SIMD
Pack/Unpack
Saturación
Comparación y Máscaras

Organización del Computador II

18 de abril de 2017

Hoy vamos a ver

1. Brevísimo repaso
2. Instrucciones comunes
   - Saturación
   - Empaquetamiento/Desempaquetamiento
   - Comparación y Máscaras
3. Ejercicios
   - Máxima distancia
   - Normalización

Brevísimo repaso

- SSE (Streaming SIMD Extensions) es un set de instrucciones que implementa el **modelo de cómputo SIMD** (una misma instrucción para varios datos a la vez).
- Existen 16 **registros de 128 bits** (XMM0...XMM15).
- Varios posibles **tipos de datos**: **enteros**, **flotantes de precisión simple** (32 bits) y **flotantes de precisión doble** (64 bit).
- **Formas de operar**:
  - P = *Packed* (**Empaquetado / Paralelo**).
  - S = *Scalar* (**Escalar**).
- Sufijos para las instrucciones:
  - SS: scalar single-precision (float)
  - SD: scalar double-precision (double)
  - PS: packed single-precision (float)
  - PD: packed double-precision (double)
- Las instrucciones que comienzan con *P* sólo son para operar sobre enteros (ejemplo PADDD). Las otras, sin *P*, son para operar sobre flotantes (ejemplo ADDPS).

Saturación

Ejemplo: Escribir una función que aumente el brillo de una imagen en blanco y negro utilizando instrucciones SSE.

Saturación

Se tiene el brillo de cada píxel de la imagen de entrada en una matriz `I` de `N` filas por `M` columnas, y un parámetro `b` entero según el cual se debe incrementar cada píxel:

```
unsigned char I[N][M];
unsigned char b;
```

Saturación

Se tiene el brillo de cada píxel de la imagen de entrada en una matriz `I` de `N` filas por `M` columnas, y un parámetro `b` entero según el cual se debe incrementar cada píxel:

```
unsigned char I[N][M];
unsigned char b;
```

Podríamos hacer:

```
for( int i = 0; i < N; i++ )
    for( int j = 0; j < M; j++ )
        I[i][j] += b;
```

Saturación

Se tiene el brillo de cada píxel de la imagen de entrada en una matriz `I` de `N` filas por `M` columnas, y un parámetro `b` entero según el cual se debe incrementar cada píxel:

```
unsigned char I[N][M];
unsigned char b;
```

Podríamos hacer:

```
for( int i = 0; i < N; i++ )
    for( int j = 0; j < M; j++ )
        I[i][j] += b;
```

¿Qué pasaría si `I[i][j]+b` *fuera mayor a 255?*

Saturación: ejemplo de incremento de brillo sin saturación

(a) Original

(b) +50 de brillo

(c) +100 de brillo

Saturación: problemas con la no saturación

El resultado de la suma podría no entrar en el registro.

- Se trunca el resultado.
- Se pasa de colores más claros (cercanos al 255) a colores más oscuros (cercanos al 0).

Saturación: problemas con la no saturación

El resultado de la suma podría no entrar en el registro.

- Se trunca el resultado.
- Se pasa de colores más claros (cercanos al 255) a colores más oscuros (cercanos al 0).

Una forma de evitar esto, podría ser:

Incremento de brillo con saturación

```
for( int i = 0; i < N; i++ )
    for( int j = 0; j < M; j++ )
        I[i][j] = min( 255, I[i][j] + b );
```

Saturación: ejemplo de incremento de brillo con saturación

(d) Original

(e) +50 de brillo

(f) +100 de brillo

Saturación: instrucciones comunes

Como esto es algo común en el procesamiento de señales, existen instrucciones específicas para operar de esta manera:

- **PADDUSB**/**PSUBUSB**: Suma/Resta enteros sin signo con saturación sin signo.
- **PADDSB**/**PSUBSB**: Suma/Resta enteros con signo con saturación con signo.

En este caso son operaciones de a **byte**. También existen de a **word**. Ver el manual de Intel.

Empaquetamiento/Desempaquetamiento

Ejemplo: Pasar una imagen RGB a escala de grises.

Empaquetamiento/Desempaquetamiento

Una forma muy común de hacer ésto es mediante la fórmula:

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

Empaquetamiento/Desempaquetamiento

Una forma muy común de hacer ésto es mediante la fórmula:

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

- ¿Qué podría pasar con $(r + 2g + b)$?

