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
Nombre	XXX/XX	`mail`
Nombre	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 procesar información de manera eficiente cuando los mismos requieren:

1. Transferir grandes volumenes 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 suceden al utilizar instrucciones SIMD en comparación con código C con diversos grados de optimización realizados por el compilador.

Para ello se realizarán cuatro filtros de fotos, Cropflip, Bandas, Sierpinski y Motion Blur, tanto en código assembler, que aproveche las instrucciones SSE brindadas para los procesadores de arquitectura Intel, como en código C, al que se le aplicarán los distintos flags de optimización -O0 (predeterminado), -O1, -O2 y -O3.

El primer filtro, Cropflip, se utilizará para mostrar cuanto mejora la performance al utilizar los registros XMM para transferir grandes cantidades de información.

El segundo, tercer y cuarto filtro, se centrarán en la variación de performance (en comparación al código en C) al utilizar instrucciones SIMD, no sólo para transferir grandes volúmenes de datos sino también para procesarlos en forma paralela, es decir, realizar diversos cálculos (sumas, multiplicaciones, divisiones) tanto en representación de enteros como punto flotante.

2. Enunciado y solución

2.1. Enunciado

2.2. Filtro cropflip

Programar el filtro *cropflip* en lenguaje C y luego en ASM haciendo uso de las instrucciones vectoriales (**SSE**).

Experimento 1.1 - análisis el código generado

En este experimento vamos a utilizar la herramienta `objdump` para verificar como el compilador de C deja ensamblado el código C.

Ejecutar

```
objdump -Mintel -D cropflip_c.o
```

¿Cómo es el código generado? Indicar *a*) Por qué cree que hay otras funciones además de `cropflip_c` *b*) Cómo se manipulan las variables locales *c*) Si le parece que ese código generado podría optimizarse

Experimento 1.2 - optimizaciones del compilador

Compile el código de C con flags de optimización. Por ejemplo, pasando el flag `-O1`[1]. Indicar 1. Qué optimizaciones observa que realizó el compilador 2. Qué otros flags de optimización brinda el compilador 3. Los nombres de tres optimizaciones que realizan los compiladores.

Luego de optimizar el codigo, se observa que ahora el mismo solo realiza los accesos a memoria minimos indispensables, lo que tambien implica que ahora utiliza registros para guardar los datos. Ademas el codigo esta mas comprimido, y resulta mas claro de leer.

Ademas precalcula los valores que seran utilizados muchas veces, lo que aumenta la performance, principalmente en casos de instancias grandes.

Los otros flags de optimizacion son -O2, -O3, -Og, -Os, -Ofast.

Ademas encontramos los flags -msse, -msse2, -msse3, -mmmx, -m3dnow, pero al intentar compilar con varios de ellos vimos que gcc no es capaz como para utilizar instrucciones simd.

Tres nombres de optimizaciones son: -fipa-profile, -fipa-reference ,-fmerge-constants

2.3. Mediciones

Realizar una medición de performance *rigurosa* es más difícil de lo que parece. En este experimento deberá realizar distintas mediciones de performance para verificar que sean buenas mediciones.

[1] agregando este flag a `CCFLAGS64` en el makefile

En un sistema “ideal” el proceso medido corre solo, sin ninguna interferencia de agentes externos. Sin embargo, una PC no es un sistema ideal. Nuestro proceso corre junto con decenas de otros, tanto de usuarios como del sistema operativo que compiten por el uso de la CPU. Esto implica que al realizar mediciones aparezcan “ruidos” o “interferencias” que distorsionen los resultados.

El primer paso para tener una idea de si la medición es buena o no, es tomar varias muestras. Es decir, repetir la misma medición varias veces. Luego de eso, es conveniente descartar los outliers [2], que son los valores que más se alejan del promedio. Con los valores de las mediciones resultantes se puede calcular el promedio y también la varianza, que es algo similar el promedio de las distancias al promedio[3].

Las fórmulas para calcular el promedio μ y la varianza σ^2 son

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i \qquad \sigma^2 = \frac{\sum_{i=1}^{n}(x_i - \mu)^2}{n}$$

[2] en español, valor atípico: `http://es.wikipedia.org/wiki/Valor_atpico`

[3] en realidad, elevadas al cuadrado en vez de tomar el módulo

Experimento 1.3 - calidad de las mediciones

1. Medir el tiempo de ejecución de cropflip 10 veces.
2. Implementar un programa en C que no haga más que ciclar infinitamente sumando 1 a una variable. Lanzar este programa tantas veces como *cores lógicos* tenga su procesador. Medir otras 10 veces mientras estos programas corren de fondo.
3. Calcular el promedio y la varianza en ambos casos.
4. Consideraremos outliers a los 2 mayores tiempos de ejecución de la medicion a) y también a los 2 menores, por lo que los descartaremos. Recalcular el promedio y la varianza después de hacer este descarte.
5. Realizar un gráfico que presente estos dos últimos items.

A partir de aquí todos los experimentos de mediciones deberán hacerse igual que en el presente ejercicio: tomando 10 mediciones, luego descartando outliers y finalmente calculando promedio y varianza.

Experimento 1.4 - secuencial vs. vectorial

En este experimento deberá realizar una medición de las diferencias de performance entre las versiones de C y ASM (el primero con -O0, -O1, -O2 y -O3) y graficar los resultados.

Experimento 1.5 - cpu vs. bus de memoria

Se desea conocer cual es el mayor limitante a la performance de este filtro en su versión ASM.

¿Cuál es el factor que limita la performance en este caso? En caso de que el limitante fuera la intensidad de cómputo, entonces podrían agregarse instrucciones que realicen accesos a memoria extra y la performance casi no debería sufrir. La inversa puede aplicarse, si el limitante es la cantidad de accesos a memoria. [4]

Realizar un experimento, agregando 4, 8 y 16 instrucciones aritméticas (por ej `add rax, rbx`) analizando como varía el tiempo de ejecución. Hacer lo mismo ahora con instrucciones de acceso a memoria, haciendo mitad lecturas y mitad escrituras (por ejemplo, agregando dos `mov rax, [rsp]` y dos `mov [rsp+8], rax`).[5]

Realizar un único gráfico que compare: 1. La versión original 2. Las versiones con más instrucciones aritméticas 3. Las versiones com más accesos a memoria

Acompañar al gráfico con una tabla que indique los valores graficados.

Filtro *Sierpinski*

Programar el filtro *Sierpinski* en lenguaje C y en en ASM haciendo uso de las instrucciones vectoriales (**SSE**).

Experimento 2.1 - secuencial vs. vectorial

Analizar cuales son las diferencias de performace entre las versiones de C y ASM de este filtro, de igual modo que para el experimento 1.4.

Experimento 2.1 - cpu vs. bus de memoria

¿Cuál es el factor que limita la performance en este filtro? Repetir el experimento 1.5 para este filtro.

[4] también podría pasar que estén más bien balanceados y que agregar cualquier tipo de instrucción afecte sensiblemente la performance

[5] Notar que en el caso de acceder a `[rbp]` o `[rsp+8]` probablemente haya siempre hits en la cache, por lo que la medición no será de buena calidad. Si se le ocurre la manera, realizar accesos a otras direcciones alternativas.

