



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

## Índice

<b>1. Objetivos generales</b>	<b>3</b>
<b>2. Enunciado y solución</b>	<b>3</b>
2.1. Enunciado . . . . .	3
2.2. Filtro cropflip . . . . .	3
2.3. Mediciones . . . . .	3
2.4. Desensamblado de código C y Optimización . . . . .	7
2.5. Calidad de las Mediciones . . . . .	8
2.6. Diferencias de performance en Cropflip . . . . .	10
2.7. cpu vs. bus de memoria en Cropflip . . . . .	11
2.8. Diferencias de performance en Sierpinski . . . . .	12
2.9. cpu vs. bus de memoria en Sierpinski . . . . .	13
2.10. Diferencias de performance en Bandas . . . . .	14
2.11. cpu vs. bus de memoria en Bandas . . . . .	15
2.12. Diferencias de performance en Motion Blur . . . . .	16
<b>3. Conclusiones y trabajo futuro</b>	<b>17</b>

## 1. Objetivos generales

El objetivo de este Trabajo Práctico es mostrar las variaciones en la performance que suceden al utilizar instrucciones SIMD en comparacion con codigo C con diversos grados de optimizacion realizados por el compilador.

Para ello se realizaran cuatro filtros de fotos, Cropflip, Bandas, Sierpinski y Motion Blur, tanto en codigo assambler que aproveche las instucciones SSE brindadas para los procesadores de arquitectura Intel como codigo C, al que se le apilcarán los distintos flags de optimizacion -O0, -O1, -O2 y -O3.

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

El segundo, tercer y cuarto filtro, se sentrán, en la variacion de performance entre utilizar instrucciones SIMD, para realizar diversos calculos (sumas, multiplicaciones, divisiones) tanto en representacion de enteros como punto flotante.

## 2. Enunciado y solucion

### 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<sup>1</sup>. 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.

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

---

<sup>1</sup>agregando este flag a CCFLAGS64 en el makefile

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 <sup>2</sup>, 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 al promedio de las distancias al promedio<sup>3</sup>.

Las fórmulas para calcular el promedio  $\mu$  y la varianza  $\sigma^2$  son

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

---

<sup>2</sup>en español, valor atípico: [http://es.wikipedia.org/wiki/Valor\\_atpico](http://es.wikipedia.org/wiki/Valor_atpico)

<sup>3</sup>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 medición 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.<sup>4</sup>

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`).<sup>5</sup>

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

---

<sup>4</sup>también podría pasar que estén más bien balanceados y que agregar cualquier tipo de instrucción afecte sensiblemente la performance

<sup>5</sup>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 -01 (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 algoritmo el código del Cropflip del algoritmo 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, lo que es innecesario, esta escribiendo en memoria todas las variables utilizadas.

También notamos que utiliza las variables locales desde memoria en vez de guardarlas en registros.

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

Además 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, lo que también implica que ahora utiliza registros para guardar los datos. Además el código está más comprimido, y resulta más 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.

