pueden adaptar a este tipo de procesamiento. Por ejemplo, lo que están haciendo en el **tp1**.

- ¿En qué casos es útil?

  Para *procesamiento multimedia*, es decir, procesamiento de **imágenes**, **videos** y **audio**, y cualquier otro tipo de procesamiento que involucre aplicar la misma operación sobre una gran cantidad de datos.

Implementación del modelo SIMD

- **SSE** (**S**treaming **SI**MD **E**xtensions) es un set de instrucciones que implementa el modelo de cómputo **SIMD**
- Se introdujo por primera vez en el año 1999 por **Intel** como sucesor de **MMX** (introducido en 1997)
- **SSE** extiende a **MMX** con nuevos registros y nuevos tipos de datos
- **SSE** se fue extendiendo hasta llegar a la versión **SSE4.2**, nuevas instrucciones
- Además en nuevos procesadores se implemento un nuevo set de instrucciones denominado **AVX**

Tipos de datos / Registros

SSE puede operar con los siguientes tipos de datos

- **enteros** (de 8, 16, 32, 64 y 128 bits)
- **floats**
- **doubles** (a partir de **SSE2**)

Cuenta con 16 registros de **128 bits** (**16** bytes)

- **XMM0**, **XMM1**, ... , **XMM15**

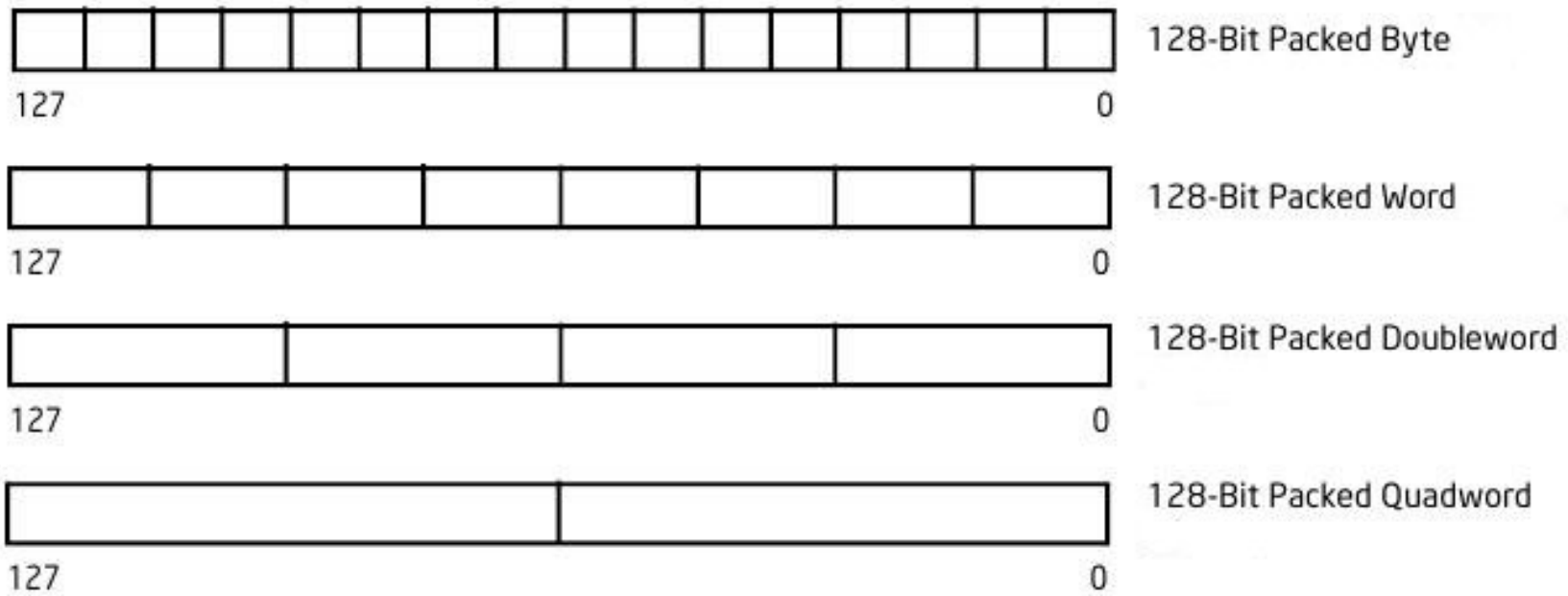
Empaquetamiento

En un registro de **SSE** (**16 bytes**) podemos poner

- **16** datos de tamaño **1 byte** (entero (**char**))
- **8** datos de tamaño **2 bytes** (entero (**short**))
- **4** datos de tamaño **4 bytes** (entero (**int**))
- **2** datos de tamaño **8 bytes** (entero (**long long int**))
- **4** datos de tamaño **4 bytes** (punto flotante (**float**))
- **2** datos de tamaño **8 bytes** (punto flotante (**double**))

Esto se lo conoce como *empaquetamiento* (más de un dato en un mismo registro).

