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

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

Descripcíon de un ciclo de la iteración del filtro diferencia.
Primermo pedimos memoria para declarar las máscaras que vamos a usar y armamos el stackframe (omitido)

```
section .rodata
mascara_copia_pixels db 2,2,2,2,6,6,6,6,10,10,10,10,14,14,14,14
mascara_asigna_alpha db 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 xmm$4, 5$ para operar en la siguiente parte.

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

Ahora shifteamos con packed shift xmm$4,5$ 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

El blur gaussiano consta de dos ciclos uno anidado sobre el otro, el anidado itera sobre el kernel y el que anida itera sobre la matriz, usamos SIMD para hacer los calculos sobre cuatro pixeles en simultaneo (es decir sobre 16 componentes en total), además en vez de hacer offsets para atras como el codigo en C nos movemos siempre para adelante, por ejemplo, empezamos en el pixel 0 y recorremos la submatriz que va de $0, 2r$ en filas y $0, 2r$ en columnas, donde r es el radio, a la par de la matriz del kernel (que también tiene este tamaño) haciendo las multiplicaciones sobre cuatro pixeles y acumulandolas, luego cuando terminamos de iterar sobre el kernel, insertamos en los cuatro pixel destino, que esta en la posición (r, r) hasta $(r+3, r+3)$, luego aumentamos la columna y seguimos este proceso hasta el ancho $-2r-16$ donde nos paramos en la proxima fila y repetimos hasta llegar a la altura $-2r$, es decir siempre que insertamos los pixeles estamos en rango. Podemos hacer 4 pixeles a la vez pues el ancho es un múltiplo de 4. Ahora pasamos a describir más en detalle las operaciones que usamos para esto.

Nota: Siempre vamos a poner el código sobre un pixel porque las operaciones sobre los otros 3 son las mismas operaciones sobre distintos registros en seguidilla, vamos a aclarar estos casos en la explicación.

Antes de entrar al ciclo preparamos algunos registros para saber hasta donde iterar, tenemos que: rax guarda el último valor hasta el que vamos a recorrer que es igual a [(filas $-2r$)*columnas] $-2r$, rbp va a guardar hasta donde vamos a hacer cada subciclo es decir $(2r+1)^2$, r11 tiene el valor $2r$, r12 la matriz fuente, r13 la matriz destino, r8 la cantidad de columnas y rbx (ebx) el offset necesario para insertar el pixel en la matriz destino.

```
.ciclo_kernel:
    cmp rcx, rbp
    je .insert
    mov r9, rsi
    add r9, rdi
    movd        xmm0, [ r12 + r9 ]
    punpcklbw   xmm0, xmm8
    punpcklwd   xmm0, xmm8
    cvtdq2ps    xmm0, xmm0
```

Primero verificamos si tenemos que insertar el pixel, para esto comparamos rcx, el registro que cuenta sobre el tamaño del kernel contra rbp que mantenia hasta donde teniamos que iterar para haber recorrido el kernel, si no, levantamos de memoria 4 bytes y los desempaquetamos a double words y convertimos a float, repetimos esto con los offsets $r12 + r9 + 4/8/12$, es decir los primeros 16 bytes, y guardando en los registros xmm4, xmm5 y xmm6 respectivamente.

```
    movd        xmm1, [ r10 + rcx*4 ]
    movq        xmm3, xmm1
    pslldq      xmm3, 4
    paddd       xmm3, xmm1
    pslldq      xmm3, 4
    paddd       xmm3, xmm1
    mulps       xmm0, xmm3
    addps       xmm2, xmm0
```

Como rcx itera el kernel linealmente, movemos el valor de la matriz de convolución apuntado por rcx con movd (un float por eso el $*4$) a xmm1, shifteamos 4 bytes con pslldq, ya que queremos propagar este valor para que multiplique las 3 componentes (rgb) a la vez, movemos un quadword shifteamos y hacemos un packed add (queremos copiar el valor del float) shifteamos xmm3 y volvemos a sumar una vez más, para que cada doubleword de xmm3 tenga el flaot. Luego multiplicamos el registro con el valor del pixel del kernel, contra los valores del pixel que desempaquetamos en xmm0 dejando en xmm0 este valor y lo acumulamos en xmm2. De vuelta repetimos esto con $xmm4/5/6$ y acumulamos en $xmm7/9/10$

```
    cmp rdx, r11
    je .sumar_fila_kernel
    add rdx, pixel_size
    inc rcx
    add rsi, pixel_size
    jmp .ciclo_kernel
```

rdx mantiene el contador para iterar sobre la fila, si este es igual a 2r saltamos a sumar_fila_kernel de lo contrario adelantamos rdx y rsi en pixel_size (que es un %define con el valor 4) y entramos al ciclo de vuelta.