Filtro *Bandas*

Programar el filtro *Bandas* en lenguaje C y en en ASM haciendo uso de las instrucciones vectoriales (**SSE**).

Experimento 3.1 - saltos condicionales

Se desea conocer que tanto impactan los saltos condicionales en el código de filtro Bandas con `-O1` (la versión en C).
Para poder medir esto de manera aproximada, remover el código que detecta a que banda pertenece cada pixel, dejando sólo una banda. Por más que la imagen resultante no sea correcta, será posible tomar una medida aproximada del impacto de los saltos condicionales. Analizar como varía la performance.

Experimento 3.2 - secuencial vs. vectorial

Repetir el experimento 1.4 para este filtro.

Filtro *Motion Blur*

Programar el filtro *mblur* en lenguaje C y en ASM haciendo uso de las instrucciones **SSE**.

Experimento 4.1

Repetir el experimento 1.4 para este filtro

2.4. Desensamblado de código C y Optimización

Comenzamos analizando el código de Cropflip realizado en C.

Este básicamente solo mueve datos de un lugar de la RAM a otros, sin afectar mayormente la imagen.

Realizamos un objdump para ver el código que genera el compilador gcc. Al desensamblar el código pudimos observar, primero que nada, que C guarda todos los parámetros en la pila y además está escribiendo en memoria todas las variables locales utilizadas, lo cual es innecesario ya que pueden ser almacenadas en registros.

También puede observarse que C utiliza saltos incondicionales, lo que puede sugerir que intenta sacar provecho al sistema de predicción de saltos.

Ademas C genera, luego de la función, un montón de secciones que comienzan con debug_xxx. Estas secciones sirven para ser interpretadas por GDB u otros debuggers.

Como ya dijimos, el código podría optimizarse para no realizar tantos accesos a memoria innecesarios guardando variables locales por ejemplo en registros, lo cual disminuiría el tiempo de ejecución.

Luego de esto, procedemos a compilar el código utilizando el flag -O1, y nuevamente realizamos un objdump para ver el código desensamblado. Se observa que ahora el mismo solo realiza los accesos a memoria mínimos indispensables, utilizando los registros para guardar los datos. Además el código es más compacto, y resulta mas claro de leer. Además precalcula los valores que serán utilizados muchas veces, lo que aumenta la performance, principalmente en casos de instancias grandes.

Los otros flags de optimización son -O2, -O3, -Og, -Os, -Ofast. También podemos encontrar los flags -msse, -msse2, -msse3, -mmmx, -m3dnow, pero al intentar compilar con varios de ellos vimos que gcc no utilizó instrucciones SIMD.

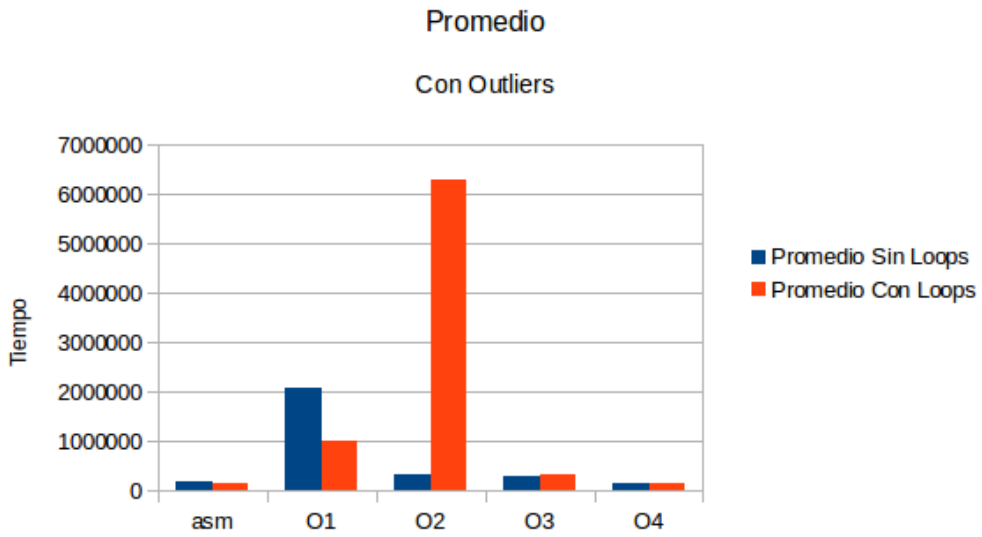
Tres nombres de optimizaciones son: -fipa-profile, -fipa-reference ,-fmerge-constants .

2.5. Calidad de las Mediciones

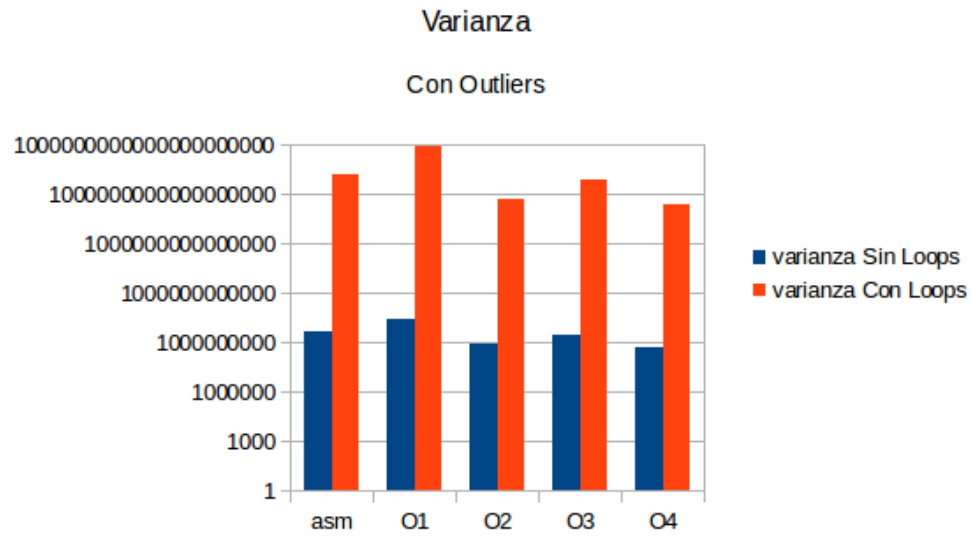
Para este experimento vamos a ver cómo se pueden ver afectados nuestros algoritmos frente a diversos factores de ruido e interferencias que podrían alterar nuestras mediciones.

Para este experimento se utilizó un procesador Intel Atom, de 2 núcleos a 1.6 GHZ con Hyper-Threading, por lo que la cantidad de núcleos lógicos asciende a 4. Por otro lado, para que las pruebas sean mas concisas y exactas, se deshabilitó el scaling dinamico del CPU, ya que esto podría generar ruido innecesario en nuestras mediciones.

