DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

SIMD

Organización del Computador II

Primer Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Alejandro Mignanelli	609/11	minga_titere@hotmail.com
Facuuuuu	XXX/xx	chaboooooon@hotmail.com
Iaaaaaaan	XXX/xx	me_la_super_como@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Resumen

En el presente trabajo se describe la problemática de procesar información de manera eficiente cuando los mismos requieren:

1. Transferir grandes volúmenes de datos.
2. Realizar las mismas instrucciones sobre un set de datos importante.

Índice

1. Objetivos generales

El objetivo de este Trabajo Práctico es mostrar las variaciones en la performance que pueden ocurrir al utilizar instrucciones SIMD cuando se manejan grandes volúmenes de datos que requieren un procesamiento similar, en comparación con implementaciones que no lo utilizan.

Para ello se realizarán distintos experimentos sobre tres filtros de foto, Blur, Merge y HSL, tanto en código assembler, que aproveche las instrucciones SSE brindadas para los procesadores de arquitectura Intel, como en código C compilado con gcc, al que se le aplicará el grado de optimización grados de optimización -O3.

FALTA PONER QUE QUIERO VER PARTICULARMENTE DE CADA UNO!!!!

2. Preámbulo

2.1. Calidad de las Mediciones

3. Experimentación

4. Blur

4.1. Idea de las implementaciones

A continuación se explicará un ciclo de ejecución de cada una de las implementaciones de Blur:

4.1.1. Implementación 1

Debido a que este filtro usa pixeles de otras posiciones de la imágen original para modificar cada pixel, se tiene una copia de la imágen cuyo objetivo es tener un lugar seguro que no .[en]sucia"los datos de entrada de cada pixel debo modificar. Esto se hace antes de un ciclo de ejecución del algoritmo, pero es importante remarcarlo. Ahora si, una vez dentro de un ciclo de ejecución, lo primero que hace nuestro algoritmo es cargar en tres xmm pixeles de manera tal de tener en un xmm los vecinos superiores, y un pixel extra, en otro xmm los vecinos inferiores y un pixel extra, y en otro los vecinos del pixel a modificar y un pixel extra, como se muestra a continuación.

(cada campo tiene 32 bits)
$xmm0 = basura|vecino_{sup1}|vecino_{sup2}|vecino_{sup3}$

$xmm1 = basura|vecino_3\ |pixel\ |vecino_4$

$xmm2 = basura|vecino_{inf6}|vecino_{inf7}|vecino_{inf8}$

Esto lo cargamos desde la imagen copia. Como tomamos un pixel que no tiene nada que ver con el filtro, lo transformamos en 0 y desempaquetamos los tres xmm, obteniendo lo siguiente:

(cada campo tiene 64 bits ; r,g,b,a tienen tamaño word cada una)

$xmm0 = 0|b_{sup1}\ g_{sup1}\ r_{sup1}\ a_{sup1}$

$xmm1 = b_{sup2}\ g_{sup2}\ r_{sup2}\ a_{sup2}|b_{sup3}\ g_{sup3}\ r_{sup3}\ a_{sup3}$

$xmm2 = 0|b_4\ g_4\ r_4\ a_4$

$xmm3 = b_{pixel}\ g_{pixel}\ r_{pixel}\ a_{pixel}|b_5\ g_5\ r_5\ a_5$

$xmm4 = 0|b_{inf6}\ g_{inf6}\ r_{inf6}\ a_{inf6}$

$xmm5 = b_{inf7}\ g_{inf7}\ r_{inf7}\ a_{inf7}|b_{inf8}\ g_{inf8}\ r_{inf8}\ a_{inf8}$

Luego se realiza una suma vertical entre todos los xmm, obteniendo asi un xmm que tiene en una mitad la suma de una parte de los vecinos, y en la otra mitad la suma de la otra parte de los vecinos. Desempaquetamos ese xmm en dos xmm y realizamos nuevamente una suma vertical, obteniendo asi un xmm con la suma de todos los vecinos del pixel a modificar. Convertimos ese numero a float, lo dividimos por nueve (que son la cantidad de vecinos del pixel), lo volvemos a convertir a int, y luego empaquetamos el resultado hasta tener el tamaño original del pixel en un sector de un xmm. Luego, utilizando la función movmaskdqu, insertamos en la imagen original el pixel modificado, y seteamos los registros para volver a entrar a la iteración para modificar el próximo píxel.

