Organización del Computador II
TP2

27 de abril de 2015

Integrante	LU	Correo electrónico
Martin Baigorria	575/14	martinbaigorria@gmail.com
Martin Baigorria	575/14	martinbaigorria@gmail.com
Gonzalo Ciruelos Rodríguez	063/14	gonzalo.ciruelos@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Introducción

El presente trabajo practico tiene como objetivo aprender a utilizar instrucciones que operan con múltiples datos SIMD (Single instruction, multiple data), soportada por la familia de procesadores x86-64 de Intel. En general, este tipo de instrucciones se utilizan mucho en procesamiento digital de señales y procesamiento de gráficos. Debido a restricciones de la cátedra, solo utilizaremos SSE (Streaming SIMD Extensions) y no el set de instrucciones mas nuevo AVX (Advanced Vector Extensions).

1.1. Historia

El set de instrucciones MMX (MultiMedia eXtension) fue el primero en implementar SIMD en la arquitectura Intel en el año 1997, con los procesadores Pentium II. La idea era, como lo indica el nombre, permitir el procesamiento de operaciones sobre múltiples datos en simultaneo. Esto en general se conoce como paralelismo a nivel de datos, donde un único proceso es capaz de ejecutar estas instrucciones.

MMX define 8 nuevos registros de 64 bits, desde el MM0 al MM7. A nivel arquitectura, estos registros eran simplemente un alias de los registros de la FPU, por lo que cualquier operación en la FPU afecta a los registros MMn.

En el año 1999, con el objetivo de responder a las mejoras introducidas por la competencia, en general por el AltiVec en las PowerPc de Motorola y el sistema POWER de IBM, Intel introduce el set de instrucciones SSE con la serie de procesadores Pentium III. El set de instruccciones SSE tenia 70 nuevas instrucciones. Aqui Intel agrega 8 nuevos registros de 128bits, XMM0 a XMM7, independientes de la FPU. Este set de instrucciones luego se actualizo con SSE2 (2001), SSE3 (2004), SSSE3 (2006) y SSE4(2006).

Figura 1: Intel Pentium P5

Figura 2: Power PC

En el año 2011 finalmente se introduce al mercado las extensiones al set de instrucciones AVX (Advanced Vector Extensions), primero implementado por la linea de procesadores de Intel Sandy Bridge. AVX expande el tamanio de los registros de 128 a 256 bits, y se renombra a estos registros como YMM0-YMM7. Introduce las operaciones vectorizadas de tres operandos.

Intel introduce AVX2 en el año 2013 con la linea de procesadores Haswell, que expande muchas de las instrucciones de SSE y AVX a 256 bits.

Actualmente Intel esta por lanzar este año la linea de procesadores Xeon Phi Knights Landing, que busca expandir las operaciones y los registros de AVX2 a 512 bits.

1.2. Filtros de Imágenes

Una aplicación típica de este tipo de instrucciones es el procesamiento de imágenes. Esto se debe a que una imagen esta representada por un mapa de pixeles, y en general al procesar imágenes se esta llevando a cabo el mismo tipo de procedimiento muchas veces. A continuación se analizaran diferentes filtros de imágenes, y luego se cuantificara la ganancia en tiempo al implementar los filtros utilizando SSE.

A fines prácticos, solo utilizaremos imágenes de tamaño mxn en formato bmp cuya longitud sea múltiplo de 4. Notaremos a j como el indice de las filas, a i como el indice de las columnas y a k como el indice de los componentes de color de cada pixel. $m[j][i][k]$ por lo tanto sera el componente k del pixel asociado a la fila j, columna i.

2. Blur

2.1. Introducción

Blur es un filtro que suaviza la imagen. Esto lo hace asignándole a cada pixel el promedio (media aritmética) con sus pixeles vecinos. Es decir:

$m[j][i][k] = (m[j-1][i-1][k] + m[j-1][i][k] + m[j-1][i+1][k] +$
$m[j][i-1][k] + m[j][i][k] + m[j][i+1][k] +$
$m[j+1][i-1][k] + m[j+1][i][k] + m[j+1][i+1][k])/9$

Notar que esto significa que no vamos a procesar los pixeles en los bordes. Esto se debe a que no tienen la cantidad suficiente de pixeles vecinos.

También debemos tener en cuenta el hecho de que a medida que procesemos pixeles, no podemos simplemente pisarlos en memoria. Esto se debe a que un pixel adyacente luego lo puede necesitar para calcular el promedio, por lo que estaríamos mezclando la imagen nueva y la vieja. Hay dos opciones para resolver este problema:

1. Asignar en memoria un nuevo espacio para guardar los nuevos pixeles. Luego se deben agregar los bordes que no fueron procesados.