```
.sumar_fila_kernel:
    xor rdx, rdx
    sub rsi, r11
    add rsi, r8
    inc rcx
    jmp .ciclo_kernel
```

En sumar fila limpiamos rdx, para volver a poder iterar de 0 a $2r$, le restamos $2r$ a rsi, este iteraba sobre la fila para la matriz mas grande y le sumamos al mismo la cantidad de columnas, es decir rsi apunta al comienzo de la fila siguiente, incrementamos rcx asi en el proximo ciclo apunta a la siguiente fila tambien y saltamos al ciclo de vuelta.

```
.insert:
   pxor     xmm4, xmm4
   cvtps2dq xmm2, xmm2
   packusdw xmm2, xmm4
   packuswb xmm2, xmm4
   movd     xmm7, ebx
   add      rbx, rdi
   movd     [ r13 + rbx ], xmm2
   ..
   movd  [ r13 + rbx + 4], xmm7
   ..
   movd     ebx, xmm7
   pxor     xmm7, xmm7
   xor rsi, rsi
   xor rcx, rcx
   cmp rdi, r15
   je .sumar_fila
   add rdi, pixel_size
   jmp .ciclo_matriz
```

En .insert hacemos casi lo contrario a levantar, convertimos los floats empaquetados que estaban en xmm2 a doublewords empaquetados y re-empaquetamos a word y luego a byte. Luego sumamos rbx e rdi para obtener la posición en la que vamos a insertar el pixel (los 4 bytes) e insertamos el pixel, de vuelta repitiendo para los 4 pixeles con el offset $+4/8/12$. Una vez que esto pasa ponemos rsi y rcx en 0 y vemos si llegamos a la ultima columna que teniamos que iterar, si pasa esto saltamos a sumar_fila sino sumamos 4*pixel_size a rdi y saltamos al ciclo del kernel de vuelta.

```
.sumar_fila:
   add rdi, r11
   add r15, r8
   jmp .ciclo_matriz
```

En .sumar_fila lo que hacemos es sumarle a rdi lo que faltaba para llegar a la proxima columna, $2r$ y le sumamos a r15 la cantida de columnas, r15 en este caso mantiene (acorde al rdi) hasta que columna tenemos que iterar. Basicamente r15 empieza en cols $-2r$ y se le van sumando el total de columnas, en la segunda iteracion tiene 2cols $-2r$ etc. Aca saltamos a ciclo_matriz y no ciclo_kernel, este contiene lo siguiente.

```
.ciclo_matriz:
   cmp rdi, rax
   jg .end
```

Si rdi es mayor que rax (que marcaba el ultimo pixel sobre el que podiamos iterar) saltamos al final, desarmamos el stackframe y retornamos.

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, -O3 y -Os. Basicamente los flags de -O# con # un número entre $0, 3$ intentan optimizar principalmente para velocidad y luego tamaño, 0 siendo el menos 'optimizado' y 3 el más optimizado, mientras que -Os optimiza principalmente por tamaño del ejecutable, por ejemplo, luego de hacer un disassembly, notamos que a partir de -O2 gcc ya no armaba un stackframe, ni resguardaba rbp, y pusheaba a medida que necesitaba los registros, alineando rsp cuando fuese necesario usar la pila, diferencias observables entre -O2 y -O3 fue el uso de operaciones SIMD, por ejemplo de empaquetado y desempaquetado, multiplicaciones packed etc en -O3 (para el blur), mientras que esto no pasaba en -O2, extrañamente la performance es la misma y tal vez se debe a otros factores como el modelo de procesador (Intel c2d E840) o que en general gcc hace muchos mas accesos a memoria y estos hacen sombra a la performance ganada por las instrucciones.

A continuación dejamos los gráfico con el promedio (con y sin outliers) de la diferencia bajo distintos flags de optimización y en assembly.

