



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
Fosco, Martin Esteban	449/13	mfosco2005@yahoo.com.ar
Palladino, Julián	231/13	julianpalladino@hotmail.com
De Carli, Nicolás	164/13	nikodecarli@gmail.com



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**

<b>1. Objetivos generales</b>	<b>3</b>
<b>2. Desarrollo</b>	<b>3</b>
2.1. Explicación de Implementaciones en assembler . . . . .	3
2.2. Experimentos . . . . .	4
<b>3. Enunciado y solucion</b>	<b>13</b>
3.1. Filtro cropflip . . . . .	13
3.2. Mediciones . . . . .	13
<b>4. Conclusiones y trabajo futuro</b>	<b>15</b>

## 1. Objetivos generales

Este Trabajo Práctico se ha centrado en explorar el modelo de programación **SIMD**, usándolo para una aplicación popular del set de instrucciones SIMD de intel (SSE), procesamiento de imágenes y videos. En particular, se implementaron 4 filtros de imágenes: Cropflip, Sierpinski, Bandas y Motion Blur.

## 2. Desarrollo

### 2.1. Explicación de Implementaciones en assembler

**Bandas-** El filtro Bandas lo intentamos implementar tres veces. En esas tres fuimos variando la forma de pensar el filtro, sobretodo el 'branching' que generaba la parte de pasar de la suma de R, G y B al número que teníamos que guardar en el destino.

Con condicionales:

La primer manera (la más ineficiente de las 3) fue intentar replicar el código C y hacer los condicionales correspondientes. Esta forma no sólo resultó ser la más difícil de implementar (El código se tornó muy largo y confuso), sino que nos dimos cuenta rápidamente que era extremadamente ineficiente y no llegamos a terminarla.

Con cálculos:

La segunda forma consistió en aplicar una cuenta matemática para pasar de la suma (de R, G y B) al resultado final (La cuenta en sí era  $res = \lceil ([suma/96] + 1)/2 \rceil * 64$ ). Esta nueva manera de pensar el ejercicio consiguió eliminar el enorme branching que generaban los condicionales de la implementación anterior, pero todavía podría ser optimizado aún más. Con máscaras:

Finalmente, por consejo de algunos docentes, decidimos implementar el ejercicio con el uso de máscaras. La idea es, en cada iteración, levantar dos píxeles en un XMM pasando R, G, B y A a Words. Luego en otros dos XMM hacer rotaciones para que queden alineados verticalmente los R, G y B de ambos píxeles. Finalmente con una suma vertical (paddw) y dos pshufw para hacer 'broadcast', obtenemos un XMM con las sumas R+G+B de ambos píxeles 'broadcasteadas' en words. (Este proceso lo hacemos dos veces por ciclo, para trabajar con 4 píxeles por iteración). Luego, en lugar de hacer el branching innecesario, utilizamos 'pcmpgtw' para compararlos con 4 máscaras que contengan 96, 288, 480 y 672 en Words. De esta forma, obtenemos máscaras que contienen unos en aquellos words que son mayores, y ceros donde son menores o iguales. Aplicando restas, pasamos de tener dos máscaras que nos indican (x i96) y (x i288) a tener una que nos indica (96 i= x i288) (Por dar un ejemplo). A estas máscaras les aplicamos la instrucción pandw junto con una máscara con el resultado correspondiente (en este caso 64) para obtener ese resultado en los words donde habían unos, y ceros en caso contrario. Sumando todas estas máscaras finales obtenemos el resultado final, en words. Finalmente, simple 'packusdw' nos transforma estos dos XMM con los resultados finales en words, en uno solo que tiene los 4 píxeles a guardar, en Bytes.

El uso de máscaras no sólo resultó ser más claro, sino que también demostró ser altamente eficiente en comparación con el branching excesivo y las cuentas de punto flotante.

**EXPERIMENTO SALTOS CONDICIONALES:** En este experimento corrimos el programa con el flag -O1, en comparación con el mismo programa sin los condicionales. Como explicamos en el punto 1.2, -O1 ofrece ciertas optimizaciones del código que reducen sustancialmente el 'branching', reemplazándolo en lo posible por instrucciones aritméticas. Por ello, ya esperábamos cierta diferencia a favor del programa sin condicionales. Sin embargo al correr los experimentos nos sorprendió que la diferencia era abismal: El código sin condicionales corre 3 veces más rápido que el optimizado. Esto deja en evidencia lo ineficiente que es utilizar saltos condicionales, incluso si lo optimizamos.

