



DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Programacion SIMD

Organización del Computador II
Segundo Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Leandro Raffo		
Maximiliano Fernández Wortman		
Uriel Rozenberg		



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>

Índice

1. Introduccion	3
2. Implementacion	3
2.1. Diferencia	3
2.2. Blur gaussiano	4
3. Resultados	5
3.1. Diferencia	5
3.2. Blur	7
4. Conclusión	8

1. Introduccion

En este trabajo práctico realizamos la implementación de dos filtros de imagenes, con tal de ver que tan eficiente puede llegar a ser (o no) SIMD, los filtros son la diferencia de imagenes y el blur gaussiano, los cuales fueron implementados en lenguaje C (gcc y clang) y assembly, haciendo uso de instrucciones vectoriales. Luego comparamos la performance de estas implementaciones sobre diferentes imagenes y usando herramientas probabilísticas y estadísticas.

2. Implementacion

2.1. Diferencia

Descripción de un ciclo de la iteración del filtro diferencia.

Primero pedimos memoria para declarar las máscaras que vamos a usar y armamos el stackframe (omitido)

```
section .rodata
mask_5 DB 2,2,2,2,6,6,6,6,10,10,10,10,14,14,14,14
trans_2 DB 0,0,0,0,255,0,0,0,255,0,0,0,255,0,0,0,255
```

Luego de armar el stackframe tenemos

```
mov r12, rdx
mov eax, r8d
mov ecx, ecx
mul rcx
xor r15, r15
```

r12 apunta a la matriz resultado, ecx tiene la cantidad de filas, y la parte baja de rax(eax) tiene la cantidad de columnas. Al hacer mul rcx se tiene filas*columnas en rdx:rax, como movimos cosas de 32 bits obtenemos la multiplicación en rax, que es lo que vamos a usar, junto a r15 para iterar (notar que tenemos en cuenta para esto que los movimientos de registros entre dwords limpian la parte alta de los registros de 64bits, es decir extendemos el número usando que son enteros positivos). Ahora entramos al ciclo.

```
.ciclo:
    cmp r15, rax
    JE .fin
```

Comparamos si r15 es igual a rax en tal caso ya hicimos la diferencia sobre todos los pixeles y termina el ciclo. El ciclo sigue con

```
movdqu xmm3 , [RDI + r15*4]
movdqu xmm15, [RSI + r15*4]
movdqu xmm14, xmm15
pminub xmm15, xmm3
pmaxub xmm3 , xmm14
psubb xmm3 , xmm15
movdqu xmm4, xmm3
movdqu xmm5, xmm3
```

Movemos a xmm3 los primeros 4 pixeles de la primera matriz y a xmm15 los primeros 4 de la segunda matriz a comparar, estos ocupan respectivamente 16 bytes en memoria (brga por 4). Después Guardamos en xmm14 el valor de xmm15 y hacemos un pminub entre xmm15 y xmm3 lo cual deja en xmm15 el mínimo byte a byte. Lo mismo para xmm3 pero con pmaxub es decir este tiene el máximo byte a byte. Hacemos esto porque queremos calcular el valor absoluto de la forma $|x - y| = \max(x, y) - \min(x, y)$. Concluimos esta idea haciendo psubb entre xmm3 que tenia el máximo y xmm15 que tenia el mínimo y finalmente guardamos el resultado en xmm4,5 para operar en la siguiente parte.

```
pslldq xmm4, 1
pslldq xmm5, 2
movdqu xmm6, xmm5
pmaxub xmm6, xmm4
pmaxub xmm6, xmm3
pshufb xmm6, [mask_5]
paddsb xmm6, [trans_2]
movdqu [r12 + r15*4], xmm6
add r15d, 4
jmp .ciclo
```

Ahora shifteamos con packed shift `xmm4, 5` uno y dos bytes respectivamente de forma de poder tomar el máximo de entre `r g b` en paralelo, es decir 4 pixels a la vez. Por ejemplo, el primer byte de `xmm4` tiene al byte de `r`, y el de `xmm6` tiene al byte de `g`, de forma que al hacer `pmaxub` entre `xmm6` y `xmm4` nos deja en el primer byte de `xmm6` (y cada 3 bytes) $\max(r_n, g_n)$ donde $n = \{1, 2, 3, 4\}$ indican los pixeles que levantamos. Los demas bytes de este registro no nos interesan. Repetimos esto entre `xmm6` y `xmm3`, pasa de vuelta lo mismo pero ahora tenemos en el primer byte de `xmm6` (y cada 3 bytes) $\max(r_n, g_n, b_n)$ con $n = \{1, 2, 3, 4\}$. Ahora tenemos que mover este máximo a las 3 coordenadas `r, g` y `b`, hacemos esto mediante la mascara `mask_5` y mediante `trans_2` sumamos con saturación con tal de dejar en `alpha` el valor 255. Copiamos los 16 bytes correspondientes (con el offset adecuado) en la matriz destino, sumamos 4 a `r15d` y saltamos para, si es necesario, volver a hacer el ciclo completo.