Además 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

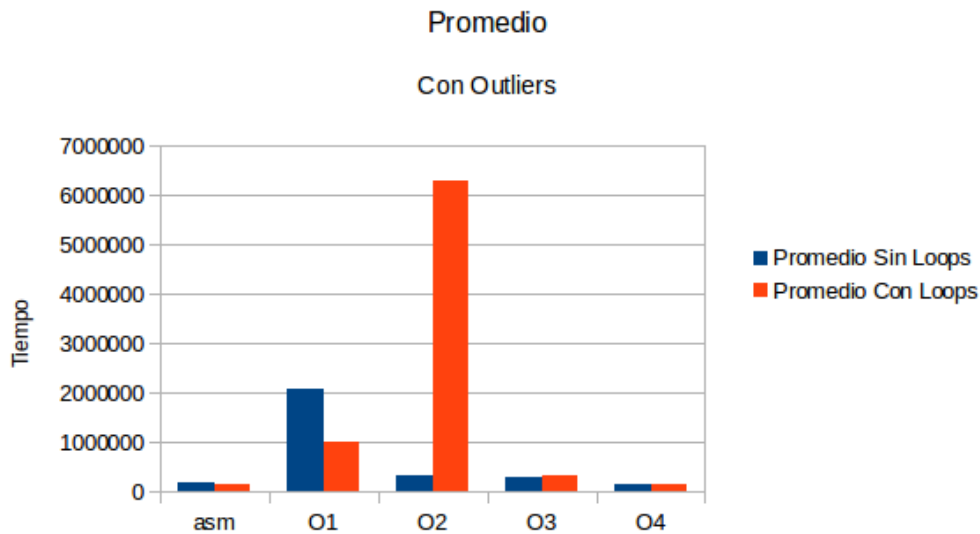
## 2.5. Calidad de las Mediciones

Para este experimento vamos ver como se puede ver afectado nuestro programa frente a diversos factores de ruido e interferencias que puedan alterar nuestras mediciones.

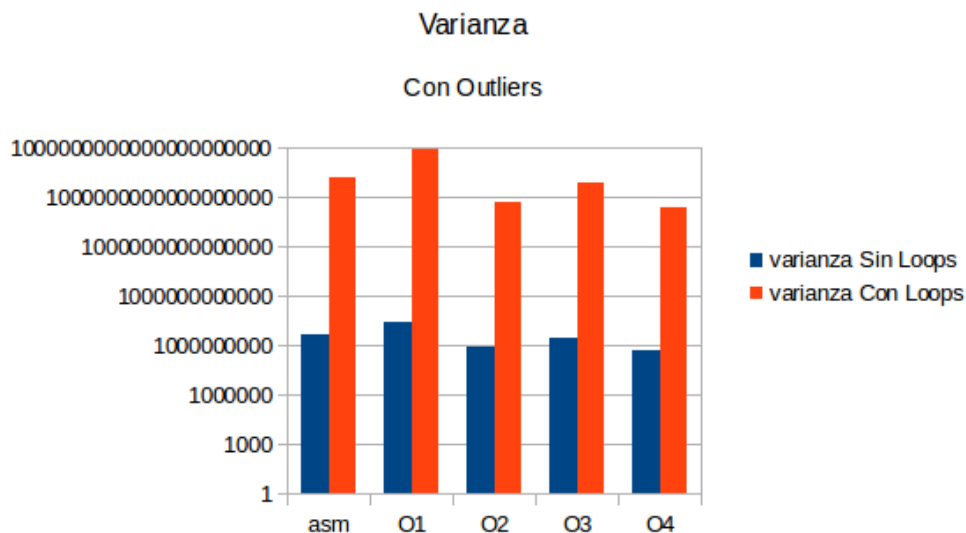
Para este experimento se utilizó un procesador Intel Atom, de 4 núcleos a 1.6 GHz con Hyper-Threading. Por lo que la cantidad de núcleos lógicos asciende a 8.

Para que las pruebas sean más concisas y exactas, se deshabilita el scaling dinámico del CPU, ya que esto podría generar ruido innecesario en nuestras mediciones.

Procedemos a tomar 10 mediciones, tanto con 8 loops corriendo en paralelo como sin los mismos. Lo que se obtiene es el siguiente gráfico:



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

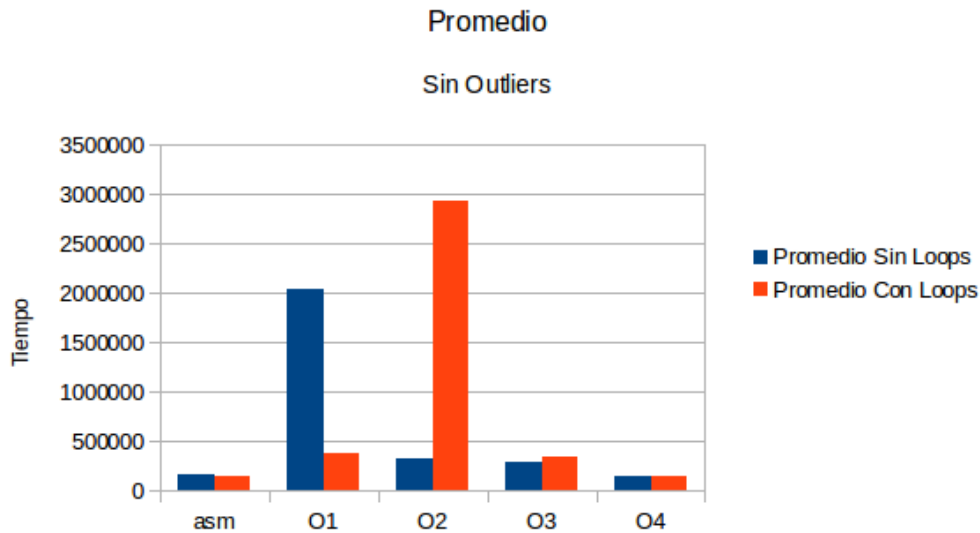


La varianza con loops es sustancialmente mayor que sin los mismos, por lo que se estaría inclinado a correr tests sin loops para obtener valores más fiables.

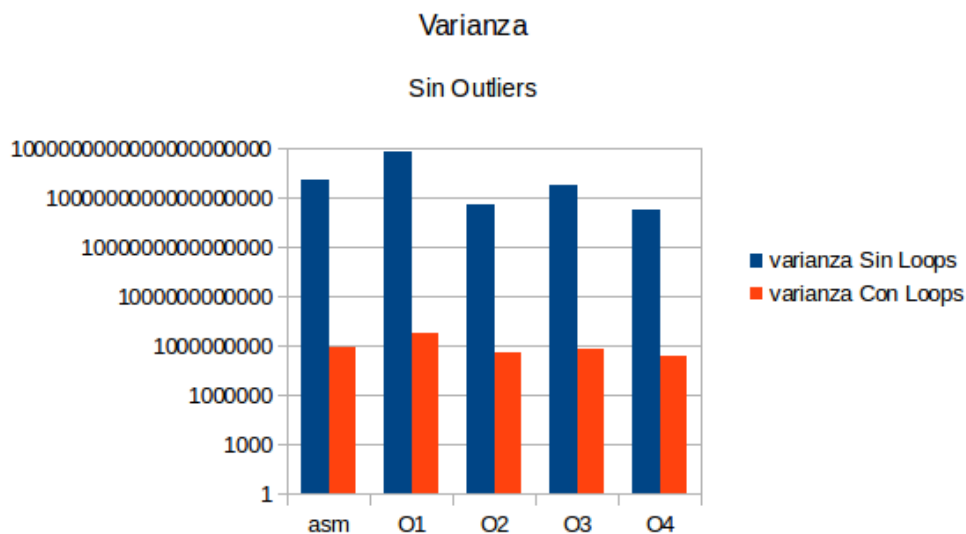


Ahora consideramos outliers a los dos valores mas grandes y a los dos valores mas chicos y volvemos a graficar los resultados.

Esto es lo que se obtiene al graficar el promedio:



En este grafico 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 drasticamente, incluso por debajo de las varianzas sin los mismos.

De aqui 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.

## 2.6. Diferencias de performance en Cropflip

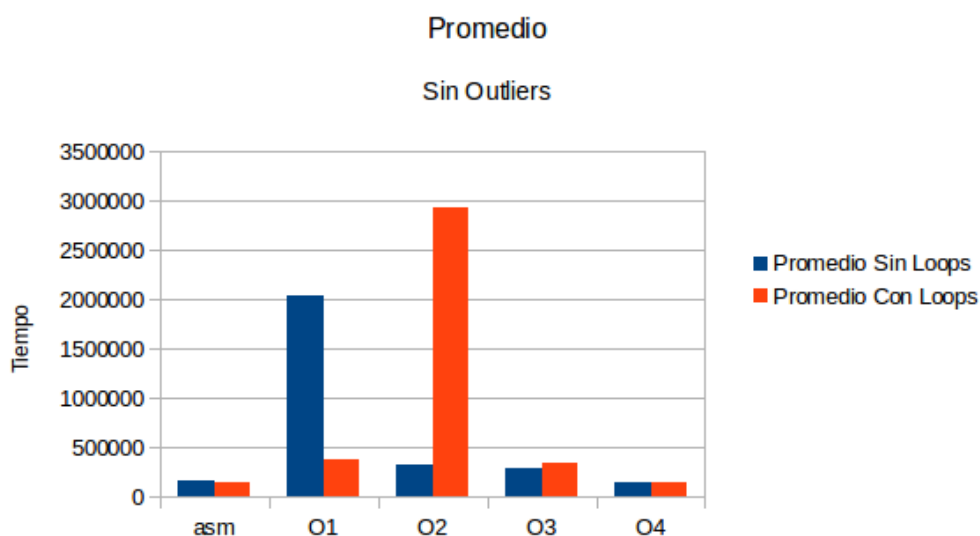
En el siguiente experimento se mediran las performances tanto de nuestro algoritmo en assembler, implementado para sacar provecho de las instrucciones SSE de Intel, como una versión alternativa hecha en C con diversos grados de optimizacion a cargo del compilador.