Procedimos a tomar 10 mediciones para cada una de las versiones del cropflip, tanto con 4 loops corriendo en paralelo como sin los mismos. Lo que se obtiene es el siguiente gráfico:



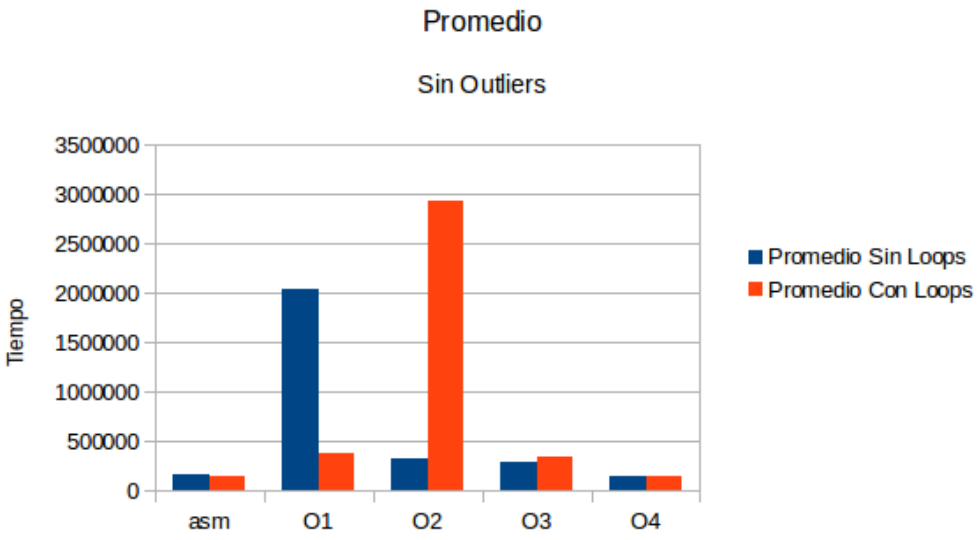
Realizamos un cálculo de la varianza para ver qué tan precisos son los resultados y se obtiene esto:



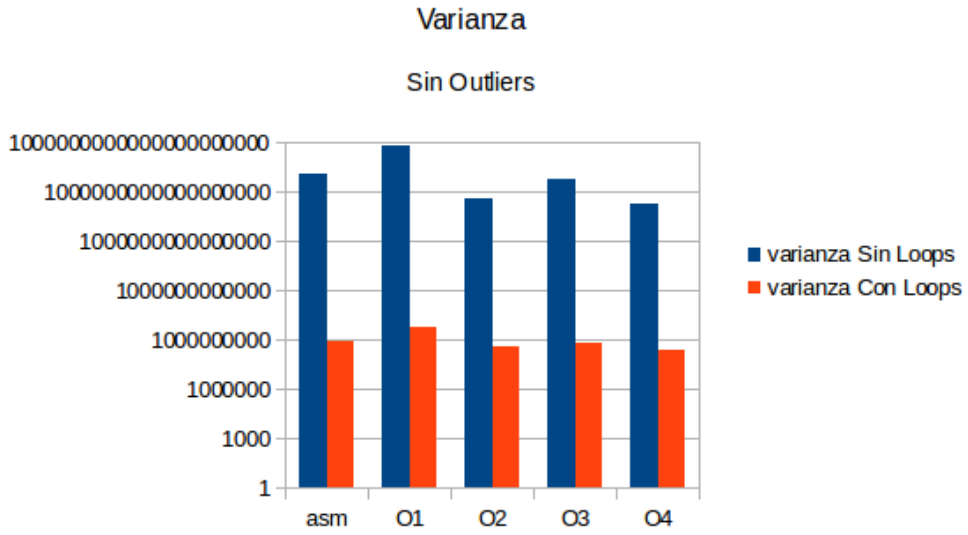
La varianza con loops es sustancialmente mayor que sin los mismos, por lo que se estaría inclinado a correr tests sin loops para obtener valores mas fiables.

Ahora consideramos outliers a los dos valores mas grandes y a los dos valores mas chicos y volvemos a graficar los resultados.

Esto es lo que se obtiene al graficar el promedio:



En este gráfico no se observan cambios significativos. Sin embargo, al calcular nuevamente la varianza, se observa lo siguiente:



Las varianzas de los tests con loops, se ven reducidas dramáticamente, incluso por debajo de las varianzas sin los mismos.

De aquí se concluye, que la mejor manera de realizar tests es con loops corriendo en paralelo y luego, cuando estos valores ya han sido obtenidos, quitando los dos valores mas grandes y los dos valores mas chicos.

3. Cropflip

3.1. Diferencias de performance en Cropflip

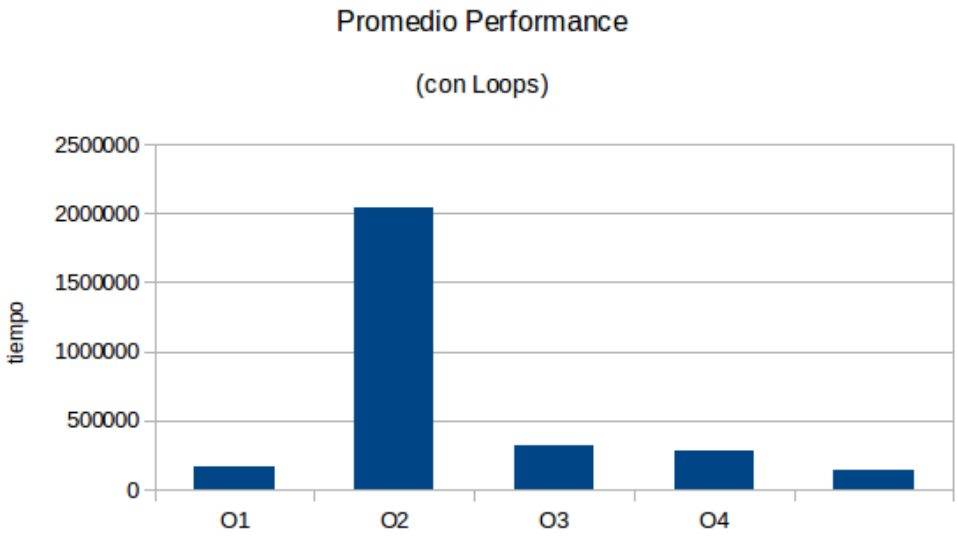
En el siguiente experimento se medirán las performances tanto de nuestro algoritmo en assembler, implementado para sacar provecho de las instrucciones SSE de Intel, como una versión alternativa hecha en C con diversos grados de optimización a cargo del compilador.

El algoritmo de Cropflip en assembler es muy sencillo. Simplemente movemos 128-bits de la imagen a un xmm y de allí al destino, que previamente ha sido seteado para colocar los bits en el lugar correcto. De esta manera, podremos mover de una sola vez, 16 bytes, lo que corresponde a 4 pixels de la imagen.