2.2. Blur gaussiano

En el blur gaussiano calculamos 4 pixeles por iteración, pero para legibilidad solo vamos a escribir lo que le pasa a uno solo, se repite el código y se apunta a memoria correctamente.

```
movdqu xmm0, [r12]
movdqu xmm1, xmm0
punpckhbw xmm0, xmm7
punpcklbw xmm1, xmm7
movdqu xmm2, xmm0
movdqu xmm3, xmm1
punpckhwd xmm0, xmm7
punpcklwd xmm2, xmm7
punpckhwd xmm1, xmm7
punpcklwd xmm3, xmm7
```

Primero levantamos 16 bytes de memoria, equivalente a 4 pixeles en `xmm0` y lo copiamos en `xmm1` ya que vamos a desempaquetar los datos. Llamamos a `punpcklbw/hbw` junto a un `xmm7` que estaba en 0, tenemos entonces que `xmm0` tiene los primeros 2 pixeles (8 bytes) extendidos a cada uno a un word. Lo mismo para `xmm1`. Luego repetimos el proceso desempaquetando a un double word, ya que queremos operar con floats y este es su tamaño.

```
cvtdq2ps xmm0, xmm0
```

3. Resultados

3.1. Diferencia

Nota: todos los tests fueron corridos sobre una PC con procesador Intel Core2Duo E8440.

Para analizar las implementaciones de C y assembly, corrimos 1000 iteraciones de cada implementación de diferencia sobre una imagen de 2308x2308 (16mb) donde las implementaciones de C se compilaron con gcc y clang usando los flags de optimización -O0, -O1, -O2 y -O3.

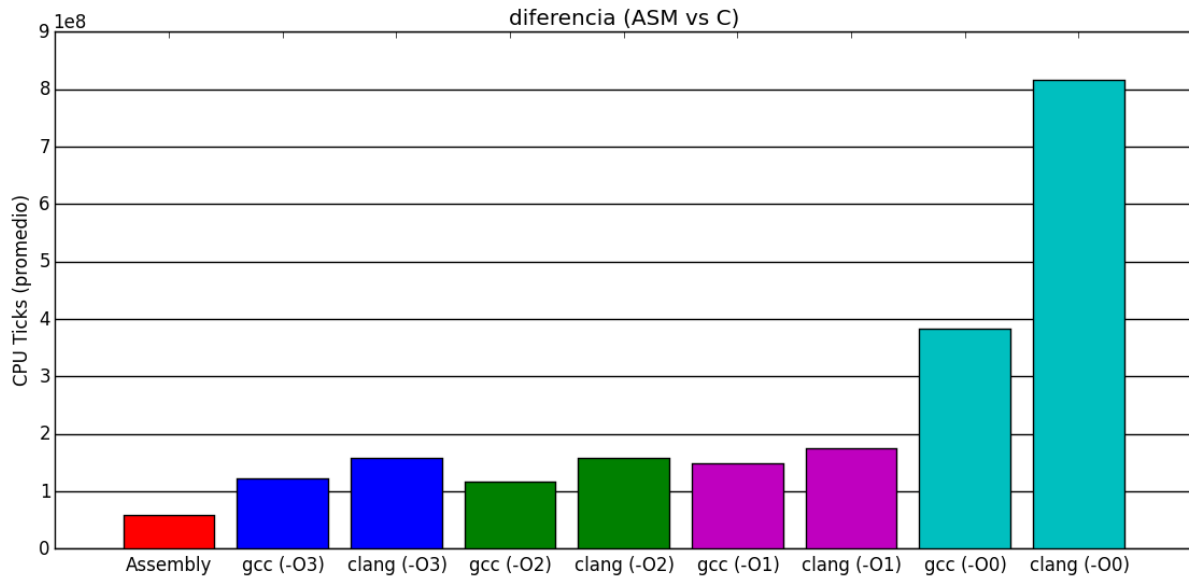


Figura 1: Promedio, sin outliers incluidos, sobre una imagen de 2308x2308 de 16mb.