El algoritmo de cropflip para Cropflip es muy sencillo. Simplemente movemos 128-bits de la imagen a un mmx y de alli al destino, que previamente a 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.

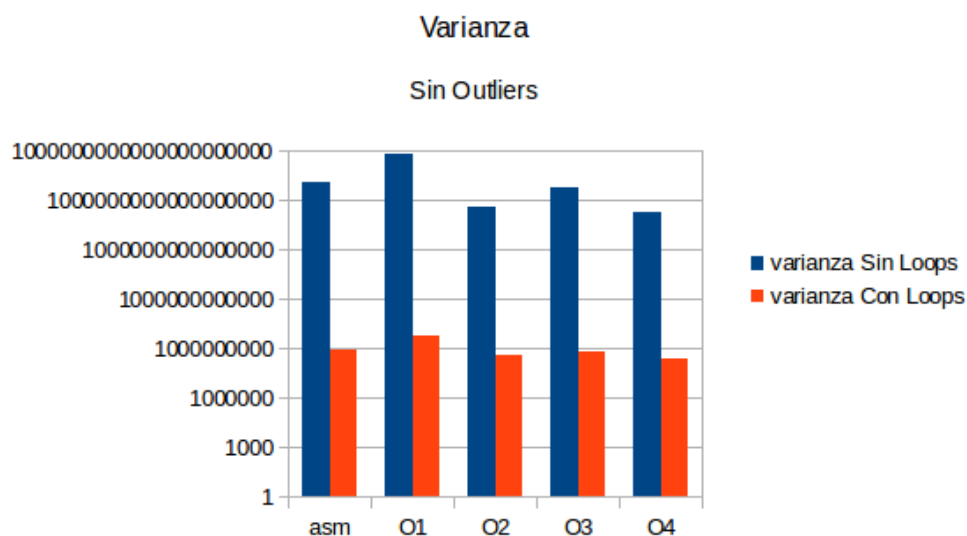
Dado que la cantidad de columnas es siempre multiplo de 4, osea, siempre tenemos 4 bytes para tomar, no es necesario chequear otros casos borde.

Las pruebas de performance, realizadas de la misma manera en que concluimos la anterior seccion, se realizaron corriendo 8 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 grafico:



Y las varianzas son:



De aqui puede verse que la implementación en assambler es tan buena, como la implementacion en C con el maximo grado de optimización.

## 2.7. cpu vs. bus de memoria en Cropflip

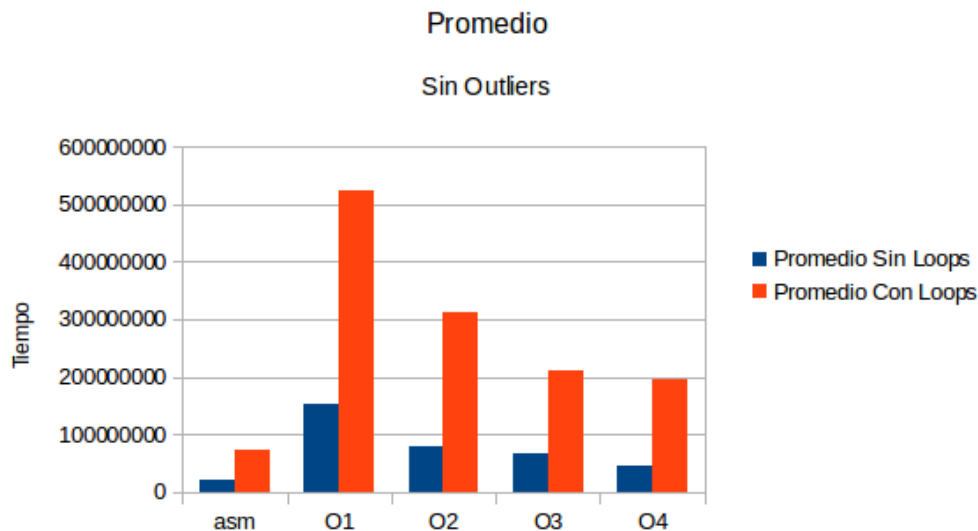
HACER =)

