SIMD

Nadia Heredia

Organización del Computador II, 2do cuat. de 2014

Recordando

- 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

Volviendo al ejemplo del aumento de brillo, teníamos el siguiente código en **C**:

Incremento de brillo

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

¿Qué pasa si $I[i][j] + b$ es mayor que 255? Por ejemplo, con $b = 100$

Ejemplo de incremento de brillo sin saturación

(a) Original, brillo (b = 50), brillo (b = 100)

Problemas con la no saturación

El problema es que el que el resultado de la suma no entra en el registro.

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

Para evitar esto, habría que hacer

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)
```

Entonces, obtenemos

Ejemplo de incremento de brillo con saturación

(b) Original, brillo saturado (b = 50), brillo saturado (b = 100)

Instrucciones para saturación

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 la suma es de a **byte**, también está para **word**.

Empaquetamiento/Desempaquetamiento

Supongamos que queremos pasar una imagen a escala de grises. Una forma de hacerlo es mediante la fórmula:

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

- 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 tenemos que hacer es pasar los datos de **byte** a algo más grande (en este caso nos alcanza con pasar a **word**).
- ¿Cómo aumentamos la precisión?
  Utilizando las instrucciones de desempaquetado.

Empaquetamiento/Desempaquetamiento

- Si tenemos un registro con números sin signo:

$$\textbf{xmm0} = a_{15} \,|\, a_{14} \,|\, \ldots \,|\, a_1 \,|\, a_0$$

- Sólo necesitamos agregar cero delante de cada número. Es decir,

$$\textbf{xmm0} = 0 \,|\, a_7 \,|\, \ldots \,|\, 0 \,|\, a_0$$

$$\textbf{xmm1} = 0 \,|\, a_{15} \,|\, \ldots \,|\, 0 \,|\, a_8$$

Empaquetamiento/Desempaquetamiento

Concretamente:

pxor xmm7, xmm7	; xmm7 $= 0 \mid 0 \mid \ldots \mid 0$
movdqu xmm2, xmm1	; xmm2 $=$ xmm1 $= a_{15} \mid a_{14} \mid \ldots \mid a_1 \mid a_0$
punpcklbw xmm1, xmm7	; xmm1 $= 0 \mid a7 \mid \ldots \mid 0 \mid a0$
punpckhbw xmm2, xmm7	; xmm2 $= 0 \mid a15 \mid \ldots \mid 0 \mid a8$

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

Empaquetamiento/Desempaquetamiento

Después de extender los datos, realizamos las operaciones que necesitamos. Finalmente, tenemos que guardar los datos (que en este caso representarían los píxeles de una imagen en escala de grises), 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 concreto de **byte** tenemos:

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

Formato de instrucciones

Desempaquetado *punpck* + *l/h* + *bw/wd/dq/qdq*

Empaquetado *pack* + *ss/us* + *wb/dw*

Tener en cuenta

- si queremos usar la parte alta o la parte baja
- los tipo de datos con los que trabajamos
- el tipo de saturación a usar (cuando empaquetamos)

Empaquetamiento/Desempaquetamiento

Concretamente, si:

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

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

Entonces:

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

Ahora cada dato es de tipo **byte**

Empaquetamiento/Desempaquetamiento. Resumen.

Resumiendo, la forma de trabajar en estos casos es

- Leer datos a procesar.
- Extender precisión (unpack).
- Realizar las cuentas necesarias.
- Volver a la precisión original (pack).
- Guardar datos procesados.

Comparación

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

Por ejemplo, supongamos que estamos trabajando con **words** y queremos saber cuáles de ellos son menores a cero.
Si tenemos los datos en xmm1, por ejemplo:

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

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

Obtenemos el resultado de la comparación en xmm7:

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

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

Comparación - Usos

Como vimos en el ejemplo, la comparación nos devuelve un registro con unos y ceros.

Podemos usar este registro

- Para extender el signo de números signados al pasar de un tipo de datos a otro.
  Usando el hecho de que los números negativos se van a extender con unos, y los positivos con ceros.
  - -5 = 1011 se extiende a 1111 1011
  - 5 = 0101 se extiende a 0000 0101
- A modo de máscaras, acompañado de instrucciones como: *PAND*, *POR*, etc.

Comparación - Extensión de signo

Para extender el signo del registro del ejemplo anterior

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

Si tenemos el resultado de la comparación en xmm7, hacemos

```
movdqu xmm2, xmm1       ; copio xmm1
punpckhwd xmm1, xmm7    ; xmm1 = 1000|...|0
punpcklwd xmm2, xmm7    ; xmm2 = 100|...|1
```

Comparación - Uso de máscaras

Supongamos ahora que se desea sumar 3 a los números menores a 0.
Al igual que antes, tenemos los números en xmm1, y el resultado de la comparación en xmm7.
Tenemos el número 3, que queremos sumar, en un registro

$$\textbf{xmm2} = 3\,|\,3\,|\ldots|\,3$$

Hacemos

```
pand xmm7, xmm2      ; xmm7 = 0 | 3 | ... | 3 | 0
paddw xmm1, xmm7     ; xmm1 = 1000 | −453 | ... | −887 | 1
```

Ejercicio 1

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);
```

Distancia

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

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            ; obtener los parámetros
  mov cx, dx              ; rcx = n

                          ; limpiar xmm0 para usar como temporal
  xorps xmm0, xmm0        ; xmm0 = 0 | 0 | 0 | 0