Empaquetamiento/Desempaquetamiento

Una forma muy común de hacer ésto es mediante la fórmula:

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

- ¿Qué podría pasar con $(r + 2g + b)$?
- En esta situación, es necesario manejar los resultados intermedios en un tipo de datos de mayor precisión para no perder información en los cálculos. La precisión original es de **byte**.
- Lo primero que podemos hacer es pasar los datos de **byte** a algo más grande (en este caso nos alcanza con pasar a **word**).
- ¿Cómo lo hacemos?

Empaquetamiento/Desempaquetamiento

Una forma muy común de hacer ésto es mediante la fórmula:

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

- ¿Qué podría pasar con $(r + 2g + b)$?
- En esta situación, es necesario manejar los resultados intermedios en un tipo de datos de mayor precisión para no perder información en los cálculos. La precisión original es de **byte**.
- Lo primero que podemos hacer es pasar los datos de **byte** a algo más grande (en este caso nos alcanza con pasar a **word**).
- ¿Cómo lo hacemos?
  Utilizando las instrucciones de desempaquetado.

Empaquetamiento/Desempaquetamiento

- Si tenemos un registro con números sin signo:

$$\textbf{xmm1} = a_{15} \mid a_{14} \mid \ldots \mid a_1 \mid a_0$$

- ¿Cómo duplicamos la precisión o el tamaño de los datos?

Empaquetamiento/Desempaquetamiento

- Si tenemos un registro con números sin signo:

$$\textbf{xmm1} = a_{15} \,|\, a_{14} \,|\, \ldots \,|\, a_1 \,|\, a_0$$

- ¿Cómo duplicamos la precisión o el tamaño de los datos?
  Solamente necesitamos agregar ceros delante de cada número.

$$\textbf{xmm1} = 0 \,|\, a_7 \,|\, \ldots \,|\, 0 \,|\, a_0$$

$$\textbf{xmm2} = 0 \,|\, a_{15} \,|\, \ldots \,|\, 0 \,|\, a_8$$

Empaquetamiento/Desempaquetamiento

En ensamblador sería así:

```
                          ; xmm1 = a15 | a14 | ... | a1 | a0

pxor xmm7, xmm7           ; xmm7 = 0 | 0 | ... | 0

movdqu xmm2, xmm1         ; xmm2 = xmm1 = a15 | a14 | ... | a1 | a0

punpcklbw xmm1, xmm7      ; xmm1 = 0 | a7 | ... | 0 | a0
punpckhbw xmm2, xmm7      ; xmm2 = 0 | a15 | ... | 0 | a8
```

Ahora cada dato es de tipo **word**.

Empaquetamiento/Desempaquetamiento

Después de extender los datos, realizamos las operaciones que necesitamos.

Y al final, tenemos que guardar los datos nuevamente. Y como en este caso representan píxeles de una imagen en escala de grises, deberían seguir siendo bytes, por lo que tenemos que volver a convertirlos a **byte**.

- ¿Cómo hacemos la conversión?

Empaquetamiento/Desempaquetamiento

Después de extender los datos, realizamos las operaciones que necesitamos.

Y al final, tenemos que guardar los datos nuevamente. Y como en este caso representan píxeles de una imagen en escala de grises, deberían seguir siendo bytes, por lo que tenemos que volver a convertirlos a **byte**.

- ¿Cómo hacemos la conversión?
  Empaquetando los datos.

Las instrucciones de empaquetamiento son varias, tienen en cuenta distintos tipos de datos y si los datos tienen signo o no. Por ejemplo, en el caso de **byte** tenemos:

- **packsswb** (saturación con signo)
- **packuswb** (saturación sin signo)

Empaquetamiento/Desempaquetamiento

Formato de instrucciones:

- **Desempaquetado:** *punpck* + *l/h* + *bw/wd/dq/qdq*
- **Empaquetado:** *pack* + *ss/us* + *wb/dw*
- **Empaquetado:** *pack* + *ss/us* + *wb/dw*

Importante a tener en cuenta:

- ¿Queremos usar la parte alta o la parte baja?
- ¿De qué tipo son los datos con los que estamos trabajando?
- Entonces: ¿Qué tipo de saturación tengo que usar?

Empaquetamiento/Desempaquetamiento

Volviendo al ejemplo anterior:

$$\textbf{xmm1} = 0 \,|\, a_7 \,|\, \ldots \,|\, 0 \,|\, a_0$$

$$\textbf{xmm2} = 0 \,|\, a_{15} \,|\, \ldots \,|\, 0 \,|\, a_8$$

Entonces:

packuswb xmm1, xmm2 ; xmm1 $= a_{15} \,|\, a_{14} \,|\, \ldots \,|\, a_1 \,|\, a_0$

Ahora cada dato es de tipo **byte**

Empaquetamiento/Desempaquetamiento

Pequeños consejos para trabajar en estos casos:

- Leer los datos a procesar.
- Extender la precisión o tamaño de los datos (*unpack*).
- Hacer las cuentas que tenemos que hacer.
- Volver a la precisión o tamaño original (*pack*).
- Guardar los datos procesados.