## 2.8. Diferencias de performance en Sierpinski

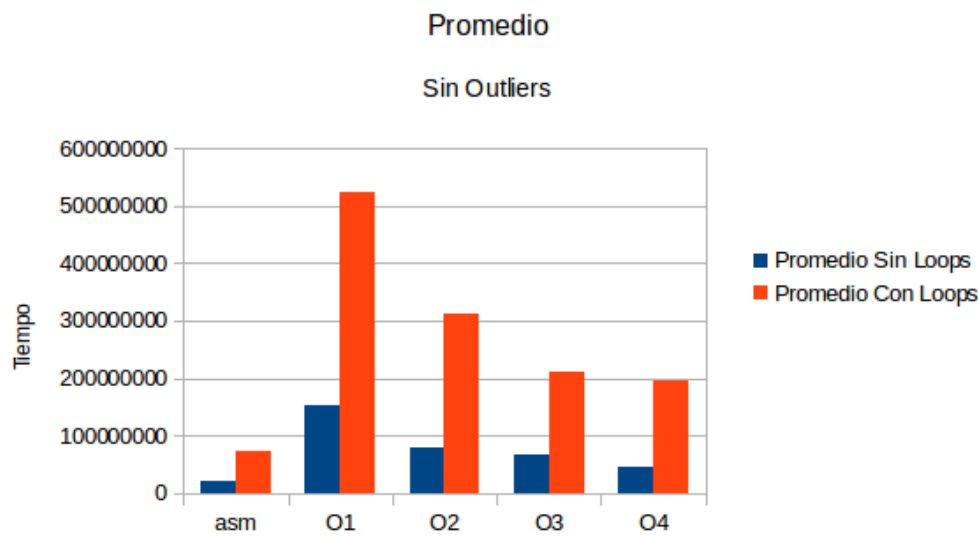
Ahora analizamos el algoritmo del Sierpinski. En este caso, el algoritmo ya es un poco mas complejo. Necesitamos calcular para cada columna, una constante diferente, que dependerá de cual sea la misma. Luego para poder paralelizar de alguna manera el algoritmo en C y sacar provecho a los registros xmm,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

```
Pongo r10 y r11 en 0, estos seran la filas (i) y columnas (j) en la que estoy parado.
Pongo en xmm8 la cantidad de columnas, lo broadcasteo y lo convierto a float
Pongo en xmm9 la cantidad de filas, lo broadcasteo y lo convierto a float
Muevo a xmm13 la constante 255 broadcasteada y en formato float
Muevo a xmm15 los valores 0,1,2,3 en formato entero.
Seteo xmm14 en 0.
Luego, para cada paso iterativo:
  Tomo 4 pixels de la fuente y los pongo en xmm0.
  Muevo r10 y r11 a xmm10 y xmm11 respectivamente, los broadcasteo.
  Sumo xmm10 y xmm15 para obtener el numero i apropiado para cada pixel. tendré (i+0,i+1,i+2,i+3).
  Convierto xmm10 y xmm11 a punto flotante.