2. En primer lugar, guardar las primeras dos filas del la imagen en una nueva posición de memoria. De esa manera podemos pisar los pixeles sobre la imagen original, evitar pedir demasiada memoria y tener que rearmar la imagen al final.

La opción mas elegante y sin lugar a duda correcta es la 2, ya que en términos de construcción y de memoria es mucho mas eficiente.

2.2. C

El filtro blur de la cátedra en primer lugar guarda los valores de los pixeles en la primera fila. Arranca en la segunda fila, la copia, y va procesando todos los pixeles de a uno, tomando el promedio de cada componente y reemplazandolo en el mapa de pixeles principal. Esto lo hace por medio de dos loops anidados, uno que recorre las filas, y otro que recorre las columnas. A medida que pasa de fila, intercambia los punteros de la fila recién copiada y la ultima fila copiada. Esto se debe a que cuando pasamos de fila ya no nos interesa saber cuales eran los valores originales dos filas antes. Luego copia la fila actual en el puntero a la fila mas vieja. Se ignora la primera y la ultima fila, y también la primera y la ultima columna. Esto se debe a que no tienen la cantidad suficiente de pixeles vecinos.

Insert pseudo-code here?

2.3. ASM1

Siguiendo la idea del código en C, ASM1 recorre el mapa de pixeles de la misma manera. El código en C procesa cada componente del pixel por separado, mientras que la idea de esta version es procesar todos los componentes de un pixel con SSE. La idea nuevamente es iterar toda la imagen, primero por filas y luego por columnas, reemplazando los pixeles en las posiciones correspondientes.

	P1	P2	P3	P4
MEM	P5	P6	P7	P8
	P9	P10	P11	P12

En primer lugar, copiamos la fila correspondiente al contador de filas. Esto da inicio al loop de las filas.

Luego, en el loop de las columnas, buscamos en la copia de la primera fila los 4 pixeles (16 bytes) correspondientes al iterador actual de las columnas. Cada pixel ocupa 32 bits, 1 byte por cada componente ARGB. En la memoria, los archivos *.bmp* guardan los componentes de los pixeles en el orden A B G R. Como la arquitectura Intel es little-endian, al mover estos pixeles a un registro, no solo se invertira el orden de los pixeles sino que también el de sus componentes.

MEM	$A_1B_1G_1R_1$	$A_2B_2G_2R_2$	$A_4B_3G_3R_3$	$A_4B_4G_4R_4$
XMM1	$R_4G_4B_4A_4$	$R_4G_3B_3A_3$	$R_2G_2B_2A_2$	$R_1G_1B_1A_1$

Aquí nosotros hemos levantado 4 pixeles de memoria. Sin embargo notar que para el promedio de los vecinos de un pixel solo necesitamos los 3 primeros. Por lo tanto limpiamos el registro `XMM1` con un shift a la izquierda y luego uno a la derecha:

XMM1	0	$R_4G_3B_3A_3$	$R_2G_2B_2A_2$	$R_1G_1B_1A_1$

Luego, hacemos las operaciones de empaquetado y desempaquetado para expandir el tamaño de cada componente de pixel de 8 a 16 bits. Esto lo hacemos para que mas adelante cuando tengamos que sumar no tengamos overflow al sumar los componentes de dos pixeles.

XMM1	R_2	G_2	B_2	A_2	R_1	G_1	B_1	A_1
XMM2	0	0	0	0	R_3	G_3	B_3	A_3

Ahora sumamos `XMM1` y `XMM2` , poniendo el resultado en `XMM1` :

XMM1	R_2	G_2	B_2	A_2	R_1+R_3	G_1+G_3	B_1+B_3	A_1+A_3

Repetimos este procedimiento tres veces, uno para cada fila. Finalmente nos queda:

XMM1	R_2	G_2	B_2	A_2	R_1+R_3	G_1+G_3	B_1+B_3	A_1+A_3
XMM2	R_6	G_6	B_6	A_6	R_5+R_7	G_5+G_7	B_5+B_7	A_5+A_7
XMM3	R_{10}	G_{10}	B_{10}	A_{10}	R_9+R_{11}	G_9+G_{11}	B_9+B_{11}	A_9+A_{11}

Ahora sumamos los tres registros en `XMM1` . Por cuestiones de claridad, lo representamos de a pixeles únicos:

XMM1	3B	3G	3B	3A	6R	6G	6B	6A

Copiamos `XMM1` en `XMM2` y lo shifteamos 8 bytes a la derecha:

XMM2	0	0	0	0	3B	3G	3B	3A

Sumando `XMM1` y `XMM2` en `XMM1` :

XMM1	3B	3G	3B	3A	9R	9G	9B	9A

Ahora empaqueto la parte baja del registro para que cada componente de cada pixel pase de 1 a 2 bytes y poder ganar presicion al momento de dividir por 9.