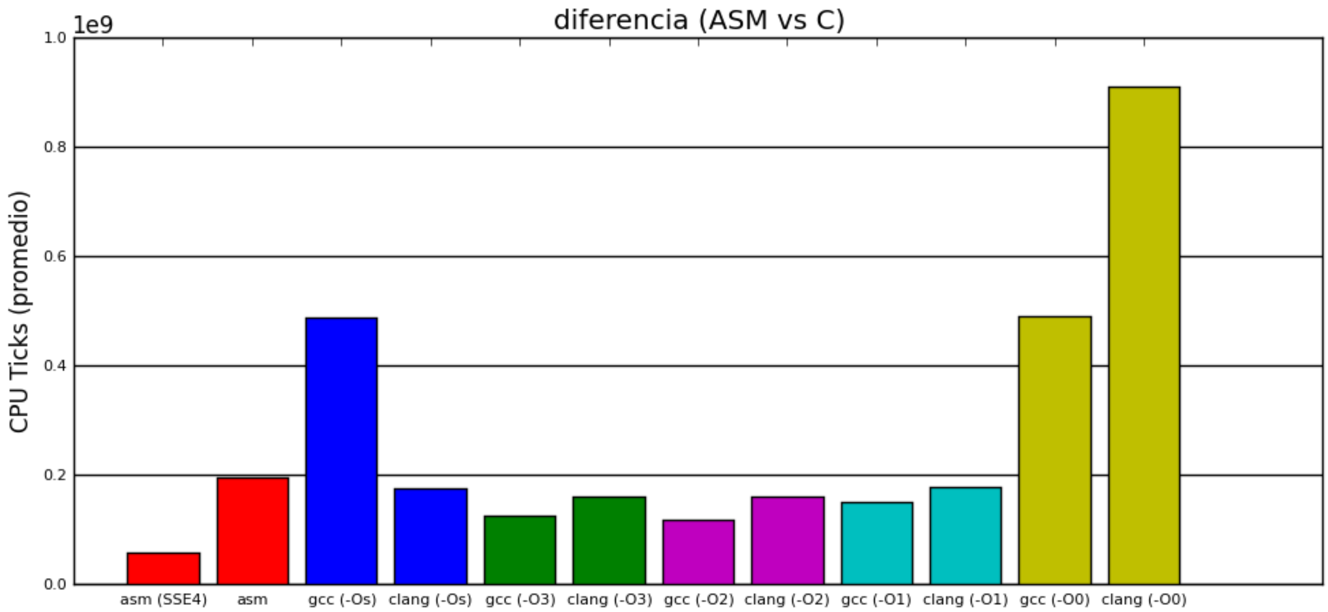


Figura 1: Promedio, 1000 iteraciones sobre una imagen de 2308x2308 pixels de 16mb.

Como podemos observar, la versión en assembler usando instrucciones vectorizadas es más de dos veces mas rapida que cualquiera de las otras implementaciones. En particular es . Otra cosa a observar es que la versión en assembler usando registros de propósito general es más lenta que cualquiera de las versiones -O3 y -O2 de los compiladores, leyendo el manual de optimización de intel vimos en la sección 3.5.1.5 y citamos: .Avoid ROTATE by register or ROTATE by immediate instructions. If possible, replace with a ROTATE by 1 instruction.Ïnteresantemente al reemplazar los ror rax, 8 por ror rax, 1 y desplegar en 8 instrucciones (4 veces se hacia cada ror por lo tanto 24 instrucciones) vimos drásticamente afectada la performance del algoritmo, pero para mal, es decir corre más lento que cualquiera de las otras implementaciones. Dejamos a continuación los valores númericos del promedio la desviación, la mediana y la esperanza.

Tipo	**Promedio**	**Esperanza**	**Desviación**	**Mediana**
asm (SSE)	1.152.187	1	1	1
asm v1	1.151.544	1	1	1
asm v2	1.151.544	1	1	1
asm v3	1.151.544	1	1	1
gcc -Os	761.937	1	1	1
gcc -O3	1.152.847	1	1	1
gcc -O2	1.152.765	1	1	1
gcc -O1	725.420	1	1	1
gcc -O0	123	1	1	1
clang -Os	649.248	1	1	1
clang -O3	1.153.061	1	1	1
clang -O2	1.152.798	1	1	1
clang -O1	1.155.718	1	1	1
clang -O0	123	1	1	1

En el siguiente experimento queria ver si el tamaño de la imagen iba a modificar la performance (incrementar el tiempo de ejecucion) 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 parecio 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 imagenes).