Empaquetamiento



¿Cómo sabemos que tipo de datos tenemos dentro de un registro?

No hay manera. Ustedes escriben el código, por ende, ustedes saben que están poniendo en cada registro. Desde instrucciones no pueden saber ni el tamaño ni el tipo de los datos.

¿Cómo operar sobre esos registros?

El set de instrucciones **SSE** brinda operaciones de

- Movimiento de datos
- Aritméticas
- Lógicas
- de Comparación
- Trascendentales

Se caracterizan según

- **tipo** de dato con el que operan
- **tamaño** de dato con el que operan

Instrucciones: ¿Cómo leer el set de instrucciones?

(La gran mayoría siguen una regla)

Para enteros: comienzan con `P`, luego el nombre de la *operación* y terminan con el *tamaño* del dato

Ejemplo:

- `PADDB`: suma de a **B**yte
- `PADDW`: suma de a **W**ord
- `PADDD`: suma de a **D**oubleword
- `PADDQ`: suma de a **Q**uadword

Para punto flotante: nombre de la *operación*, *modo* de operación y *tamaño* del dato.

Ejemplo:

- `ADDPS`: suma de a **F**loat
- `ADDPD`: suma de a **D**ouble

La `P` proviene de `Packed`, podría ser `S` de `Scalar`

Ejercicio 1

Realizar el producto interno de dos vectores de floats

- La longitud de ambos vectores es **n**, donde n es un short
- n es múltiplo de 4
- El prototipo de la función es:

```
float productoInternoFloats(float* a, float* b, short n)
```

Producto Interno

$$\langle a, b \rangle = \sum_{i=0}^{n-1} a[\,i\,] * b[\,i\,]$$

Ejercicio 1: Solución

```
productoInternoFloats:   ; rdi = a, rsi = b; dx = n

push rbp
mov rbp, rsp
xor rcx, rcx             ; Contador
mov cx, dx
shr rcx, 2               ; Proceso de a 4 elementos
pxor xmm7, xmm7          ; Acumulador
.ciclo:
                         ; Cargar los valores
     movups xmm1, [rdi]  ; xmm1 = a3 | a2 | a1 | a0
     movups xmm2, [rsi]  ; xmm2 = b3 | b2 | b1 | b0
                         ; Multiplicar
     mulps xmm1, xmm2    ; xmm1 = a3*b3 | a2*b2 | a1*b1 | a0*b0
                         ; Acumular el resultado
     addps xmm7, xmm1    ; xmm7 = sum3 | sum2 | sum1 | sum0
     add rdi, 16         ; Avanzar los punteros
     add rsi, 16
     loop .ciclo
```

Ejercicio 1: Solución

```
                        ; Sumar todo
movups xmm6, xmm7       ; xmm6 =  sum3 | sum2 | sum1 | sum0
psrldq xmm6, 8          ; xmm6 =   0   |   0   | sum3 | sum2
addps xmm7, xmm6        ; xmm7 =  .  | .  | s3 + s1 | s2 + s0

movups xmm6, xmm7       ; xmm6 = .  | .  | s3 + s1 | s2 + s0
psrldq xmm6, 4          ; xmm6 = .  | .  |    .    | s3 + s1
addps xmm7, xmm6        ; xmm7 = .  | .  |    .    | sumatoria

movss xmm0, xmm7        ; xmm0 = ... | ... | ... | sumatoria

pop rbp
ret
```

Supongamos que queremos pasar una imagen a escala de grises.

