DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

subtitulo del trabajo

Organización del Computador II
Segundo Cuatrimestre de 2014

Integrante	LU	Correo electrónico
Borgna, Agustín	079/15	aborgna@dc.uba.ar
Corleto, Alan	XXX/XX	mail
Lancioni, Franco	XXX/XX	mail

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 ...

Índice

Figura 1: Descripcion de la figura

1. Objetivos generales

El objetivo de este Trabajo Práctico es experimentar con sets de instrucciones SIMD de Intel ASM x86 sobre una serie de implementaciones de filtros en imágenes de mapas de bits (RGBA) con el propósito de entender mejor el rol del paralelismo de datos en este tipo de implementaciones y qué tan remunerador es. A partir de cada filtro, se plantearán implementaciones en C, SSE [1] (set de instrucciones sobre registros de 128 bits), AVX [2] (también sobre 128 bits pero permitiendo operaciones en formato 'no-destructivo' [3]) y su expansión de 256 bits AVX2. Acompañadas todas de un análisis para poder contrastar y discutir su rendimiento.

2. Contexto

hi

Titulo del parrafo Bla bla bla bla. Esto se muestra en la figura 1.

```
struct Pepe {

    ...

};
```

[1] Streaming SIMD Extensions
[2] Advanced Vector Extensions
[3] Preservan los operandos fuente

3. Cropflip

3.1. Descripción

La imagen destino del filtro consiste en invertir verticalmente (flip) un recorte (crop) de la imagen fuente a partir de offsets dados como parámetro. El ancho y alto en píxeles de la imagen destino también se pasa como parámetros. La descripción matemática está dada por la fórmula:

$$O^k_{i,\ j} = I^k_{tamy+offsety-i-1,\ \ offsetx+j}$$

Donde el 'crop' de la imagen input corresponde con los píxeles del tipo:

$$I^k_{offsety+i,\ \ offsetx+j} \qquad \text{con} \quad 0 \leq i < tamy \quad 0 \leq j < tamx$$

MEM

I_{30}	I_{31}	I_{32}	I_{33}	I_{34}	I_{35}
I_{24}	I_{25}	I_{26}	I_{27}	I_{28}	I_{29}
I_{18}	I_{19}	I_{20}	I_{21}	I_{22}	I_{23}
I_{12}	I_{13}	I_{14}	I_{15}	I_{16}	I_{17}
I_6	I_7	I_8	I_9	I_{10}	I_{11}
I_0	I_1	I_2	I_3	I_4	I_5

Cuadro 1: Ilustracion de la imagen fuente en memoria. En rojo los pixeles del crop (offsetx = 3, offsety = 2, tamx = 4, tamy = 4)

MEM

I_{15}	I_{16}	I_{17}
I_{21}	I_{22}	I_{23}
I_{27}	I_{28}	I_{29}
I_{33}	I_{34}	I_{35}

Cuadro 2: Ilustracion de la imagen destino en memoria

No es dificil notar que las filas del crop en la fuente no constituyen una tira contigua de píxeles en memoria, si no que se encuentran distanciadas por el tamaño en bytes de offsetx

3.2. Implementaciones

Al no involucrar operaciones aritméticas entre componentes de la imagen, las implementaciones del filtro se centran en accesos a memoria. Por lo tanto, las implementaciones solamente difieren en el modo en que se copian y asignan píxeles a la imagen output.

3.2.1. Implementaciones C y SSE

La implementación C simplemente se trata de recorrer la imagen con una sola variable aplicando pixel a pixel la transformación dada por la fórmula matemática previamente mencionada. Mientras que la implementación corresponiente a SSE recorre la imagen fuente desde la fila superior del recuadro del crop hacia abajo de a 4 píxeles por iteración como indica el siguiente pseudo-código:

```
temp = src + offsetx*4 +srcRowSize*(offsety+tamy-1)
{Apunta al primer píxel de la esquina superior izq. del crop}
for i = 0 to tamy:
  for j = 0 to tamx:
    xmm0 = [temp]
    [dst] = xmm0
    dst = dst  + 16
    temp =  temp + 16
  temp = temp - (dstRowSize + srcRowSize)
{Decrece el ancho de la imagen destino y el de la fuente para apuntar primer pixel de la fila de
 abajo a la que procesó}
  end for
end for
```

De esta manera, las lecturas y escrituras en memoria representan $\frac{1}{16} = 0,0625 = 6,25\,\%$ del total de los accesos a memoria de la implementación C.