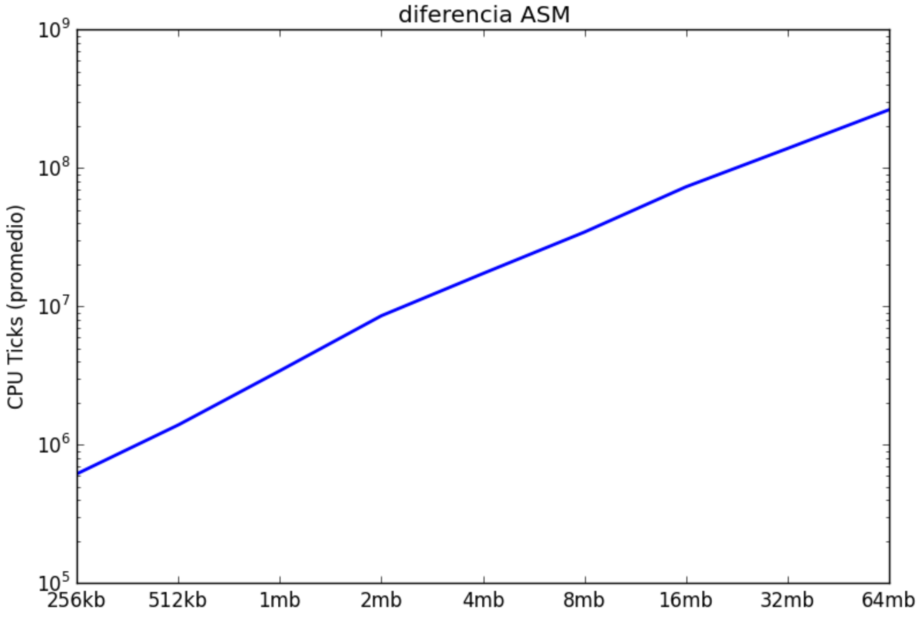


Figura 2: Diferencia en asm, escala logarítmica

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

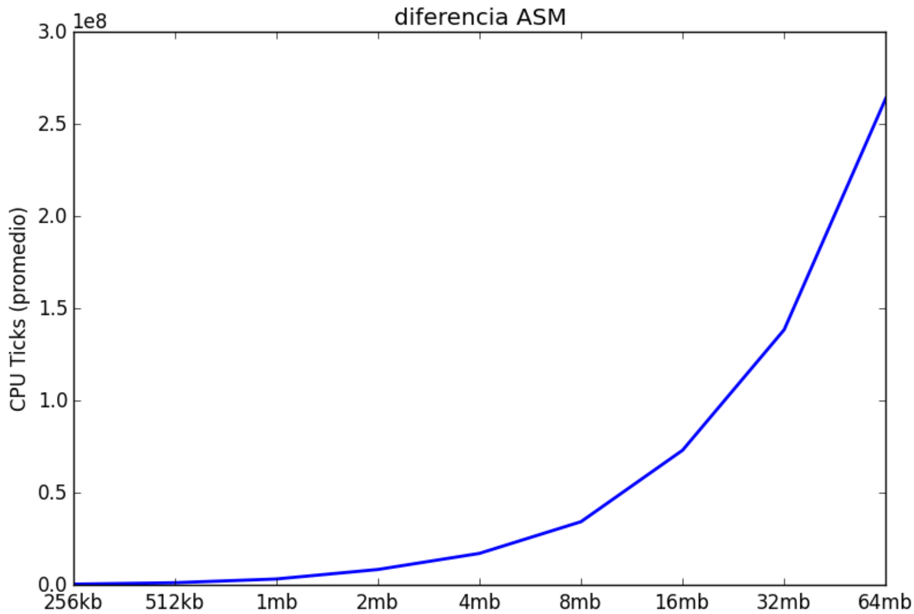


Figura 3: Diferencia en asm, escala no logarítmica

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

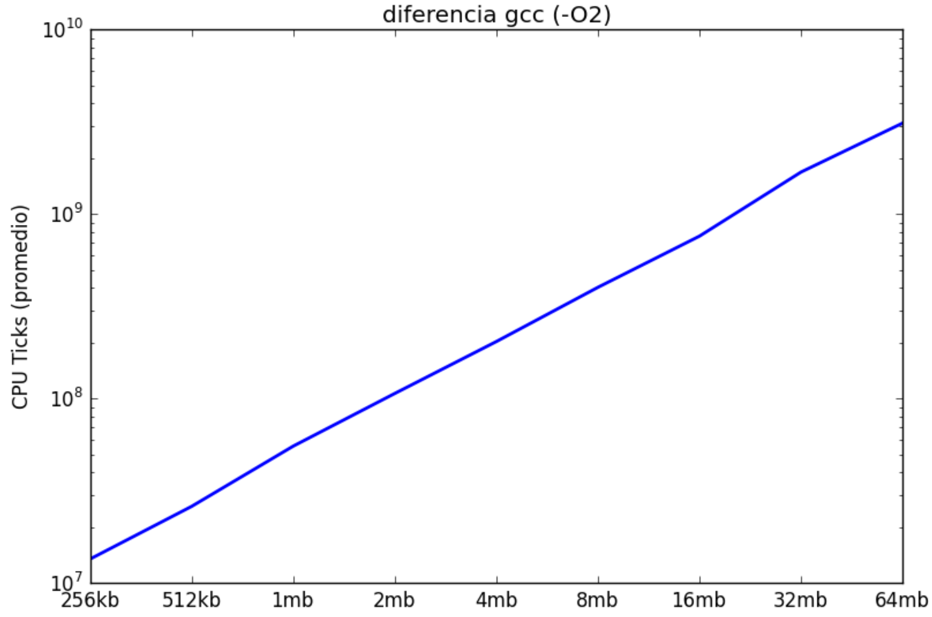


Figura 4: Diferencia en C, escala logarítmica

3.2. Blur