Comparación

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

Suponiendo que estamos trabajando con **word**s en xmm1 y queremos saber cuáles de ellos son menores a cero:

$$\textbf{xmm1} = 1000 \,|\, -456 \,|\, -15 \,|\, 0 \,|\, 100 \,|\, 234 \,|\, -890 \,|\, 1$$

Comparación

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

Suponiendo que estamos trabajando con **word**s en xmm1 y queremos saber cuáles de ellos son menores a cero:

$$\textbf{xmm1} = 1000\,|\,-456\,|\,-15\,|\,0\,|\,100\,|\,234\,|\,-890\,|\,1$$

```
pxor xmm7, xmm7          ; xmm7 = 0|0|...|0
pcmpgtw xmm7, xmm1       ; xmm7 > xmm1 ?
```

Comparación

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

Suponiendo que estamos trabajando con **word**s en xmm1 y queremos saber cuáles de ellos son menores a cero:

$$\textbf{xmm1} = 1000\,|\,-456\,|\,-15\,|\,0\,|\,100\,|\,234\,|\,-890\,|\,1$$

```
pxor xmm7, xmm7          ; xmm7 = 0|0|...|0
pcmpgtw xmm7, xmm1       ; xmm7 > xmm1 ?
```

El resultado de la comparación queda en xmm7 así:

$$\textbf{xmm7} = \\ 0x0000\,|\,0xFFFF\,|\,0xFFFF\,|\,0x0000\,|\,0x0000\,|\,0x0000\,|\,0xFFFF\,|\,0x0000$$

O sea, compara **word** a **word**. Si se cumple la condición, entonces setea todo en **unos** ($0xFFFF$) el resultado, o en **ceros** si no.

Comparación: ejemplos comunes de uso

Las instrucciones de comparación nos devuelve un registro con unos y ceros.

Entonces, podríamos usar ese registro para:

- **Extender el signo:**

  Aprovechando que los números negativos se extendienden con 1s, y los positivos con 0s. Por ejemplo:

  - `-5 = 1011` se extiende a `1111 1011`
  - `5 = 0101` se extiende a `0000 0101`

- **Generar máscaras:**

  Muy útiles para *filtrar*, *enmascarar*, o lo que necesitemos hacer sólo a algunos elementos del registro. Utilizamos además instrucciones como *PAND*, *POR*, *PXOR*, etc.

Comparación: ejemplo de extensión de signo

Para extender el signo de los **word**s en xmm1:

$$\textbf{xmm1} = 1000\,|\,-456\,|\,-15\,|\,0\,|\,100\,|\,234\,|\,-890\,|\,1$$

Genero el registro con la comparación en xmm7:

```
pxor xmm7, xmm7          ; xmm7 = 0|0|...|0
pcmpgtw xmm7, xmm1       ; xmm7 > xmm1 ?
```

Y extendemos:

```
movdqu xmm2, xmm1        ; copio xmm1
punpckhwd xmm1, xmm7     ; xmm1 = 1000|-456|-15|0
punpcklwd xmm2, xmm7     ; xmm2 = 100|234|-890|1
```

Comparación: ejemplo de uso de máscaras

Para sumar 3 a los números **word**s en xmm1 que son menores a 0:

$$\textbf{xmm1} = 1000\,|\,-456\,|\,-15\,|\,0\,|\,100\,|\,234\,|\,-890\,|\,1$$

Genero el registro con la comparación en xmm7:

```
pxor xmm7, xmm7          ; xmm7 = 0|0|...|0
pcmpgtw xmm7, xmm1       ; xmm7 > xmm1 ?
```

Generamos la máscara de *3*s sólo en los "lugares" que nos interesan:

```
                         ; xmm2 = 3|3|3|3|3|3|3|3
pand xmm7, xmm2          ; xmm7 = 0|3|3|0|0|0|3|0
```

Sumamos:

```
paddw xmm1, xmm7    ; xmm1 = 1000|−453|−12|0|100|234|−887|1
```

Ejercicio 1: Máxima distancia

Calcular la distancia máxima entre puntos.