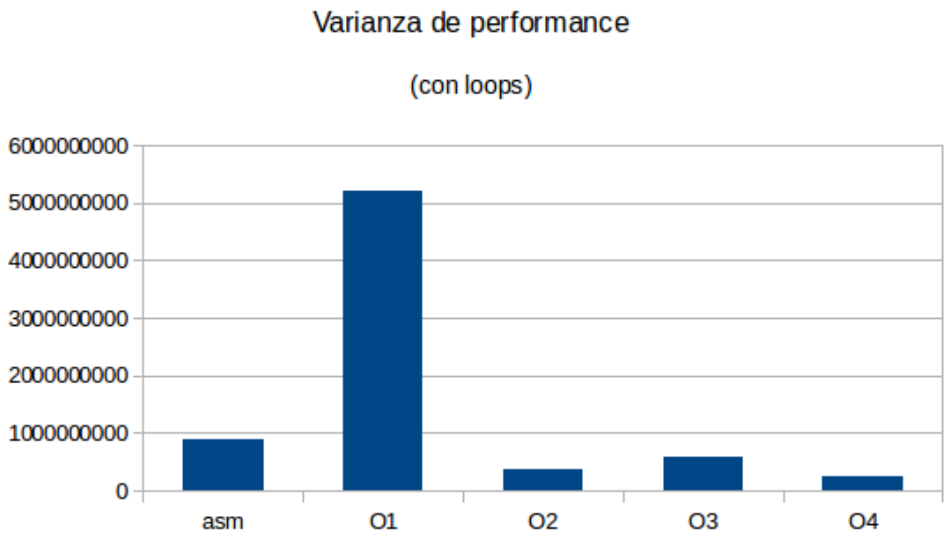
Dado que la cantidad de columnas es siempre múltiplo de 4, o sea, siempre tenemos 4 bytes para tomar, no es necesario chequear otros casos borde.

Las pruebas de performance, realizadas de la misma manera en que concluimos la anterior sección, se realizaron corriendo 4 loops en paralelo junto con los algoritmos de manera de minimizar el ruido y luego quitando los outliers.

Lo obtenido en los tests puede verse en el siguiente gráfico:

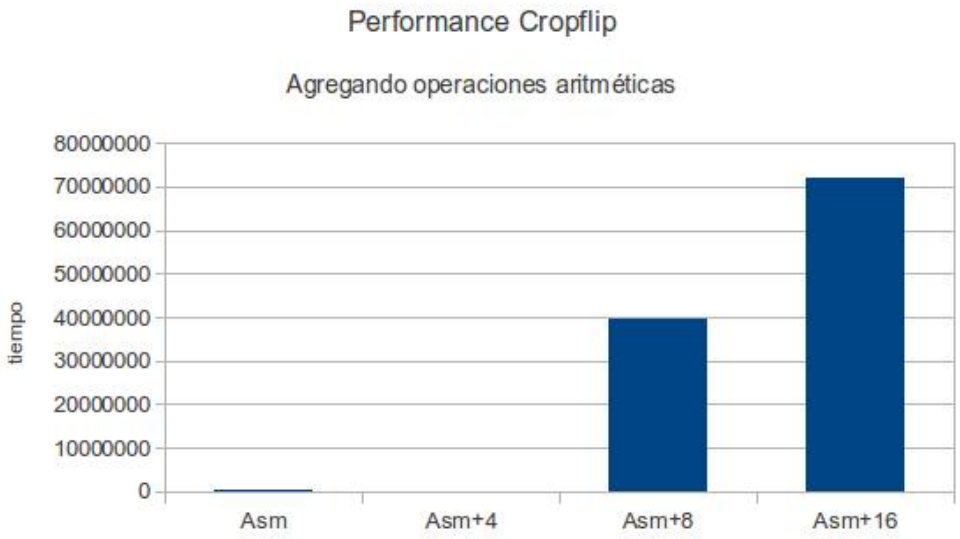


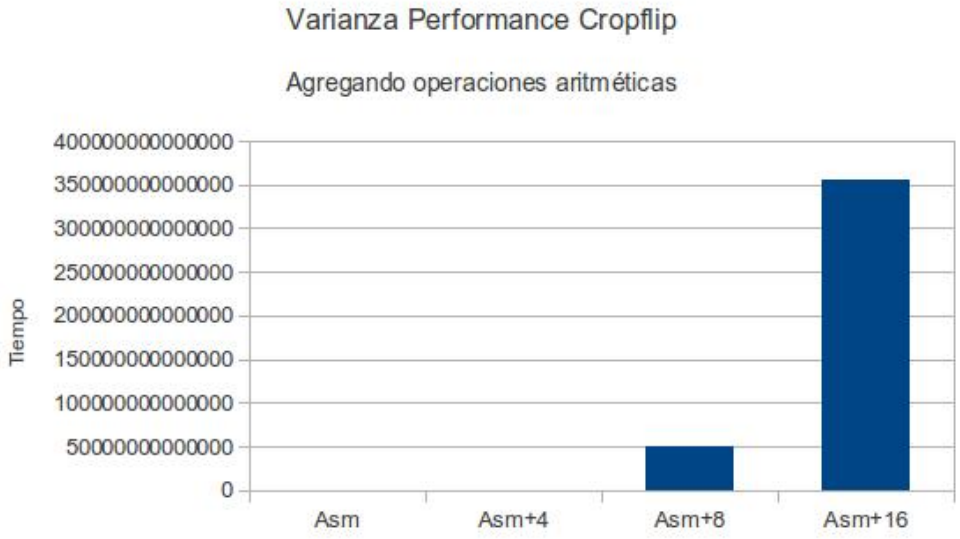
Y las varianzas son:

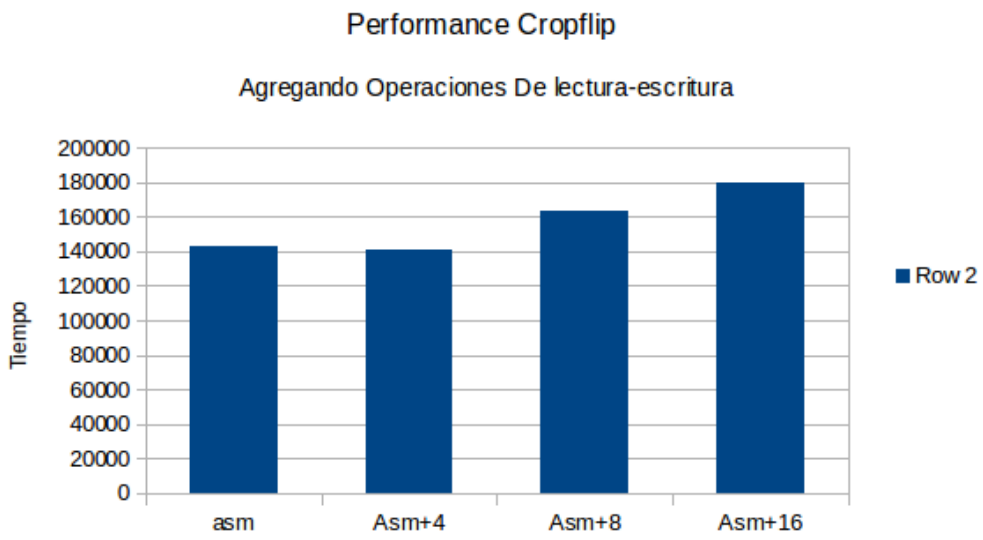


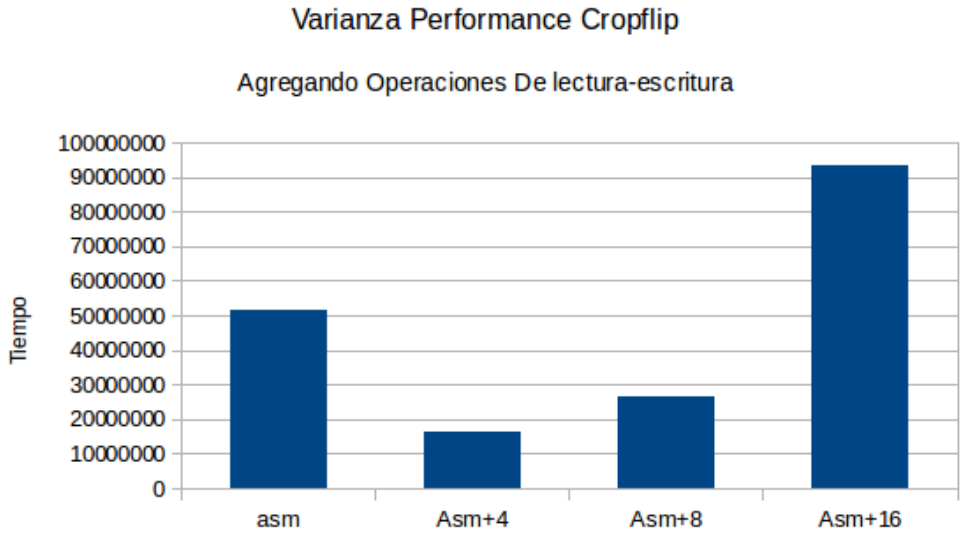
De aquí puede verse que la implementación en assembler es tan buena como la implementación en C con el máximo grado de optimización.

3.2. cpu vs. bus de memoria en Cropflip






Performance Cropflip
Agregando Operaciones De lectura-escritura
Tiempo
200000
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
asm
Asm+4
Asm+8
Asm+16
Row 2


Varianza Performance Cropflip
Agregando Operaciones De lectura-escritura
Tiempo
100000000
90000000
80000000
70000000
60000000
50000000
40000000
30000000
20000000
10000000
0
asm
Asm+4
Asm+8
Asm+16

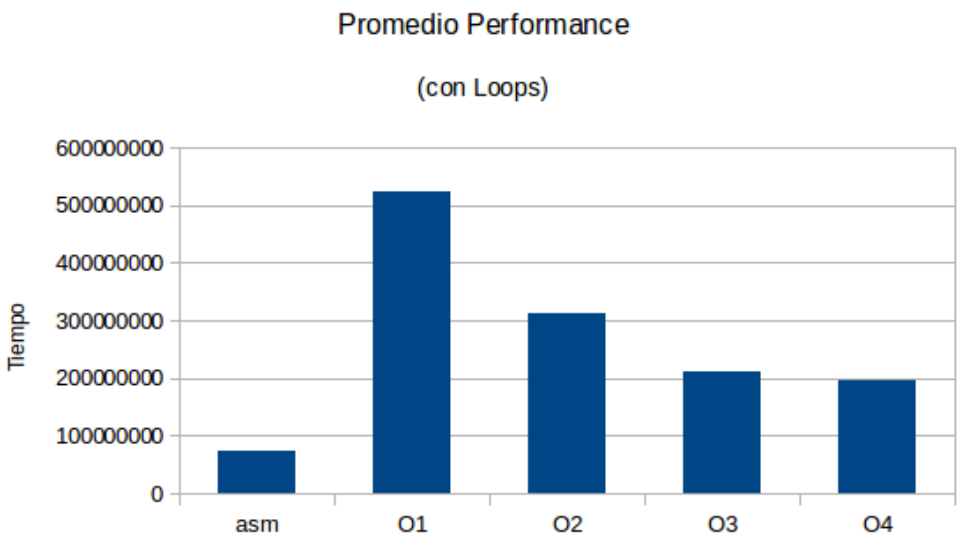
4. Sierpinski

4.1. Diferencias de performance en Sierpinski

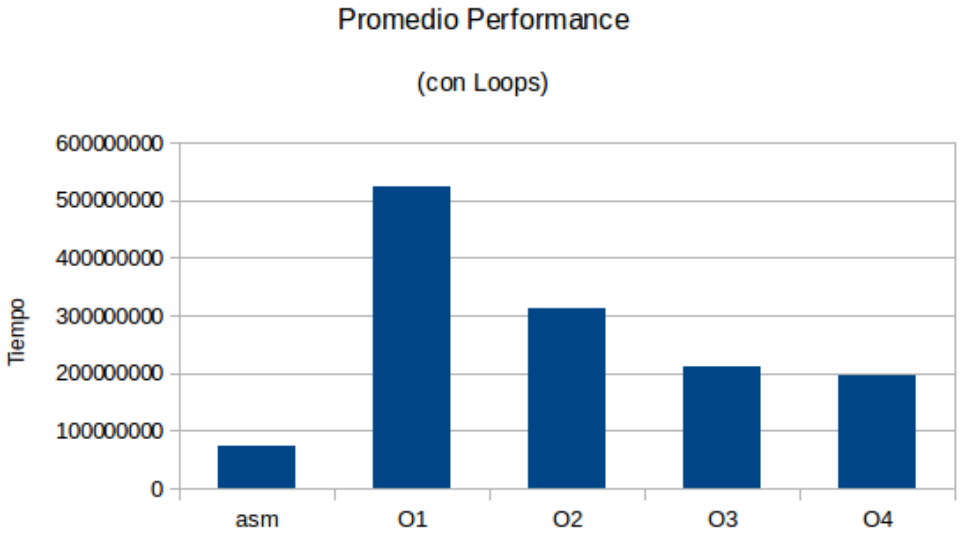
Ahora analizamos el algoritmo del Sierpinski. En este caso, el algoritmo ya es un poco más complejo. Necesitamos calcular para cada columna un valor constante diferente, que dependerá de cuál sea la misma.
Luego para poder paralelizar de alguna manera el algoritmo en C y sacar provecho a los registros xmm, es necesario calcular 4 constantes a la vez y multiplicárselas a sus respectivos pixels.
Luego la idea del algoritmo será algo así:

- En la seccion data guardamos una constante con el valor 255 brodcasteado en punto flotante.
- Previo al ciclo, guardamos este valor en un registro.
- Ya dentro del ciclo, leemos 4 pixels y los guardamos en un xmm.
- A continuación calculamos de manera paralela para cada pixel el coeficiente correspondiente.
- Primero, realizamos la división de i por la cantidad de filas y j por la cantidad de columnas, para la cual previamente pasamos de entero a punto flotante.
- Multiplicamos ambos valores por 255 y luego convertimos a entero con truncamiento.
- Realizamos un xor entre ambos y despues, volvemos a convertir a punto flotante para dividir por 255.
- Ya con los coeficientes calculados, solo nos resta multiplicar cada uno por el pixel correspondiente.
- Para ello desempaquetamos cada byte de cada pixel leido a word, y luego de word a dword, para así convertirlo a punto flotante.
- Una vez hecha la conversión, brodcasteamos cada coeficiente para poder multiplicarlo por el pixel correspondiente.
- Luego convertimos a entero nuevamente y empaquetamos todo de dword a word y de word a byte.
- Y finalmente movemos los pixels procesados al destino.