3.2.2. Implementaciones SIMD paralelo en 128 y 256 bits

Se diferencian de la implementación anterior de SSE en el uso de la mayor cantidad posible de registros XMM (YMM de 256 bits en el caso de las implementaciones de AVX2) para las operaciones de transferencia de bloques de pixeles de la imagen fuente a la destino.

Dado que las lecturas y escrituras en memoria de cada registro son independientes entre sí, la ejecución fuera de orden del procesador hace que la transferencia por bloque sea más rápida que la implementación individual (que recorre la imagen con un único registro XMM). Por cuestiones de vecindad espacial de los píxeles en la memoria, los accesos a memoria de cada registro tienen chances particularmente altas de hitrate en caché, no demorando así las lecturas del resto de los registros.

Recordando que los registros de AVX2 tienen 32 bytes de capacidad y cada píxel en nuestro formato ocupa 4 bytes, cada registro YMM tendrá capacidad para:

$$\frac{32\ bytes}{4\ \frac{bytes}{px}} = 8\ px$$

Por lo cual, suponiendo $tamx \equiv 0\ (mod\ 8)$ (como es el caso de una imagen de salida de 512x512), tendríamos $\frac{1}{32} = 0,03125 = 3,125\,\%$ del total de accesos a memoria de la implementación en C.

3.2.3. Copiado paralelo de vectores

La asignación de píxeles se hace llamando a las funciones externas 'copyN_sse' y 'copyN_avx2'[4] que copian tiras de píxeles de una imagen a otra por bloques de 64/4/1 ó 128/8/1 píxeles (sse y avx2 respectivamente) según sea posible. A modo de ejemplo (los otros casos son análogos), para copiar bloques de 64px (256B) desde la posición indicada por rsi a la indicada por rdi se cargan los valores correspondientes de la siguiente manera:

xmm0 ← [rsi]
xmm1 ← [rsi+16]
...
xmm14 ← [rsi+224]
xmm15 ← [rsi+240]

[4] ../entregable/tp2-bundle.v1/codigo/lib

[rdi] ← xmm0
[rdi+16] ← xmm1
...
[rdi+224] ← xmm14
[rdi+240] ← xmm15

En el caso del filtro Cropflip, las tiras de píxeles corresponden a tiras de tamaño 'tamx' (es decir, al ancho del crop).

3.3. Rendimiento y análisis

A diferencia de los otros dos filtros, donde el procesamiento de imágenes involucra operaciones aritméticas paralelas, la implementación AVX2 de Cropflip no presenta un 'speedup' tan significativo respecto de las implementaciones de SSE con registros de 128 bits en paralelo.

4. LDR (Low Dynamic Range)

4.1. Descripción

El filtro LDR modifica las zonas mas brillantes de la imagen, realzando o disminuyendo su brillo según el parametro alpha sea positivo o negativo.

Formalmente,

$$O^k_{i,j} = ldr^k_{i,j} = I^k_{i,j} + \alpha \frac{sumargb_{i,j}}{max} I^k_{i,j} \qquad \forall k \in \{r,g,b\},\ i,j \in \mathbb{N} \text{ tq } 2 \leq i < w-2 \ \wedge\ 2 \leq j < h-2$$

donde

$$sumargb_{i,j} = \sum_{-2 \leq u,v \leq 2,\ k \in \{r,g,b\}} I^k_{i+u,j+v}$$

$$max = 5 * 5 * 255 * 3 * 255$$

y los $O^k_{i,j}$ no definidos mantienen su valor inicial.

Como para procesar cada pixel necesitamos el valor de sus vecinos no podremos aplicar el filtro en los bordes de la imagen, por lo que quedan sin modificar.

Inicialmente se nos ocurrieron tres algoritmos para realizar el filtro:

1. Iterar sobre cada pixel. Si está en el borde copiarlo directamente, sino calcular su sumargb leyendo los 25 vecinos y guardar los valores correspondientes al pixel.