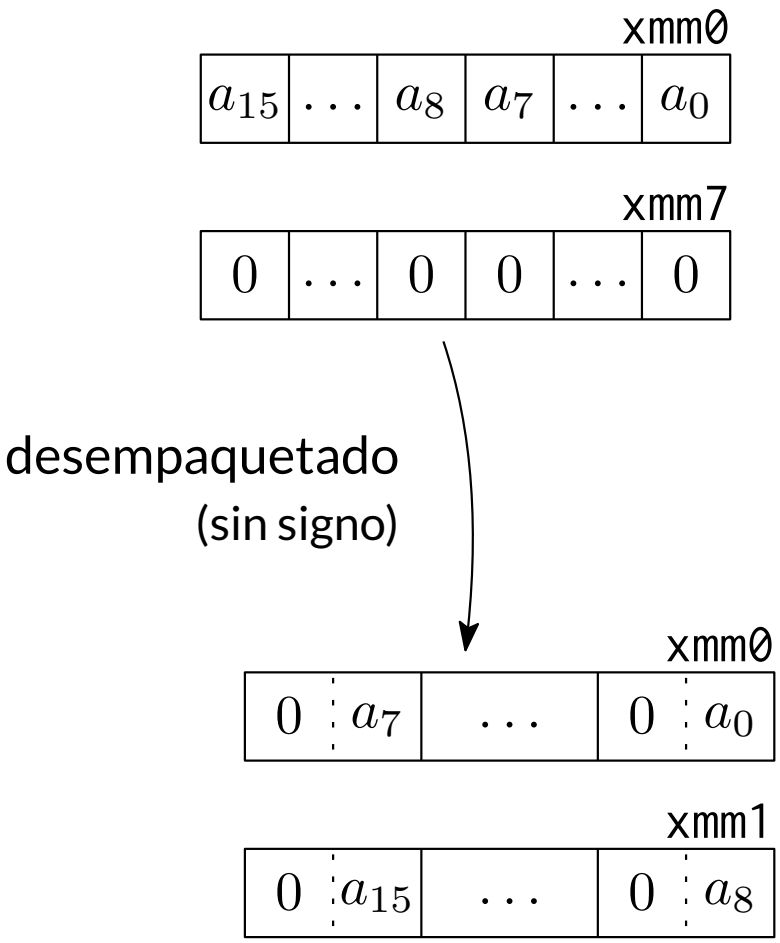
Una forma de hacerlo es mediante la fórmula:

$$f(r, g, b) = \frac{1}{4} \cdot (r + 2g + b)$$

¡Atención! **Puede haber overflow en la suma.**

- Para no perder información en los cálculos, es necesario manejar los resultados intermedios en un tipo de datos con mayor rango.
  - El tipo de datos original es de **byte**.
- Deberíamos operar con datos en tamaño **word**.
- ¿Cómo hacemos esto?
  Utilizando las instrucciones de desempaquetado.

Desempaquetado
En código




```
pxor xmm7, xmm7
movdqu xmm1, xmm0
punpcklbw xmm0, xmm7
punpckhbw xmm1, xmm7
```

Ahora tenemos los valores originales en tamaño **word**.

Desempaquetado: `punpck{l,h}{bw,wd,dq,qdq}`

- `l`, `h`
  parte baja (**l**ow) o alta (**h**igh)
- `bw`, `wd`, `dq`, `qdq`
  de **b**yte a **w**ord, de **w**ord a **d**word, etc. . .

Empaquetado: `pack{ss,us}{wb,dw}`

- `ss`, `us`
  signed / **u**nsigned, con **s**aturación
- `wb`, `dw`
  de **w**ord a **b**yte, de **d**word a **w**ord

En SSE también existen instrucciones de comparación, aunque se comportan un poco diferente a lo que veníamos usando.

Claramente **no se puede usar saltos condicionales** porque estamos trabajando con muchos datos a la vez.

Entonces **se usan máscaras** obtenidas a partir de comparaciones.

	-7	42	-5	57
$<$	0	0	0	0
	1	0	1	0

Supongamos que estamos trabajando con **words** y queremos saber cuáles de ellos son menores a cero.

Datos en `xmm0`

1000	-456	-15	0	100	234	-890	1

```
pxor xmm7, xmm7 ; xmm7 = 0 | 0 | ... | 0
pcmpgtw xmm7, xmm0 ; xmm7 > xmm0 ?
```

Resultado en `xmm7`

0x0000	0xFFFF	0xFFFF	0x0000	0x0000	0x0000	0xFFFF	0x0000

Es decir, compara **word a word** y si se cumple la condición setea **unos** (0xFFFF en este caso) en el resultado, o **ceros** si no.

La comparación nos devuelve un registro con unos y ceros.

Podemos usarlo para:

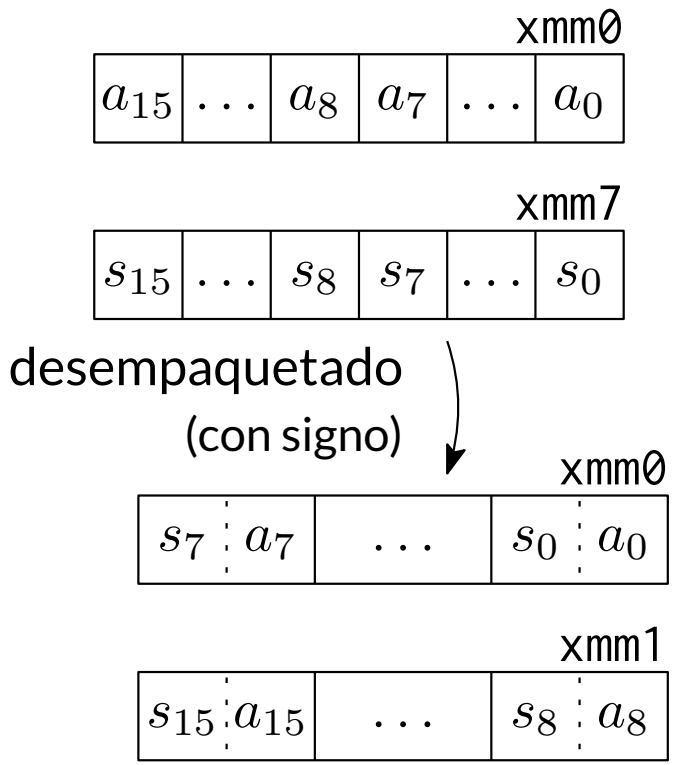
- **Extender el signo** de números signados al desempaquetar. Ya que los números negativos se extienden con unos, y los positivos con ceros.
  - $-5 = 1011 \rightsquigarrow 1111\,1011$
  - $+5 = 0101 \rightsquigarrow 0000\,0101$
- **Como una máscara**, acompañado de instrucciones como: `PAND`, `POR`, etc.

Para extender el signo del registro del ejemplo anterior

xmm0							
1000	-456	-15	0	100	234	-890	1

Si tenemos el resultado de la comparación en `xmm7`

```
movdqu xmm1, xmm0
punpckhwd xmm0, xmm7
punpcklwd xmm1, xmm7
```



Cálculo de Padding
¿Qué es el padding?

- Como vimos anteriormente, el modelo de procesamiento SIMD se usa para procesamiento multimedia, en particular de imágenes.
- Además vimos que hay instrucciones, como por ejemplo `mov`, que están optimizadas para mover datos alineados.

Supongamos que tenemos la siguiente matriz floats, que representan la intensidad de cada pixel en una imagen de escala de grises de 3x5:

0.13	0.1	0.13	0.34	0.99
0.14	0.913	0.15	0.36	0.0
0.94	0.15	0.43	0.59	0.9

Introducimos un padding al final de cada fila de manera que la siguiente esté alineada a 16 bytes. De esta manera, la imagen anterior estará representada de la siguiente manera en memoria:

0.13	0.1	0.13	0.34	0.99			
0.14	0.913	0.15	0.36	0.0			
0.94	0.15	0.43	0.59	0.9			

Las celdas azules son porciones de memoria de 4 bytes que se utilizan como padding. Notar que ahora cada fila está alineada a 16 bytes.

Ojo: Las celdas de color negro no son nuestras, por lo que accederlas podría causar un invalid read.

Cálculo de Padding
Recorriendo la matriz

- Para Recorrer una fila que no es la última, no nos importa levantar la memoria del padding y procesarla, **sin embargo, esto depende fuertemente de lo que estemos haciendo con la matriz, en general conviene descartar los datos, por ejemplo, haciendo una operación de shift**.
- La última fila es más delicada, y en general procesar su última parte debe ser procesada por separado, de tal manera de no leer memoria que no nos pertenece. Una forma es, por ejemplo, volver para atrás y leer floats que ya leímos, aunque hay que tener en cuenta que el acceso a memoria **no será alineado (usar** `movaps` **provocará un segfault)**.

Cálculo de Padding
Pasándolo a código...

0.13	0.1	0.13	0.34	0.99			
0.14	0.913	0.15	0.36	0.0			
0.94	0.15	0.43	0.59	0.9			

Supongamos que `rbx` apunta a la posición de memoria marcada con rosa. Como dijimos antes, es incorrecto hacer

```
movaps xmm0, [rbx] ; MAL!!
```

Pues estamos leyendo las posiciones memoria negras. Una forma correcta podría ser:

```
sub rbx, 12
movups xmm0, [rbx] ; desalineado!
                   ; xmm0 = 0.9 | 0.59 | 0.43 | 0.15
psrldq xmm0, 12    ; xmm0 = 0   |  0   |  0   | 0.9
```

Procesamiento vectorial, ¿en dónde me voy a equivocar?

- Usar instrucciones de enteros para punto flotante y viceversa.
- No desempaquetar enteros y causar overflow (o desempaquetar cuando no hace falta).
- Usar instrucciones que no existen (en el parcial).
- Usar branches en vez de máscaras.

¿Preguntas?