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

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

Resolución

ciclo:

movups xmm1, [rsi]	; xmm1 = $y_2 \mid x_2 \mid y_1 \mid x_1$
movups xmm2, [rdi]	; xmm2 = $y_2' \mid x_2' \mid y_1' \mid x_1'$
subps xmm1, xmm2	; xmm1 = $(y_2 - y_2') \mid (x_2 - x_2') \mid (y_1 - y_1') \mid (x_1 - x_1')$
mulps xmm1, xmm1	; xmm1 = $(y_2 - y_2')^2 \mid (x_2 - x_2')^2 \mid$
	$(y_1 - y_1')^2 \mid (x_1 - x_1')^2$
movdqu xmm2, xmm1	
psrldq xmm2, 4	; xmm2 = $0 \mid (y_2 - y_2')^2 \mid (x_2 - x_2')^2 \mid (y_1 - y_1')^2$
addps xmm1, xmm2	; xmm1 = $* \mid ((x_2 - x_2')^2 + (y_2 - y_2')^2) \mid$
	$* \mid ((x_1 - x_1')^2 + (y_1 - y_1')^2)$
sqrtps xmm1, xmm1	; xmm1 = $* \mid \sqrt{(x_2 - x_2')^2 + (y_2 - y_2')^2} \mid$
	$* \mid \sqrt{(x_1 - x_1')^2 + (y_1 - y_1')^2}$
maxps xmm0, xmm1	; xmm0 = $* \mid$ max(dist_impares) $\mid$
	$* \mid$ max(dist_pares)
add rsi, 16	; avanzar los punteros
add rdi, 16	
loop ciclo	

Resolución

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

xmm0 $= * \mid$ **max(dist_impares)** $\mid * \mid$ **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                    ; retornar
```

Terminamos

Para hacer:

- ¿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

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);
```

Debemos buscar el máximo y el mínimo, y aplicar a cada elemento

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

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               ; armar el stack frame
  push r12
  push r13

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

  mov rdi, r13               ; rdi = n
  shl rdi, 2                 ; rdi = n * 4
  call malloc                ; pedir memoria
```

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]        ; setear valores iniciales para los máximos y mínimos
movups xmm1, [rdi]

ciclo:
  movups xmm2, [rdi]      ; leer los próximos 4 valores
  minps xmm0, xmm2        ; actualizar mínimos
  maxps xmm1, xmm2        ; actualizar máximos

  add rdi,16              ; avanzar el puntero
  loop ciclo
```

Resolución

Conseguimos el mínimo de $xmm0$ y lo ponemos en cada posición

movdqu xmm2, xmm0	; xmm2 = xmm0
psrldq xmm2, 4	; xmm2 = $0 \mid xmm0_3 \mid xmm0_2 \mid xmm0_1$
minps xmm0, xmm2	; xmm0 = $* \mid min(xmm0_2, xmm0_3) \mid$ $* \mid min(xmm0_0, xmm0_1)$
movdqu xmm2, xmm0	
psrldq xmm2, 8	; $xmm2 = 0 \mid 0 \mid * \mid min(xmm0_2, xmm0_3)$
minps xmm2, xmm0	; $xmm2 = * \mid * \mid * \mid min(xmm0_0, xmm0_1, xmm0_2, xmm0_3)$
xorps xmm0, xmm0	; xmm0 = $0 \mid 0 \mid 0 \mid 0$
addss xmm0, xmm2	; xmm0 = $0 \mid 0 \mid 0 \mid min$
pslldq xmm0, 4	; xmm0 = $0 \mid 0 \mid min \mid 0$
addss xmm0, xmm2	; xmm0 = $0 \mid 0 \mid min \mid min$
pslldq xmm0, 4	; xmm0 = $0 \mid min \mid min \mid 0$
addss xmm0, xmm2	; xmm0 = $0 \mid min \mid min \mid min$
pslldq xmm0, 4	; xmm0 = $min \mid min \mid min \mid 0$
addss xmm0, xmm2	; xmm0 = $min \mid min \mid min \mid min$

Resolución

Realizamos algo similar para el máximo y le restamos el mínimo al final

movdqu xmm2, xmm1	; xmm2 = xmm1
psrldq xmm2, 4	; xmm2 = $0 \mid xmm1_3 \mid xmm1_2 \mid xmm1_1$
maxps xmm1, xmm2	; xmm1 = $* \mid min(xmm1_2, xmm1_3) \mid$ $* \mid max(xmm1_0, xmm1_1)$
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 = $0 \mid 0 \mid max \mid max$
pslldq xmm1, 4	; xmm1 = $0 \mid max \mid max \mid 0$
addss xmm1, xmm2	; xmm1 = $0 \mid max \mid max \mid max$
pslldq xmm1, 4	; xmm1 = $max \mid max \mid max \mid 0$
addss xmm1, xmm2	; xmm1 = $max \mid max \mid max \mid max$
subps xmm1, xmm0	; xmm1 = diferencia entre el máximo y el mínimo 4 veces

Resolución

Llenamos las posiciones del nuevo vector, asignamos el valor de retorno, y restauramos el stackframe

```
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        ; copiar valores normalizados al nuevo vector

  add rdi, 16               ; avanzar punteros
  add rsi, 16
  loop ciclo2

pop r13
pop r12
pop rbp
ret                         ; retornar
```

¿Preguntas?