   Este método tiene una implementación trivial, pero un rendimiento pésimo debido a que debe acceder 26 veces a memoria para procesar cada pixel y realiza cálculos repetidos de las sumas de cada pixel.

2. Recorrer cada pixel de la imagen calculando la suma de sus canales r, g, b, y guardar el resultado en una matriz auxiliar.

   Luego recorrer nuevamente los pixeles no-borde y realizar la función usando los valores precalculados.

   Con este método evitamos repetir los cálculos de la suma de los canales de cada pixel, pero implica usar $\mathcal{O}(ancho * altura)$ memoria adicional (como la suma de tres bytes siempre entra en un word y no necesitamos copiar el alpha, la matriz tendrá la mitad del tamaño de la imagen). Además agrega el overhead de los accesos a memoria y ocupa espacio extra en la caché.

3. Recorrer cada pixel excepto los de las filas inferiores y superiores calculando

   $$\sum_{-2 \leq v \leq 2,\ k \in \{r,g,b\}} I^k_{i,j+v}$$

   esto es, la suma de su columna ± 2. Mantener siempre los últimos cuatro resultados en memoria y al calcular el nuevo valor realizar la suma entre ellos y usarlos para procesar el pixel $(i-2, j)$, si no es un borde.

   De esta forma logramos evitar algunos cálculos repetidos pero mantenemos el uso de memoria extra constante.

De estos algoritmos decidimos implementar la primer opción en C, ya que es el metodo trivial contra el que queremos comparar.

Para nuestras implementaciones en assembler utilizamos el tercer método, ya que consideramos que era el que ofrecía mas potencial para paralelizar con instrucciones SSE.

4.2. Implementaciones

4.2.1. C

El código de C es bastante simple, recorre cada pixel de la imagen y:

- Si es un borde, lo copiamos directamente.
- Si no es un borde, recorremos los 25 vecinos acumulando la suma de sus componentes, y así obtenemos sumargb. Luego calculamos el resto de la fórmula y guardamos el valor final.

4.2.2. Asm - SSE

Para implementar el algoritmo (3) recorremos la imagen procesando de a 4 píxeles, manteniendo la suma de las 4 ultimas columnas y calculando 4 nuevas en cada loop.

	$I_{i-2,j+2}$	$I_{i-1,j+2}$	$I_{i,j+2}$	$I_{i+1,j+2}$	$I_{i+2,j+2}$	$I_{i+3,j+2}$	$I_{i+4,j+2}$	$I_{i+5,j+2}$
	$I_{i-2,j+1}$	$I_{i-1,j+1}$	$I_{i,j+1}$	$I_{i+1,j+1}$	$I_{i+2,j+1}$	$I_{i+3,j+1}$	$I_{i+4,j+1}$	$I_{i+5,j+1}$
MEMORIA	$I_{i-2,j}$	$I_{i-1,j}$	$I_{i,j}$	$I_{i+1,j}$	$I_{i+2,j}$	$I_{i+3,j}$	$I_{i+4,j}$	$I_{i+5,j}$
	$I_{i-2,j-1}$	$I_{i-1,j-1}$	$I_{i,j-1}$	$I_{i+1,j-1}$	$I_{i+2,j-1}$	$I_{i+3,j-1}$	$I_{i+4,j-1}$	$I_{i+5,j-1}$
	$I_{i-2,j-2}$	$I_{i-1,j-2}$	$I_{i,j-2}$	$I_{i+1,j-2}$	$I_{i+2,j-2}$	$I_{i+3,j-2}$	$I_{i+4,j-2}$	$I_{i+5,j-2}$

Cuadro 3: Ilustracion de la memoria en el ciclo de ldr. En gris los pixeles que queremos procesar, en verde las columnas de las cuales ya tenemos la suma guardada y en naranja las columnas que debemos calcular.

Obviaremos escribir las variables i y j en las siguientes ilustraciones para mantener la claridad.

El proceso del loop comienza con la suma de las columnas anteriores en $\texttt{XMM}_0$, guardadas como word ya que su valor máximo es $5 * 3 * 255 < 2^{16}$:

$\texttt{XMM}_0$	0	0	0	0	$sumC_1$	$sumC_0$	$sumC_{-1}$	$sumC_{-2}$

Como cada pixel ocupa 32 bytes podemos cargar los cuatro píxeles que vamos a necesitar de cada fila en 5 registros.
Luego descomprimimos cada componente a tamaño word (usando 5 registros mas) y realizamos las sumas de los componentes para obtener la suma de cada pixel, cuidandonos de borrar el alpha.