XMM1	9R	9G	9B	9A

Tomo el promedio de los componentes de cada pixel dividiendo por el registro | 9.0 | 9.0 | 9.0 | 9.0 |.

XMM1	R_p	G_p	B_p	A_p

Finalmente escribo el registro en la posición de memoria correspondiente. Luego incremento el contador de las columnas o de las filas y vuelvo al ciclo correspondiente.

2.4. ASM2

2.5. Comentarios

Al comparar utilizando la imagen de diferencias la imagen generada por el blur de la catedra y la de ASM1, notamos lo siguiente:

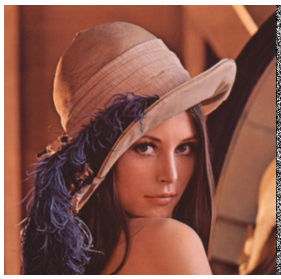

Figura 3: Blur C

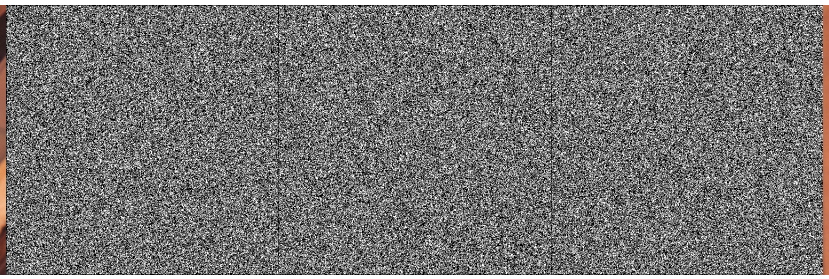

Figura 4: R channel Figura 5: G channel Figura 6: B channel

Figura 7: Blur ASM

Aunque las imágenes parecían exactamente iguales, habían diferencias en algunos pixeles. Estas diferencias se debian al redondeo hacia arriba llevado a cabo por la conversion desde float a entero. Mientras que el codigo en C redondeaba hacia abajo por default, las conversiones en ASM redondeaban hacia arriba por default.

Para resolver esto, existe un flag de SSE llamado $MXCSR$. Para mas información, ver la sección 10.2.3.1 del Volumen 1 de la guía de arquitectura Intel. Cada bit de este flag codifica algún comportamiento de las operaciones SSE. El valor por default de este flag es $0x1F80$. Para que redondee hacia abajo, utilizando la codificación de los bits del flag, había que poner el bit 12 y 13 en 1. Por lo tanto, simplemente setteamos el flag con la instruccion *ldmxcsr* en $0x7F80$ y empezó a dar perfecto.

3. Experimentacion Blur

4. Merge

4.1. C

El código de C es bastante simple. Lo que hace al inicio es castear ambos punteros a punteros a matrices de matrices, que tienen elementos de w bytes de tamaño, que a su vez tienen elementos de 4 bytes de tamaño.

Luego se realizan 3 loops, el primero sobre las filas, el segundo sobre las columnas, y el tercero sobre los canales (exceptuando el canal de transparencia, notar que ii se inicializa en 1).

Finalmente se realiza la cuenta que se debe hacer para hacer el merge, casteando todo adecuadamente (se castea el valor del canal, que es un entero de 1 byte a float, se lo multiplica por value o 1-value segun corresponda, se lo suma con el otro valor y se lo vuelve a castear a entero de 8 bytes).

4.2. ASM1

En la primera versión del codigo de ASM, iniciamos guardando los parámetros que nos pasan en algunos registros auxiliares para no perderlos mas adelante. Luego calculamos h*w dado que es la cantidad de iteraciones que vamos a realizar.

`RCX` nos servirá de iterador y `RBX` de puntero a la posición actual.

Luego precalculamos dos vectores que vamos a usar en todas las iteraciones

`XMM3`	$value$	$value$	$value$	1,0
`XMM4`	$1 - value$	$1 - value$	$1 - value$	0,0

Luego empieza el loop principal de la rutina. Aquí cargamos 1 pixel de cada imagen (solamente podemos cargar un pixel porque vamos a hacer operaciones con punto flotante de precisión simple, entonces cargar 2 pixeles en una misma iteracion tiene el mismo costo que cargar hacerlo en 2 iteraciones separadas.