Luego eliminamos los outliers y graficamos de vuelta obteniendo.

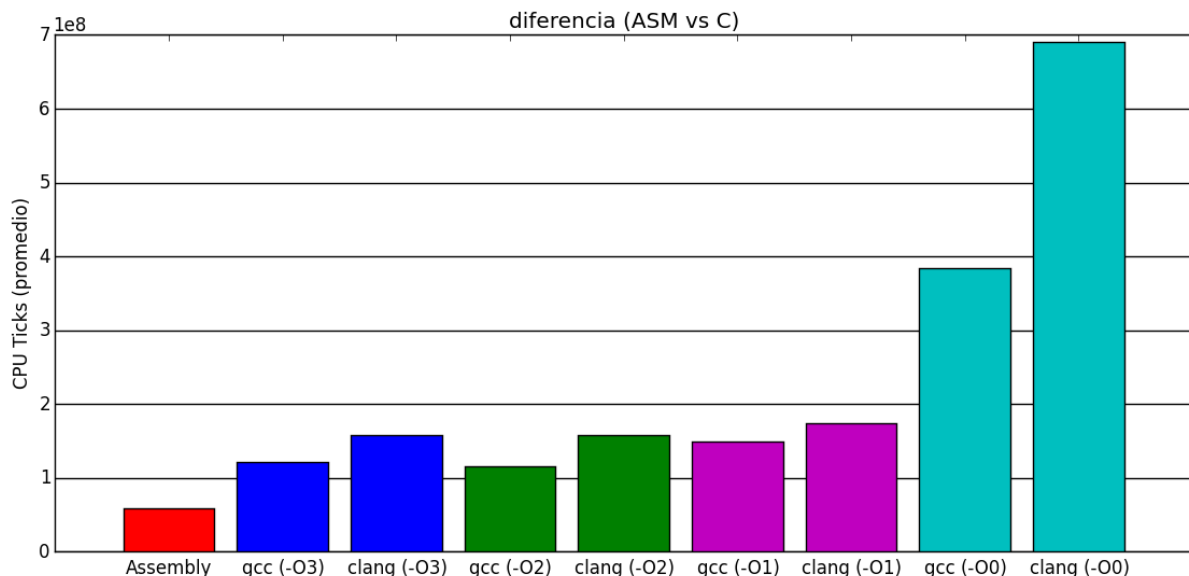


Figura 2: Promedio, con outliers incluidos, sobre una imagen de 2308x2308 de 16mb.

Vemos que clang en -O0 se comporta un poco mejor sacando esos resultados extremos, y lo que podemos concluir es que el algoritmo implementado en assembly con SIMD corre 2 veces más rápido que su contraparte en C.

En el siguiente experimento quería ver si el tamaño de la imagen iba a modificar la performance (incrementar el tiempo de ejecución) del algoritmo debido a cache misses. Para esto corrimos diferencia en Assembly y C, de vuelta bajo gcc y clang, pero esta vez con el flag -O2, ya que me pareció que era el sweet spot para esta implementación ya que los flags de optimización no garantizan que corra con mejor performance, sobre imágenes que iban de 256kb a 64mb como se ve en la figura (Los tamaños utilizados están en el shellscript convertir.sh que es el que usamos para generar las imágenes).