Los resultados comparativos de performance para este algoritmo comparado con uno iterativo desarrollado en C fueron los siguientes:



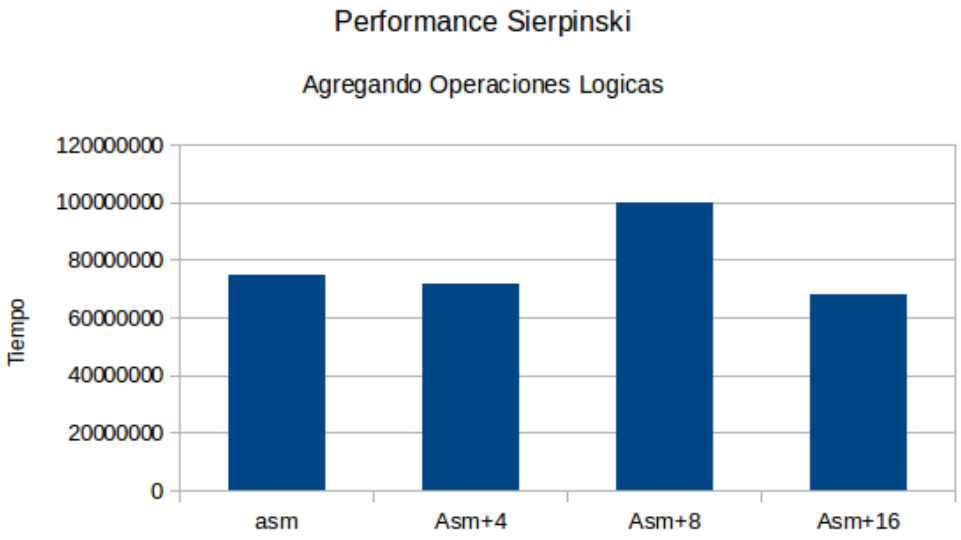
Y la varianza:

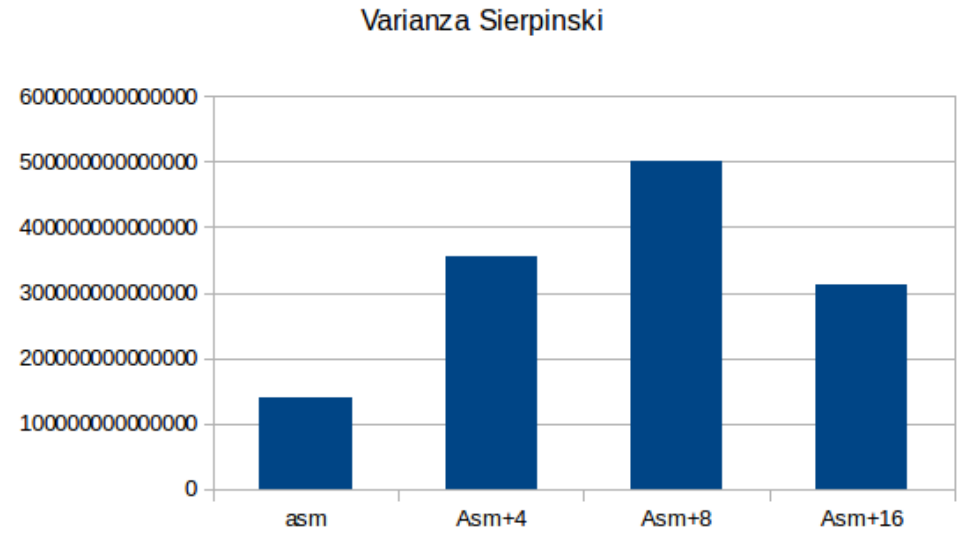


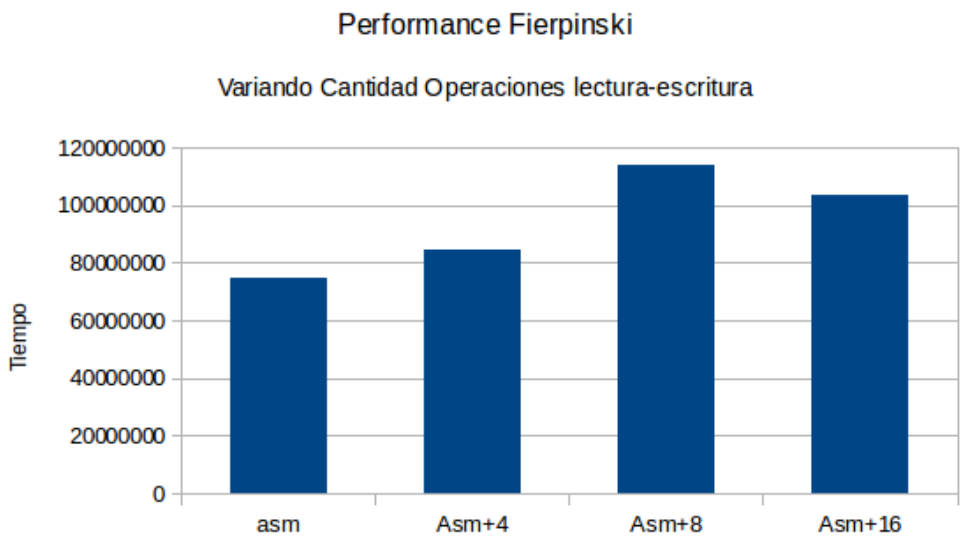
Puede verse que en este caso nuestro modelo que toma y calcula de a 4 pixels utilizando instrucciones SIMD es incluso mejor que la versión de C con mayor grado de optimización.

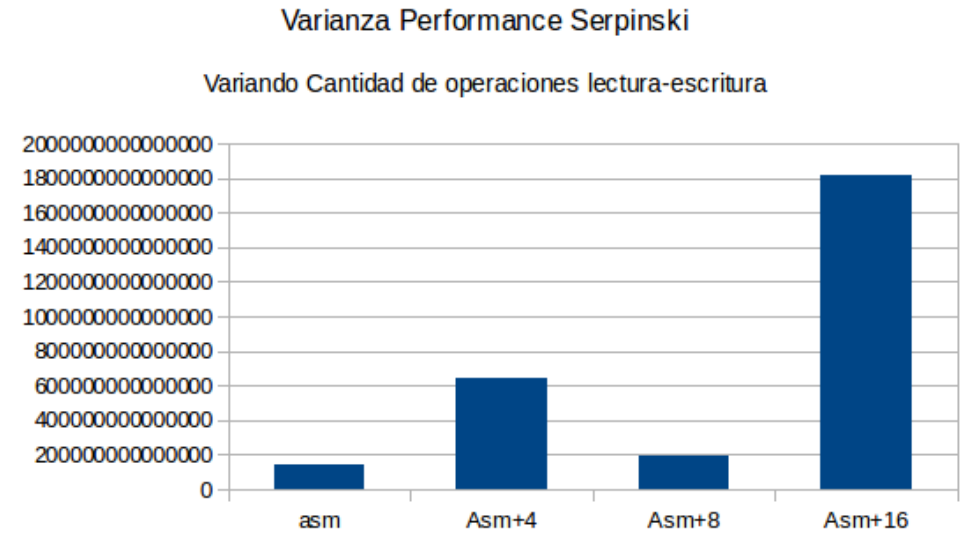
4.2. cpu vs. bus de memoria en Sierpinski