- Los puntos **(x,y)** están almacenados como dos números contiguos de punto flotante de precisión simple.
- **n** indica la cantidad de puntos, y es múltiplo de 2.
- Se deben procesar dos puntos en paralelo.
- La distancia se calcula entre los puntos de v y los de w, elemento a elemento.
- El prototipo de la función es:

```
float maximaDistancia( float *v, float *w, unsigned short n );
```

Ayuda: distancia entre dos puntos

$$\text{dist}((x,y),(x',y')) = \sqrt{(x-x')^2 + (y-y')^2}$$

Ejercicio 1: Máxima distancia — Resolución

Obtenemos los parámetros, chequeamos que los vectores no estén vacíos y preparamos los registros para el ciclo:

```
global maximaDistancia

section .text

maximaDistancia:          ; rdi = v, rsi = w; dx = n
  xor rcx, rcx
  mov cx, dx              ; rcx = n

                          ; limpio xmm0 para usarlo como temporal
  xorps xmm0, xmm0        ; xmm0 = 0 | 0 | 0 | 0

  cmp rcx, 0              ; chequeo si los vectores están vacíos
  je .fin

  shr rcx, 1              ; rcx = n / 2
```

Ejercicio 1: Máxima distancia — Resolución

```
.ciclo:

    movups xmm1, [rdi]      ; xmm1 = y1 | x1 | y0 | x0
    movups xmm2, [rsi]      ; xmm2 = y1' | x1' | y0' | x0'
    subps xmm1, xmm2        ; xmm1 = (y1 - y1') | (x1 - x1') | (y0 - y0') | (x0 - x0')
    mulps xmm1, xmm1        ; xmm1 = (y1 - y1')^2 | (x1 - x1')^2 | (y0 - y0')^2 | (x0 - x0')^2
    movaps xmm2, xmm1
    psrldq xmm2, 4          ; xmm2 = 0 | (y1 - y1')^2 | (x1 - x1')^2 | (y0 - y0')^2
    addps xmm1, xmm2        ; xmm1 = ? | ((x1 - x1')^2 + (y1 - y1')^2)| ? | ((x0 - x0')^2 + (y0 - y0')^2)
    sqrtps xmm1, xmm1       ; xmm1 = ? | sqrt((x1 - x1')^2 + (y1 - y1')^2)| ? | sqrt((x0 - x0')^2 + (y0 - y0')^2)
    maxps xmm0, xmm1        ; xmm0 = ? |  max(dist_impares) | ? |  max(dist_pares)

    add rdi, 16             ; avanzo los punteros
    add rsi, 16
    loop .ciclo
```

Ejercicio 1: Máxima distancia — Resolución

Tenemos ahora dos máximos (pares e impares). Calculamos cuál de los dos es el mayor. Y terminamos la función:

xmm0 = ? | **max(dist_impares)** | ? | **max(dist_pares)**

```
movdqu xmm1, xmm0     ; xmm1 = xmm0
psrldq xmm1, 8        ; xmm1 = 0 | 0 |?|  max(dist_impares)
maxps xmm0, xmm1      ; xmm0 = ?|?|?| max(dist_pares, dist_impares)

.fin:
ret                   ; retorno
```

Ejercicio 1: Máxima distancia — Resolución

Para pensar:

- ¿Qué pasa si ahora tenemos que comparar todos los puntos de v contra todos los de w?
- ¿Y si **n** no fuera múltiplo de 2?
- ¿Podemos procesar más de 2 puntos en paralelo?

Ejercicio 2: Normalización

Normalizar un vector.

- **n** representa la longitud del vector y es múltiplo de 4.
- Se debe devolver un vector, de manera que el máximo elemento sea uno, y el mínimo cero.
- Se debe procesar la máxima cantidad posible de elementos en paralelo.
- El prototipo de la función es:

```
float *normalizarVector( float *v, int n );
```

Ayuda: para normalizar buscamos máx, mín y aplicamos a cada elemento

$$x[i] = \frac{(x[i] - min)}{(max - min)}$$

Ejercicio 2: Normalización — Resolución

Obtenemos los parámetros y pedimos memoria para el vector a devolver:

```
extern malloc                ; rdi = v, esi = n
global normalizarVector

section .text

normalizarVector:
  push rbp
  mov rbp, rsp               ; armo el stack frame
  push r12
  push r13

  mov r12, rdi               ; obtengo los parámetros
  xor r13, r13
  mov r13d, esi

  mov rdi, r13               ; rdi = n
  shl rdi, 2                 ; rdi = n * 4
  call malloc                ; pido memoria (n*4)
```