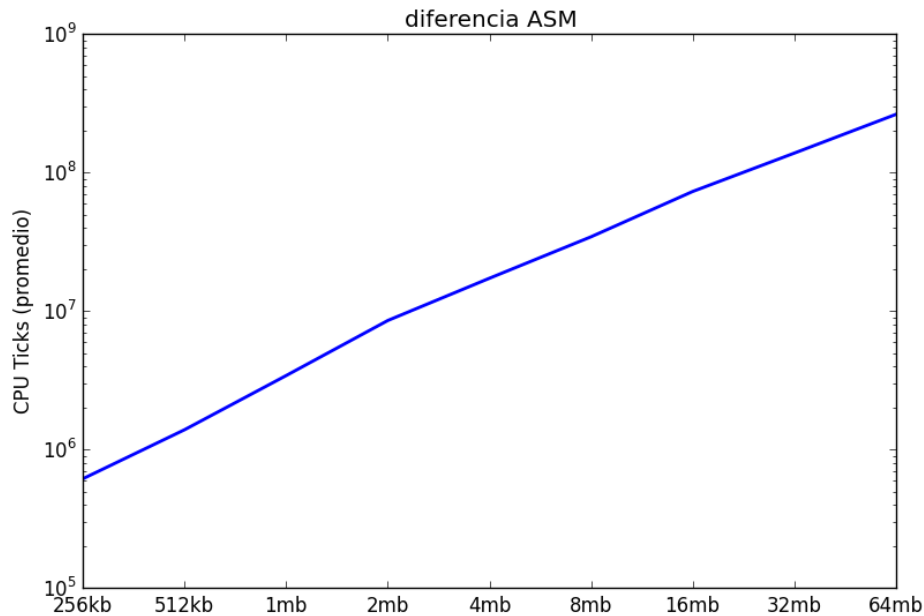


Figura 3: Diferencia en asm, escala logarítmica

Lo que se obtuvo es una curva que sube suavemente, lo cual implicaría que el algoritmo tiene un crecimiento bastante predecible hasta el tamaño donde se lo probó (imágenes de 4000x4000 pixels con un tamaño de 64mb).

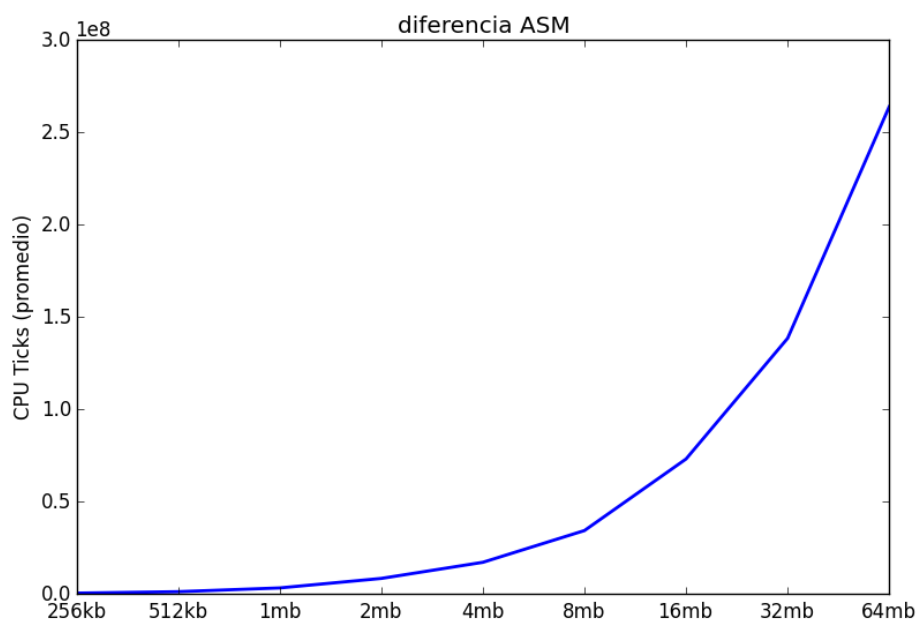


Figura 4: Diferencia en asm, escala no logarítmica

Lo mismo pasa con las implementaciones en C, en este caso gcc con -O2.