HACER =)









5. Bandas

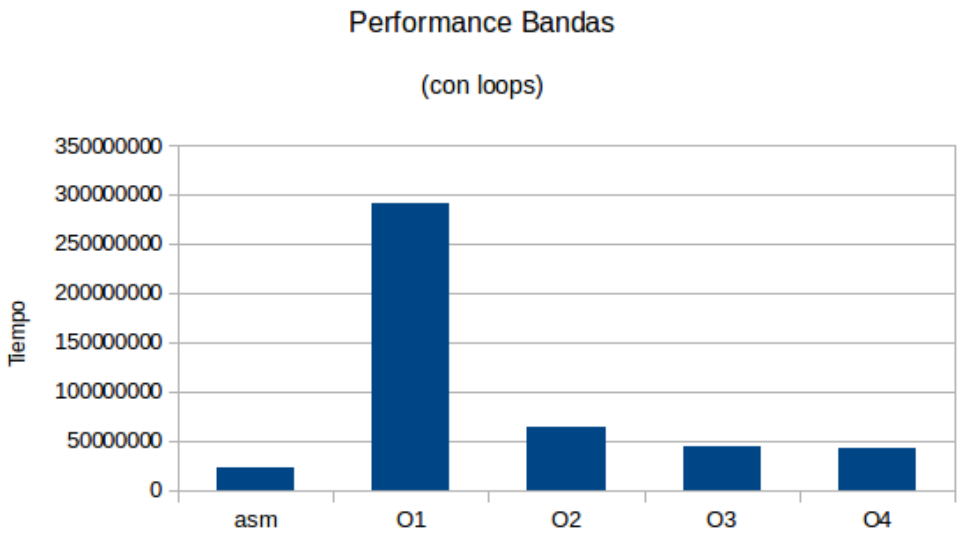
5.1. Diferencias de performance en Bandas

Para el algoritmo de bandas se nos presenta otro desafío: debemos tomar los tres colores de la imagen(r,g,b), sumarlos, y luego comparar cada uno de ellos para ver si se encuentra en un rango determinado.
Para resolver la primera problemática usaremos las instrucciones $phaddw$, que nos permitirá a través de una suma horizontal, sumar los valores r,g,b de manera cómoda solamente utilizando dos registros.
El segundo problema será comparar estos valores obtenidos en la suma de una manera eficiente. Querríamos compararlos todos a la vez y a partir de esas comparaciones determinar que valores deberá ir en cada pixel. Esta se resolverá utilizando broadcasting. El algoritmo irá comparando en cada paso contra un valor y en caso de cumplirse una condición, restará donde corresponda.

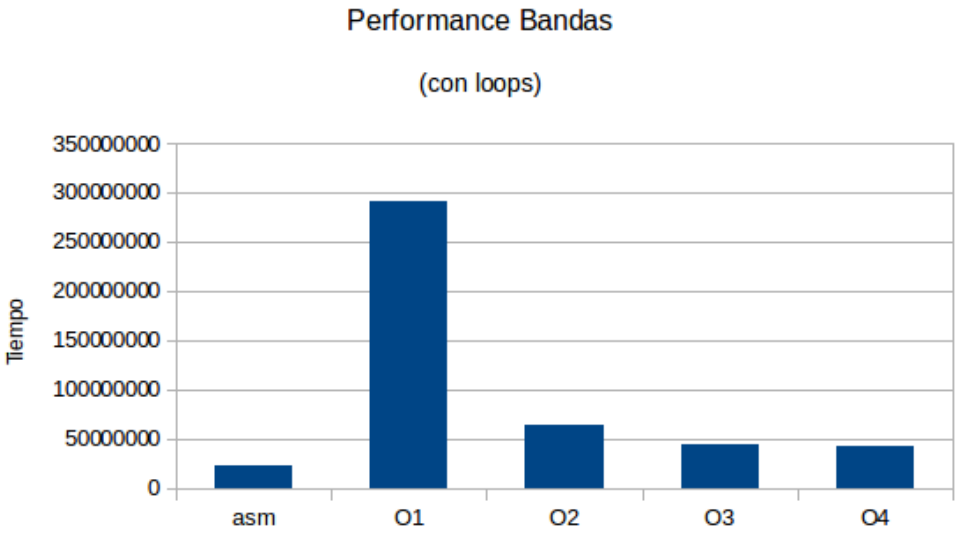
- En la sección data tenemos mascaras y constantes brodcasteadas que utilizaremos para hacer las comparaciones.

- Ademas en esta sección tenemos una doble qword la cual usaremos para el shuffle final.
- Previo al ciclo donde procesamos los pixels, movemos estos datos a registros xmm.
- Ya una vez dentro del ciclo, leemos 4 pixels y los guardamos en un xmm.
- Desempaquetamos los bytes a word para poder hacer la suma de las componentes de cada pixel sin irnos de la representación.
- Hacemos dos sumas horizontales para obtener los cuatro valores de b.
- Luego utilizamos las mascaras para comparar en paralelo cada pixel con el b correspondiente y asignando el valor de rgb segun corresponda.
- Nos queda en cada word de la parte baja de un xmm, los 4 valores que se asignaran a cada rgb de cada pixel.
- Finalmente, empaquetamos estos valores para volver a byte y los shuffleamos para dejarlos en los lugares correspondiente y lo asignamos en el destino.

Los resultados de los tiempos comparativos son los siguientes:



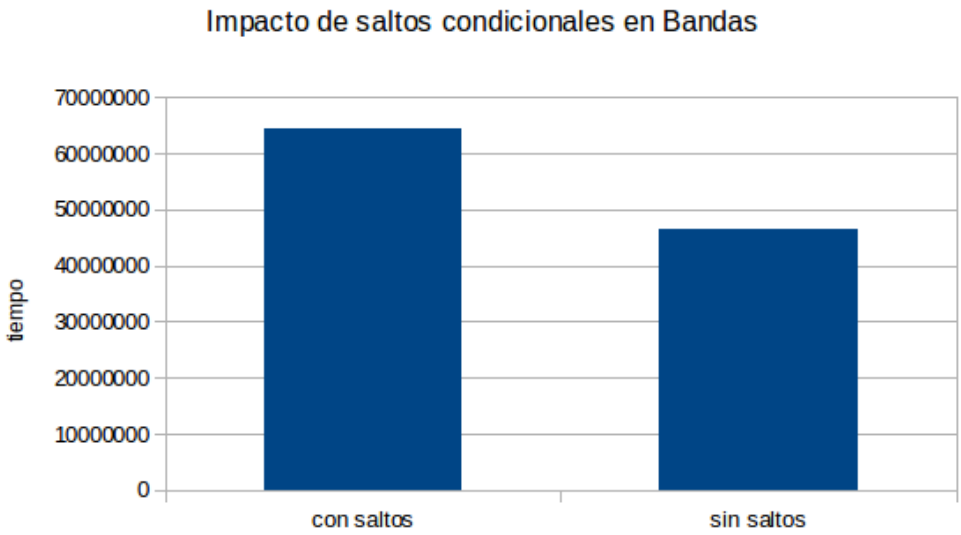
Y la varianza:

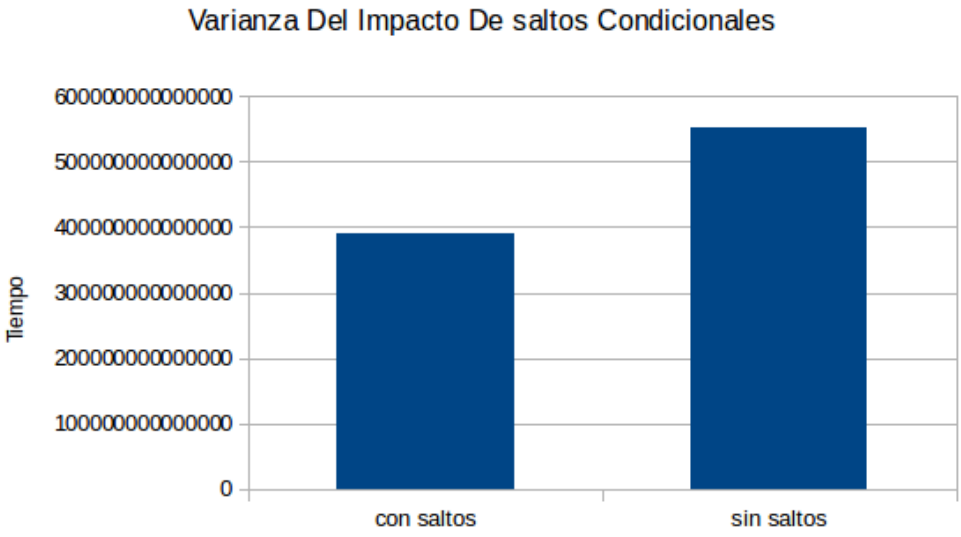


Nuestro algoritmo obtiene una performance mucho mayor a la del código sin optimizar, y una performance casi idéntica al del código C con el mayor grado de optimización.

5.2. saltos condicionales

HACER =)





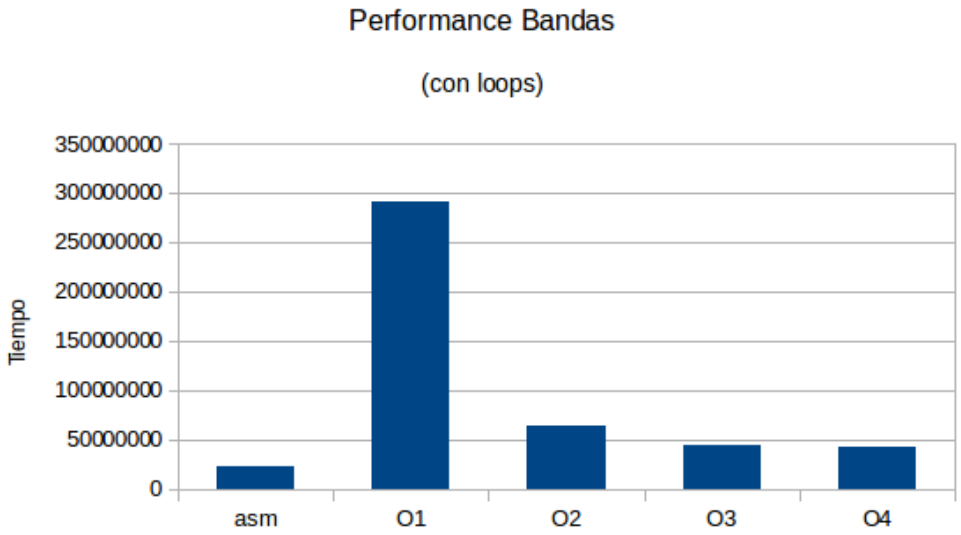
5.3. Motion Blur

5.4. Diferencias de performance en Motion Blur

Para este algoritmo por cada pixel se deben tomar 5 pixels, multiplicar cada uno de los colores por 0.2 y luego sumarlos.
Para ello debemos tener cuidado de tomar correctamente los casos borde y no aplicar motion blur donde no corresponde.
Una idea general de cómo funciona el algoritmo en assembler es la siguiente:

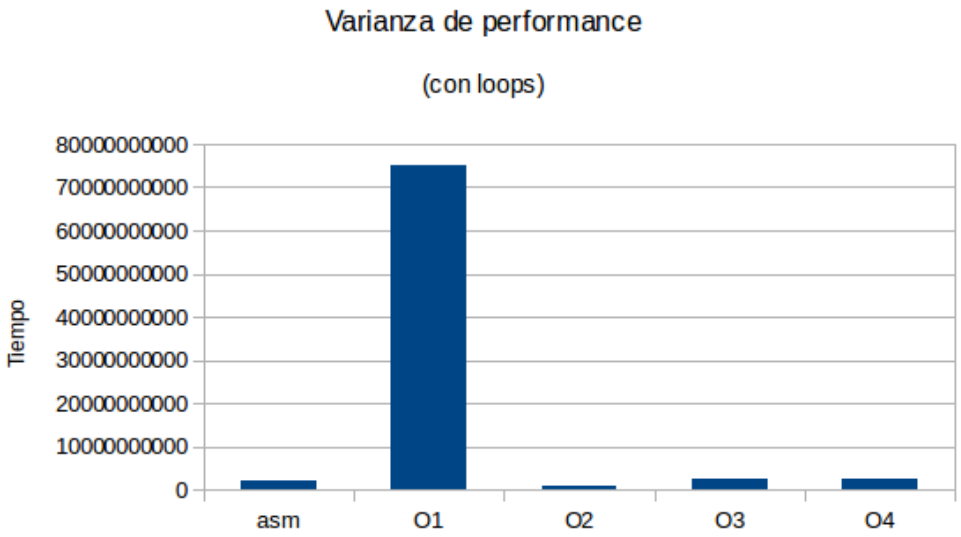
- Tenemos, en la sección datos, el valor 0.2 brodcasteado que utilizaremos para realizar la multiplicación.
- Previo al procesamiento de datos, guardamos en un xmm este valor.
- Dentro del ciclo, para procesar el pixel (i,j) tomamos los 4 pixels de las filas $i-2$, $i-1$, i, $i+1$ e $i+2$ a partir de la columna $j-2$, $j-1$, j, $j+1$ y $j+2$, respectivamente, y almacenamos todo en registros.
- Desempaquetamos todo a word para poder hacer la suma sin desbordar.
- Luego de realizar las sumas, desempaquetamos a dwords para así poder convertir a punto flotante.
- Con la conversión ya hecha, podemos multiplicar cada valor por 0.2.
- Convertimos a entero nuevamente y empaquetamos de dword a word y de word a byte.
- Finalmente, copiamos los 4 pixels resultantes al destino.

Al realizar el testing se obtuvieron los siguientes resultados:



Y la varianza:
En este caso nuestro algoritmo supera ampliamente incluso al código C con mayor grado de optimización.

6. Conclusiones y trabajo futuro


Varianza de performance
(con loops)
Tiempo
80000000000
70000000000
60000000000
50000000000
40000000000
30000000000
20000000000
10000000000
0
asm
O1
O2
O3
O4