## 2.2. Experimentos

### Experimento 1.1)

a) Al hacer el `objdump` no solo se imprime la función `cropflip.c.c`, sino que también se imprimen varias funciones que utiliza GDB para hacer el debugging, tales como `debug_info`, `debug_abbrev`, `debug_aranges`, etc. También imprime `commentz eh_header`, que contienen información sobre el Linker y el compilador.

b) Las variables locales las almacena en memoria, haciendo `movs` manualmente en el stack frame. (Por ej, poniendo las variables en `[rbp-0x58]`, `[rbp-0x60]`, `[rbp-0x64]`).

c) Se podría optimizar el almacenamiento de las variables locales. Como el acceso a memoria es mucho más ineficiente que el acceso a los registros, sería más óptimo almacenar las variables en ellos.

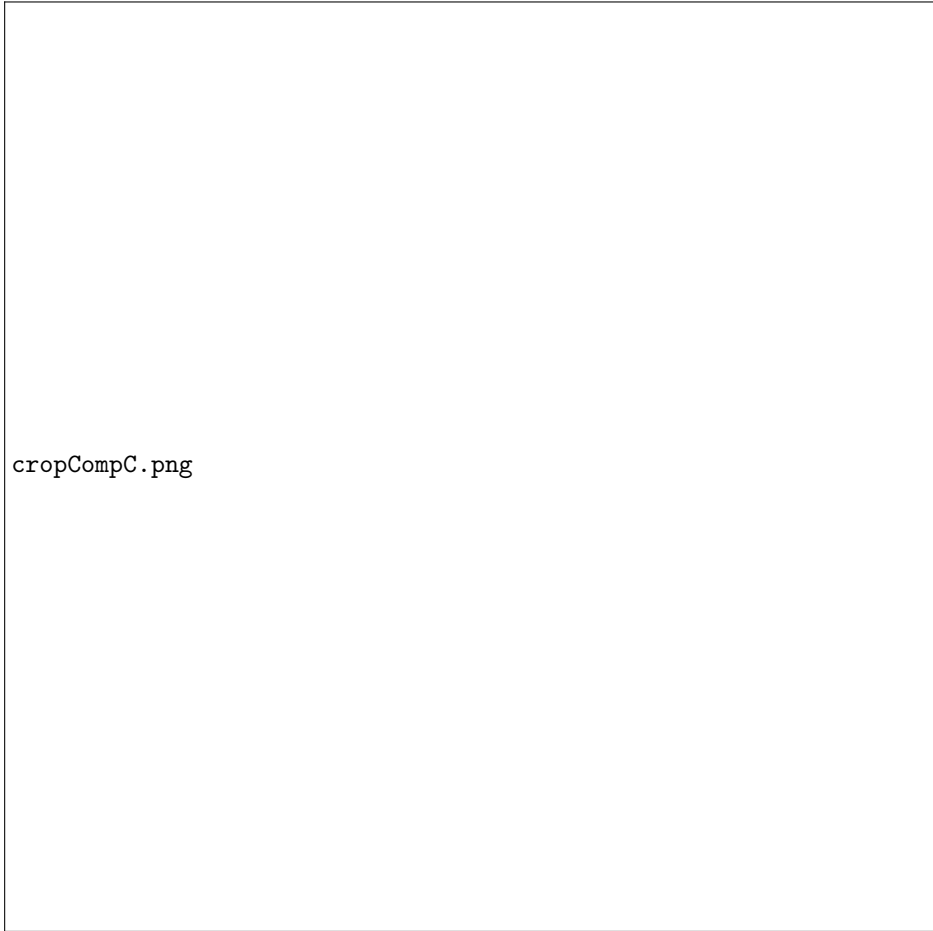
### Experimento 1.2)

a) `-O1` no reduce mucho el tiempo de compilación, y ejecuta una optimización "moderada". Se ve claramente que se reducen los accesos a memoria para las variables locales, y se reducen mucho las líneas de código.

b) Usando otros parámetros como `-O2` u `-O3` se hace la compilación con más optimizaciones cuanto más grande sea el número. También están `-Os` que optimiza el tamaño del código y `-Og` que optimiza el debugging.

c)

- **FDCE:** Hace eliminación de "dead code" (Código muerto), es decir, borra el código que consume recursos pero sus resultados nunca son utilizados.
- **FDSE:** Hace eliminación de "dead store" (Almacenamiento muerto), o sea, ignora aquellas variables que después no llegan a ser utilizadas.
- **-fif-conversion y -fif-conversion2:** Reemplazan condicionales por equivalentes aritméticos. Esto incluye funciones como movimientos condicionales, mínimos, máximos, función `abs`, entre otros. Luego de ver los resultados del experimento 3.1 (Saltos condicionales en el filtro Bandas) queda claro que esta optimización es extremadamente útil.



cropCompC.png

En el gráfico se nota la clara diferencia entre las corridas con optimizaciones y sin ellas. El criterio fue el mismo que aplicamos en los experimentos de más adelante (Haciendo 1000 iteraciones).