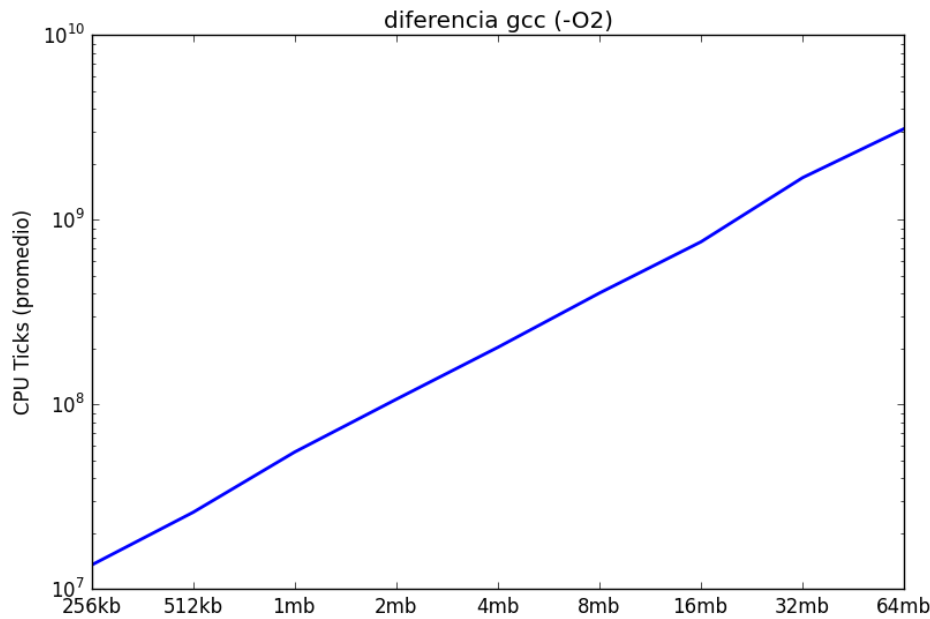


Figura 5: Diferencia en C, escala logarítmica

3.2. Blur

Una de las hipótesis que teníamos con blur es que dada una imagen con un radio pequeño iba a correr mas lento que con un radio mas grande, pero a medida que el radio domine la cantidad de pixeles sobre las cuales va a aplicar la matriz de convolución el tiempo de ejecución iba a bajar. Esto lo pudimos probar corriendo blur sobre una imagen de tamaño fijo (en este caso 584x584 pixels) e incrementando el radio como se ve en la siguiente imagen

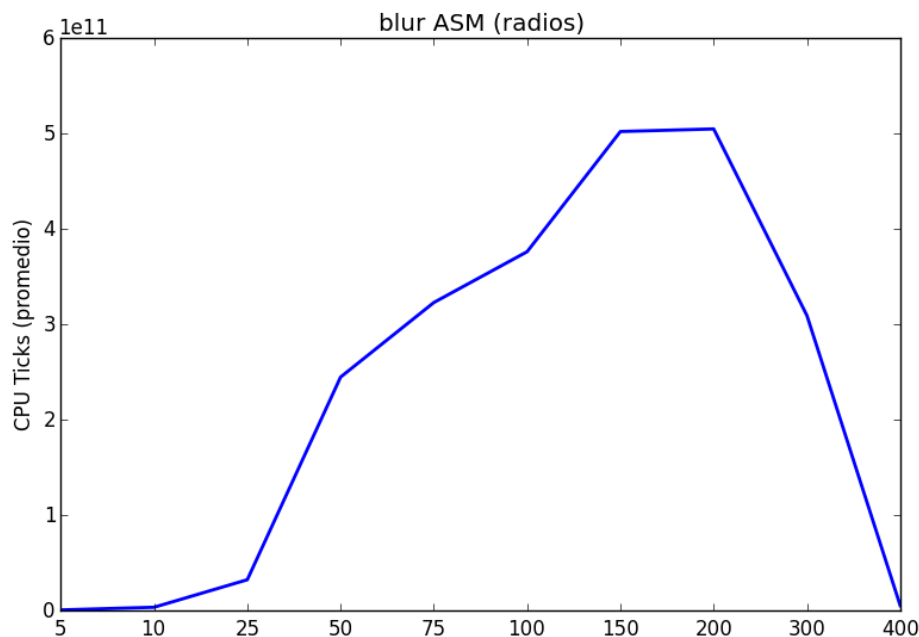
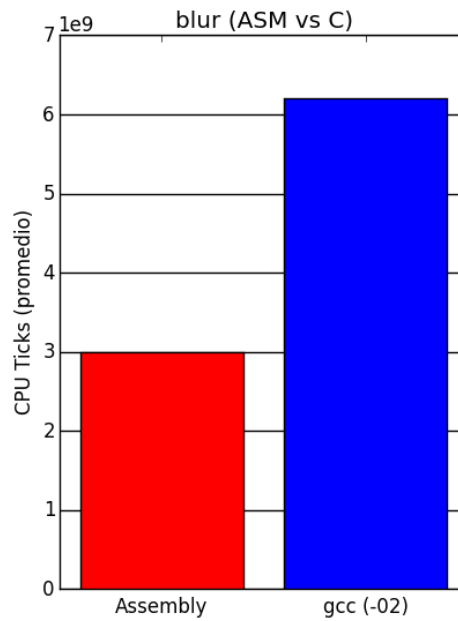


Figura 6: Blur, cambiando los radios

El algoritmo llega a un punto crítico cuando el radio es equivalente a la dimensión/2, es decir cuando el algoritmo se comporta como $O(n^4)$ sobre la dimensión.

Para comparar al algoritmo en C vs asm hicimos lo mismo que blur, corrimos los dos sobre la misma imagen y calculamos su promedio.



4. Conclusión

Para el algoritmo de diferencia, se justificaria totalmente hacer una implementación en SIMD, no solo por el hecho de que corre mas rápido, sino que además implementarlo en assembly fue bastante fácil (< 20 lineas). Pasa lo contrario con blur, donde la performance ganada no es justificable contra el costo de implementarlo.