Primero corrimos un tests parecido al de diferencia donde probamos todas las implementaciones de blur bajo distintos compiladores y tambien corrimos una versión vectorizada pero que solo trataba un pixel a la vez (sus 4 componentes) esta es llamada v1 y la que operaba 4 pixels (16 bytes) es llamada v2. Todo esto sobre una imagen de 1160x1160, radio $= 15$ y 100 iteraciones.

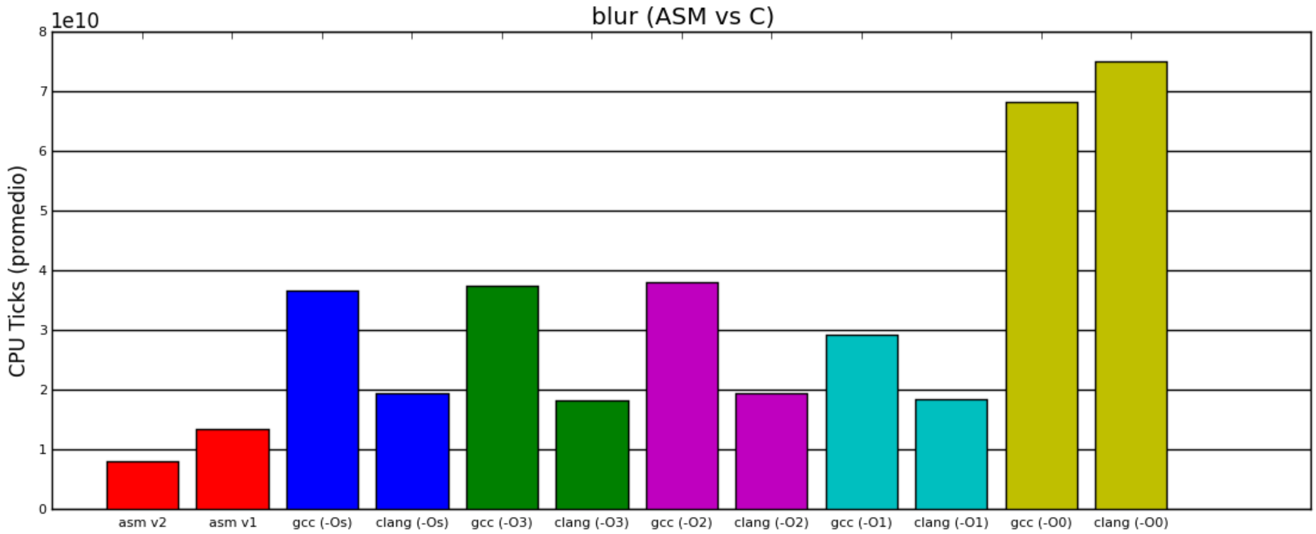


Figura 5: Promedio sobre 100 iteraciones del blur aplicado a una imagen de 1160x1160 con radio 15.