  Divido xmm10 por xmm9 y xmm11 por xmm8. (xmm10= i/filas), y (xmm11= j/columnas).
  Multiplico xmm10 y xmm 11 por xmm13, osea multiplico ambos valores por la constante 255 previamente broadcast
  Convierto xmm10 y xmm11 nuevamente a entero y realizo un xor entre ellos.
  Paso a float xmm10 nuevamente y lo divido por xmm13.
  Hasta aqui hemos calculado el coeficiente $b$ para cada uno de los 4 pixels.
  Ahora solo resta tomar los 4 pixels que teniamos en xmm0, desempaquetarlos y convertirlos a float
  Multiplicarlos cada uno de los pixels por su constante $b$
  Convertir nuevamente los valores a byte y ponerlos en el 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.

**2.9. cpu vs. bus de memoria en Sierpinski**

HACER =)

## 2.10. Diferencias de performance en Bandas

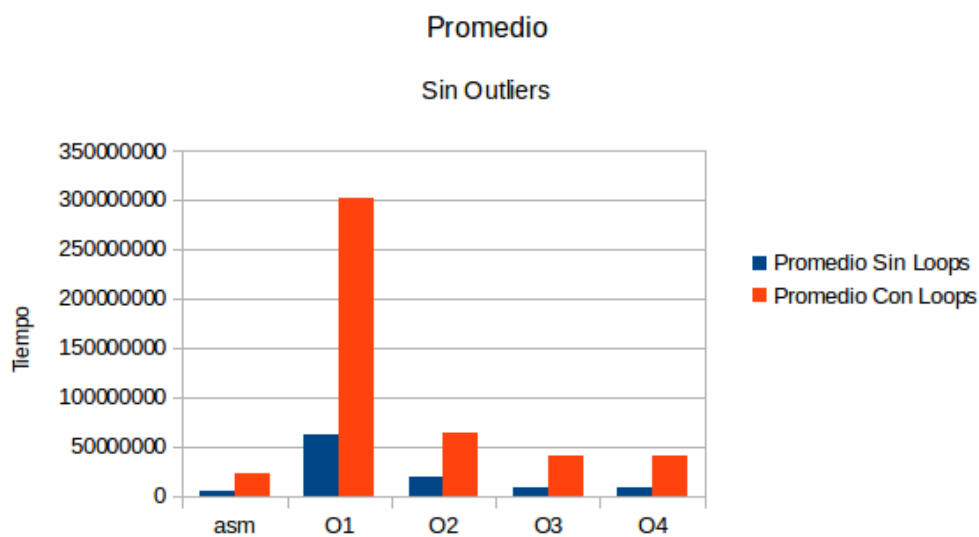
Para el algoritmo de bandas se nos presenta otro desafío: debemos tomar los tres colores de la imagen(r,g,b), sumarlos, y luego comparar cada uno de ellos para ver si se encuentra en un rango determinado.

Para resolver la primera problemática usaremos las instrucciones *phaddw*, que nos permitirá atravesar de una suma horizontal, sumar los valores r,g,b de manera cómoda solamente utilizando dos registros.