4.1.2. Implementación 2

Esta implementación es similar a la anterior, con la excepción de que se procesaran 4 píxeles por iteración en vez de uno. Al igual que en la implementación anterior, se copiará la imágen previamente, y luego se entrará al ciclo. Dentro del ciclo,para procesar el primer pixel, se cargaran en tres xmm pixeles de manera tal de tener en un xmm los vecinos superiores, y un pixel extra, en otro xmm los vecinos inferiores y un pixel extra, y en otro los vecinos del pixel a modificar y un pixel extra, de igual manera que en blur 1. Luego, se procesará de igual manera que en blur 1 pero en vez de empaquetarlo y cambiarlo en la imágen, se guardará el resultado en un xmm. Se hará lo mismo con el procesamiento del segundo pixel. Para el píxel tres se cargarán xmm inicialmente de la misma forma, pero se hará una copia de los xmm

cargados, para ser usados en el pixel 4. Se procesa el pixel 3 de igual manera que el pixel dos y uno. Para procesar el pixel cuatro, se reacomodarán las copias de los xmm cargados en el procesamiento del pixel 3, de manera tal de ”simular”que se hubieran cargado los pixels como en blur 1. Para eso, notemos que el pixel extra del procesamiento del pixel 3, es un vecino del pixel 4, y que el primer vecino de cada xmm del pixel 3, es el pixel extra del 4. Despues de este reordenamiento, se procederá de igual manera que con los pixeles 1, 2 y 3. Luego de esto, se tienen en 4 xmm los 4 píxeles procesados, se empaquetan de manera tal de tener los 4 pixeles en un xmm y se insertan en la imágen original.

4.2. Diferencias de performance en Blur

4.2.1. Resultados

4.2.2. Conclusiones

5. Merge

5.1. Idea de las implementaciones

A continuación se explicará un ciclo de ejecución de cada una de las implementaciones de Merge (tener en cuenta que este filtro toma una imágen original, y una imágen que filtra):

5.1.1. Implementación 1

De las dos imágenes, tomamos 4 píxeles y los insertamos en un xmm(uno por imágen). A cada uno de estos, los desempaquetamos hasta obtener en 4 xmm, los cuatro pixeles de una imagen, y en otros 4 xmm los píxeles de la segunda imágen. Cada uno de los xmm anteriores, los convertimos en floats.

(cada campo tiene 32 bits)
Imágen original:

$xmm3 = b3_{orig}|g3_{orig}|r3_{orig}|a3_{orig}$ (cada campo es un float)

$xmm2 = b2_{orig}|g2_{orig}|r2_{orig}|a2_{orig}$ (cada campo es un float)

$xmm1 = b1_{orig}|g1_{orig}|r1_{orig}|a1_{orig}$ (cada campo es un float)

$xmm0 = b0_{orig}|g0_{orig}|r0_{orig}|a0_{orig}$ (cada campo es un float)

Imágen Filtro:

$xmm7 = b3_{filt}|g3_{filt}|r3_{filt}|a3_{filt}$ (cada campo es un float)

$xmm6 = b2_{filt}|g2_{filt}|r2_{filt}|a2_{filt}$ (cada campo es un float)

$xmm5 = b1_{filt}|g1_{filt}|r1_{filt}|a1_{filt}$ (cada campo es un float)

$xmm4 = b0_{filt}|g0_{filt}|r0_{filt}|a0_{filt}$ (cada campo es un float)

Luego, multiplicamos los xmm correspondientes a la imágen original por value y a los xmm de la imágen filtro por 1-value. Después de esto, sumamos los xmm que representan un pixel de una imágen con su correspondiente xmm de la otra imágen(o sea, aquel que representa el mismo número de píxel) y los guardamos en xmm, obteniendo 4 xmm que representan los 4 píxeles filtrados. Los convertimos nuevamente a int, y los empaquetamos de manera tal de tener en un xmm los 4 píxeles. Luego los insertamos en la imágen original.

5.1.2. Implementación 2

5.2. Diferencias de performance en Merge

5.2.1. Resultados

5.2.2. Conclusiones

6. HSL

6.1. Idea de las implementaciones

A continuación se explicará un ciclo de ejecución de cada una de las implementaciones de Hsl:

6.1.1. Implementación 1

Esta implementación usa dos funciones en c, rgbtohsl y hsltorgb que hacen lo que sus respectivos nombres indican.

6.1.2. Implementación 2

6.2. Diferencias de performance en HSL

6.2.1. Resultados

6.2.2. Conclusiones

7. Conclusiones y trabajo futuro