Ejercicio 2: Normalización — Resolución

Preparamos los registros para el ciclo y buscamos dentro del vector el máximo y el mínimo:

```
mov rcx, r13            ; rcx = n
shr rcx, 2              ; rcx = n / 4

mov rdi, r12            ; rdi = v
movups xmm0, [rdi]      ; xmm0 = valores iniciales para mínimos
movups xmm1, [rdi]      ; xmm1 = valores iniciales para máximos

.ciclo:
  movups xmm2, [rdi]    ; bajo 4 valores
  minps xmm0, xmm2      ; xmm0 = mínimos actualizados
  maxps xmm1, xmm2      ; xmm1 = máximos actualizados

  add rdi, 16           ; avanzo el puntero
  loop .ciclo
```

Ejercicio 2: Normalización — Resolución

Consigo el mínimo de los mínimos en *xmm*0 y lo "replico" en cada posición de *xmm*0:

movdqu xmm2, xmm0	; xmm2 = xmm0
psrldq xmm2, 4	; xmm2 = 0 | $xmm0_3$ | $xmm0_2$ | $xmm0_1$
minps xmm0, xmm2	; xmm0 = ? | $min(xmm0_2, xmm0_3)$ | ? | $min(xmm0_0, xmm0_1)$
movdqu xmm2, xmm0	
psrldq xmm2, 8	; xmm2 = 0 | 0 |?| $min(xmm0_2, xmm0_3)$
minps xmm2, xmm0	; xmm2 = ? |?|?|$min(xmm0_0, xmm0_1, xmm0_2, xmm0_3)$
xorps xmm0, xmm0	; xmm0 = 0 | 0 | 0 | 0
addss xmm0, xmm2	; xmm0 = 0 | 0 | 0 | *min*
pslldq xmm0, 4	; xmm0 = 0 | 0 | *min* | 0
addss xmm0, xmm2	; xmm0 = 0 | 0 | *min* | *min*
movaps xmm3, xmm0	; xmm3 = 0 | 0 | *min* | *min*
pslldq xmm3, 8	; xmm3 = *min* | *min* | 0 | 0
addps xmm0, xmm3	; xmm0 = *min* | *min* | *min* | *min*

Ejercicio 2: Normalización — Resolución

Lo mismo para el máximo de los máximos en $xmm1$. Al final, obtenemos el valor $(max - min)$ también replicado en cada posición:

movdqu xmm2, xmm1	; xmm2 = xmm1
psrldq xmm2, 4	; xmm2 = $0 \mid xmm1_3 \mid xmm1_2 \mid xmm1_1$
maxps xmm1, xmm2	; xmm1 = $? \mid max(xmm1_2, xmm1_3) \mid ? \mid max(xmm1_0, xmm1_1)$
movdqu xmm2, xmm1	
psrldq xmm2, 8	; xmm2 = $0 \mid 0 \mid ? \mid max(xmm1_2, xmm1_3)$
maxps xmm2, xmm1	; xmm2 = $? \mid ? \mid ? \mid max(xmm1_0, xmm1_1, xmm1_2, xmm1_3)$
xorps xmm1, xmm1	; xmm1 = $0 \mid 0 \mid 0 \mid 0$
addss xmm1, xmm2	; xmm1 = $0 \mid 0 \mid 0 \mid max$
pslldq xmm1, 4	; xmm1 = $0 \mid 0 \mid max \mid 0$
addss xmm1, xmm2	; xmm1 = $0 \mid 0 \mid max \mid max$
movaps xmm3, xmm1	; xmm3 = $0 \mid 0 \mid max \mid max$
pslldq xmm3, 8	; xmm3 = $max \mid max \mid 0 \mid 0$
addps xmm1, xmm3	; xmm1 = $max \mid max \mid max \mid max$
subps xmm1, xmm0	; xmm1 = $(max - min) \mid (max - min) \mid (max - min) \mid (max - min)$

Ejercicio 2: Normalización — Resolución

Calculamos cada elemento del nuevo vector, lo escribimos, y listo:

```
mov rcx, r13                ; rcx = n
shr rcx, 2                  ; rcx = n / 4
mov rdi, r12                ; rdi = v
mov rsi, rax                ; rsi = vector nuevo

.ciclo2:
  movups xmm2, [rdi]        ; xmm2 = X_{i+3} | X_{i+2} | X_{i+1} | X_i
  subps xmm2, xmm0          ; xmm2 = X_{i+3} - min | ... | ... | X_i - min
  divps xmm2, xmm1          ; xmm2 = (X_{i+3} - min)/(max - min) | ... | ... | (X_i - min)/(max - min)
  movups [rsi], xmm2        ; escribo los valores normalizados al nuevo vector

  add rdi, 16               ; avanzo punteros
  add rsi, 16
  loop .ciclo2

pop r13
pop r12
pop rbp
ret                         ; retorno
```