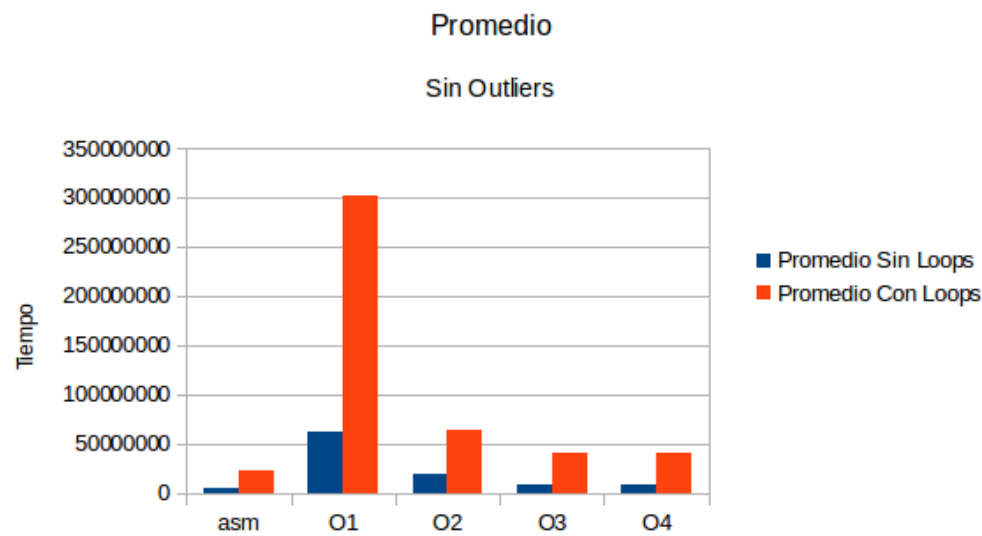
El segundo problema será comparar estos valores obtenidos en la suma de una manera eficiente. Querriamos compararlos todos a la vez y a partir de esas comparaciones determinar que valores deberá ir en cada pixel. Esta se resolverá utilizando broadcasting. El algoritmo irá comparando en cada paso contra un valor y en caso de cumplirse una condición, restará donde corresponda.

### Algoritmo

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.

2.11. cpu vs. bus de memoria en Bandas

HACER =)

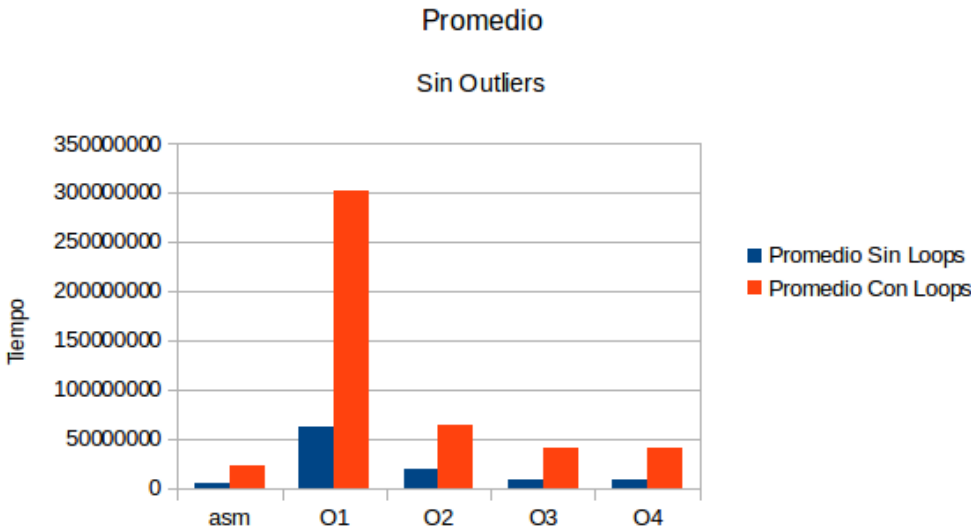
### 2.12. Diferencias de performance en Motion Blur

Para este algoritmo por cada pixel se deben tomar 5 pixcels, multiplicar cada uno de los colores por 0.2 y luego sumarlos.

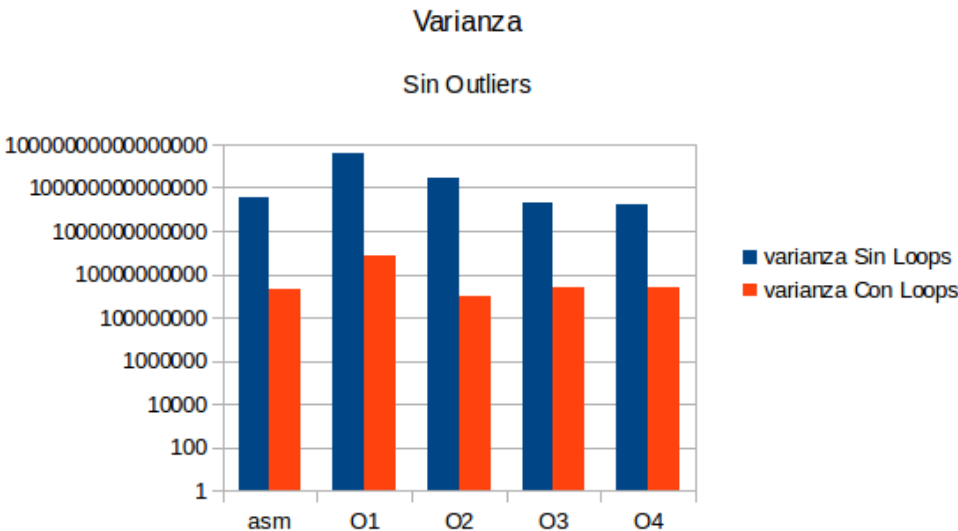
Para ello debemos tener cuidado de tomar correctamente los casos borde y no aplicar motion blur donde no corresponde.

Algoritmo

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.



### **3. Conclusiones y trabajo futuro**