$\texttt{XMM}_N$	$sum_{5,v}$	$sum_{4,v}$	$sum_{3,v}$	$sum_{2,v}$	0	0	0	0

para $-2 \leq v \leq 2$ y $N = v + 3$.

A continuacion sumamos todos los pixeles de la columna entre sí y combinamos el resultado con $\texttt{XMM}_0$, obteniendo el vector de sumas.

$\texttt{XMM}_0$	$sumC_5$	$sumC_4$	$sumC_3$	$sumC_2$	$sumC_1$	$sumC_0$	$sumC_{-1}$	$sumC_{-2}$

Luego procedemos a calcular $sumargb$. Para ello copiamos el contenido de $\texttt{XMM}_0$ a $\texttt{XMM}_5$ y luego vamos shifteando $\texttt{XMM}_0$ de a una palabra por vez mientras sumamos su valor a $\texttt{XMM}_5$ hasta que en $\texttt{XMM}_0$ queden solo las 4 sumas que acabamos de calcular y en la parte baja de $\texttt{XMM}_5$ se encuentre la suma de las 5 columnas vecinas (esto es, ya tenemos $sumargb$).
Observar que en $\texttt{XMM}_0$ ya quedaron los valores de las columnas listas para el siguiente loop.

$\texttt{XMM}_0$	0	0	0	0	$sumC_5$	$sumC_4$	$sumC_3$	$sumC_2$

$\texttt{XMM}_5$	X	X	X	X	$sumrgb_3$	$sumrgb_2$	$sumrgb_1$	$sumrgb_0$

Ahora convertimos cada $sumargb$ a punto flotante, la multiplicamos por el valor de α que habíamos almacenade en $\texttt{MM}_2$ y movemos replicamos cada una en un registro diferente.

$\texttt{XMM}_N$	$sumargb_u * \alpha$	$sumargb_u * \alpha$	$sumargb_u * \alpha$	$sumargb_u * \alpha$

para $0 \leq u \leq 3$ y $N = u + 5$

Mientras tanto cargamos los valores originales de los pixeles a procesar y los convertimos a punto flotante, ocupando cada pixel un registro entero.

$\texttt{XMM}_M$	A_u	R_u	G_u	B_u

para $0 \leq u \leq 3$ y $M = u + 9$

Multiplicamos las $sumargb$ anteriores por los canales de los píxeles y limpiamos la correspondiente al canal alpha con una máscara que nos armamos en el momento.

$\texttt{XMM}_N$	0	$R_u * sumargb_u * \alpha$	$G_u * sumargb_u * \alpha$	$B_u * sumargb_u * \alpha$

para $0 \leq u \leq 3$ y $N = u + 5$

Finalmente multiplicamos estos registros por el recíproco de MAX que teníamos previamente guardado y los sumamos a los registros con los valores de los píxeles.

$\texttt{XMM}_M$	A_u	ldr_u^r	ldr_u^g	ldr_u^b

para $0 \leq u \leq 3$ y $M = u + 9$

Solo resta aplicar max y min para saturar los valores calculados, reconvertirlos a byte y guardarlos en la imagen de salida.

.
.
.
.

Como la imagen que procesamos nunca tienen padding en la linea (ya que los píxeles ocupan 32b cada uno) podemos considerarla aplanada como una secuencia lineal de filas concatenadas.
De esta manera evitamos el uso de condicionales dentro del loop que tomen los bordes de las filas como casos especiales.
Procesar de este modo genera que en los bordes laterales se grabe basura, por lo que luego del loop debemos copiar los píxeles originales. Del mismo modo copiamos las dos filas superiores e inferiores que no fueron procesadas.

4.2.3. Asm - SSE con calculos de enteros

[1]

4.2.4. Asm - AVX/FMA

4.2.5. Asm - AVX2

4.3. Experimentos

5. Sepia

5.1. Descripción

5.2. Implementaciones

5.3. Experimentos

6. Conclusiones y trabajo futuro

Referencias

[1] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Professional, 2002.