Como vemos la versión de assembler que opera sobre 4 pixels es más de 3 veces mas rápida que cualquiera de las versiones de C, y es casi dos veces mas rapida que la versión que opera sobre un pixel, el cambio de codigo es mínimo, asi que no se justificaria nunca el no pasar a operar sobre 16 pixeles. Otra cosa a notar es que el rol entre gcc y clang cambió a diferencia de la diferencia de imagenes donde gcc era siempre mas rápido que clang aca pasa lo contrario, e inclusive la versión de clang en -O1 es la más rápida en general, esto también nos hace pensar que los flags de optimización no siempre son una solución efectiva si lo que se busca es codigo de alta performance, para esto la solución siempre va a ser escribir puramente codigo assembler. No solo el código es más fiel (la desviación sobre assembler es la más pequeña de todas) sino que más rápido en todos los casos.

Tipo	Promedio	Esperanza	Desviación	Mediana
asm v2	1.152.187	1	1	1
asm v1	1.151.544	1	1	1
gcc -Os	761.937	1	1	1
gcc -O3	1.152.847	1	1	1
gcc -O2	1.152.765	1	1	1
gcc -O1	725.420	1	1	1
gcc -O0	123	1	1	1
clang -Os	649.248	1	1	1
clang -O3	1.153.061	1	1	1
clang -O2	1.152.798	1	1	1
clang -O1	1.155.718	1	1	1
clang -O0	123	1	1	1

Una de las hipótesis que teniamos 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

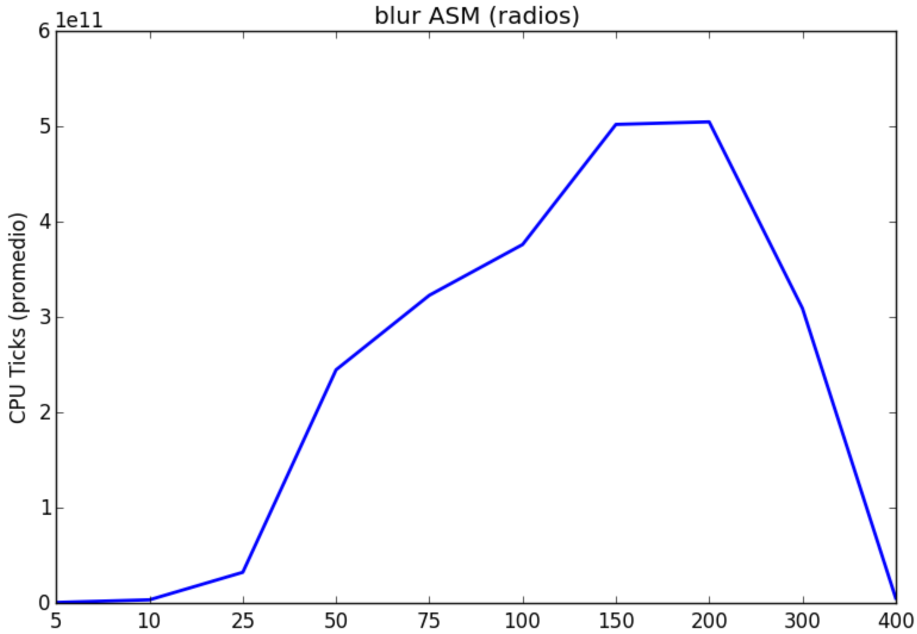
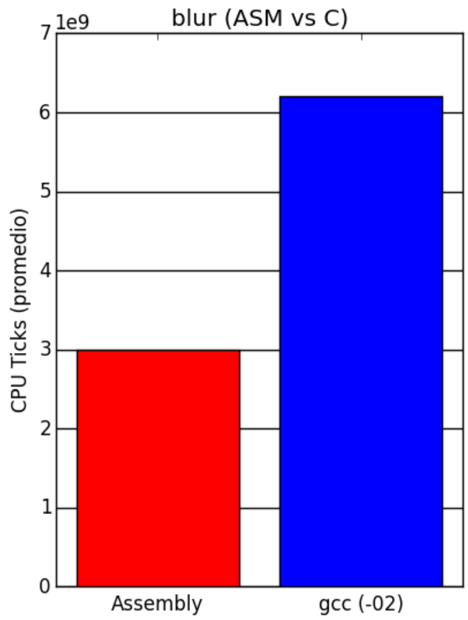


Figura 6: Blur, cambiando los radios

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).