Después hacemos unpacking para llevarlos a la forma que queremos y los convertimos a float, y los multiplicamos con los vectores que anteriormente habiamos generado, convenientemente. Entonces los registros tendran la pinta

`XMM1`	$R_1 value$	$G_1 value$	$B_1 value$	$A_1$1,0
`XMM2`	$R_2(1 - value)$	$G_2(1 - value)$	$B_2(1 - value)$	$A_2$0,0

Luego los sumamos en xmm1, y queda la suma

`XMM1`	$R_2(1 - value) + R_1 value$	$G_2(1 - value) + G_1 value$	$B_2(1 - value) + B_1 value$	$A_2$0,0 + $A_1$1,0

Luego, lo pasamos a enteros de 32 bits, y lo packeamos a enteros de 8 bits nuevamente. Notar que no se produce overflow. Tomemos por ejemplo el canal R, y supongamos que el valor del canal R para ambas imágenes es 255. Entonces $255 value + 255(1 - value)$ da 255, que entra en 8 bits (si por errores de redondeo diera más, no importa, ya que el packing lo hacemos saturado, entonces se queda en 255).

Finalmente escribimos en el lugar de la memoria correspondiente el valor obtenido y listo, repetimos este proceso hasta terminar.

4.3. ASM2

Al igual que en la rutina anterior, luego de armar el stackframe guardamos los parametros en algunos registros auxiliares para no perderlos. El resto del proceso es el mismo que antes, hasta que llegamos al precalculo de los vectores de *value*. Como aquí hay que hacer operaciones con enteros, en vez de punto flotante como antes, vamos a calcular las cosas de manera diferente.

En vez de guardarnos *value*, vamos a guardarnos $int16(256 * value)$. ¿Por qué? Porque de esta manera vamos a poder cargar de a dos pixeles y hacer las operaciones mucho mas rápido que antes. Además así guardamos un valor que antes estaba entre 0 y 1 en un valor que esta entre 0 y 256, es decir, un valor que podemos representar con una buena aproximación entera.

De esta menera nos armamos el vector igual que antes, solo que multiplicado por 256. Luego lo pasamos a entero y luego lo packeamos consigo mismo en la parte alta.

```
cvtps2dq xmm3, xmm3
packuswb xmm3, xmm3
```

De esta manera obtenemos en `XMM3` los siguientes valores

`XMM3`

$256v$	$256v$	$256v$	256	$256v$	$256v$	$256v$	256

Donde v es *value*.

Ahora, nos gustaría tener en el otro registros los numeros tal que, sumados con los de `XMM3` , dan 256. Para eso, nos aprovechamos de la representación complemento a 2, dado que lo que queremos en realidad son los inversos aditivos (en 8 bits) de estos números en el registro `XMM4` . Entonces usamos que el inverso aditivo de un número es el negado bit a bit más 1.

Cargamos en `XMM4` 8 enteros de 16 bits con valor 257, ya que es 256+1. Luego, al hacer la diferencia

```
psubw xmm4, xmm3
```

obtenemos en `XMM4` exactamente lo que queremos, es decir que si sumamos int16 a int16 en `XMM3` y `XMM4` da 256.

Luego comenzamos el loop principal. Cargamos lo que nos interesa (2 pixeles) y los unpackeamos a words (16 bits), con ceros en la parte alta. Luego hacemos los productos de las partes bajas

```
pmullw xmm1, xmm3
pmullw xmm2, xmm4
```

De esta manera, los registros quedan de la siguiente forma

`XMM1`	$R_{11}\lceil 256v\rceil$	$B_{11}\lceil 256v\rceil$	$B_{11}\lceil 256v\rceil$	$256A_{11}$	$R_{12}\lceil 256v\rceil$	$G_{12}\lceil 256v\rceil$	$B_{12}\lceil 256v\rceil$	$256A_{12}$
`XMM2`	$R_{21}(256-\lceil 256v\rceil$	$B_{21}(256-\lceil 256v\rceil)$	$B_{21}(256-\lceil 256v\rceil)$	0	$R_{22}(256-\lceil 256v\rceil)$	$G_{22}(256-\lceil 256v\rceil)$	$B_{22}(256-\lceil 256v\rceil)$	0

En realidad, en cada uno de esos registros tenemos la parte baja del producto, pero es importante notar que la parte alta es siempre 0. ¿Por qué? Simplemente porque el canal siempre es menor o igual que 255 y $\lceil 256v\rceil$ es menor o igual que 256, entonces el producto es siempre menor que 65280, que en especial es menor que 65535, el máximo numero representable por enteros sin signo en 16 bits.

5. Experimentacion Merge

6. HSL

7. Experimentacion HSL

8. Conclusiónes