### Experimento 1.3)

En este experimento consideraremos diferentes criterios realizando siempre 10 mediciones, con outliers, sin ellos, y agregando programas en C++ que trabajen en simultáneo.

**a) Resultados:**

422861, 242136, 186857,  
194709, 234097, 221979,  
247912, 227976, 223245,  
200770.

**b) Resultados:**

289209, 302654, 258757,  
254428, 248527, 254606,  
260391, 338981, 373612,  
280937

**c)**

**10 mediciones**

Esperanza: 240254.2 Desvío standard: 67250.78758 Varianza: 4522668429.51111

**10 mediciones con el programa en simultáneo:**

Esperanza: 286210.2 Desvío standard: 41607.01932 Varianza: 1731144056.62222

**d) 10 mediciones sin 2 outliers:**

Esperanza: 224103 Desvío standard: 18595.78063 Varianza: 345803057.14286

**e)**

INSERT GRAFICOU HERE

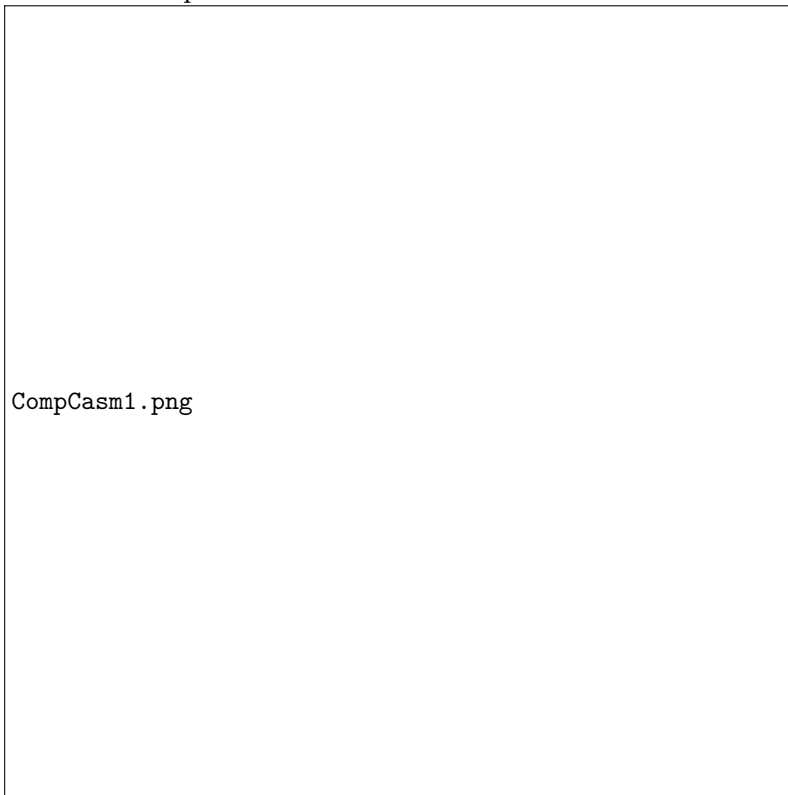
Luego de observar estas mediciones hemos notado una varianza demasiado alta (y, en consecuencia, una falta de confiabilidad en las mediciones), por lo tanto, luego de probar con distintos métodos de medición decidimos tomar el promedio de 1000 mediciones 10 veces, eliminar outliers y calcular luego nuevamente el promedio de las mediciones restantes.

### Experimento 1.4)

Este experimento comparará la performance de las implementaciones C y assembler que aplican el filtro cropflip (siguiendo el modelo de procesamiento secuencial y vectorial, respectivamente), probamos la implementación en C con 4 distintos flags de optimización.

- **asm-** 46671 ciclos
- **C O3-** 106740 ciclos
- **C O2-** 106859 ciclos
- **C O1-** 156148 ciclos
- **C O0-** 613445 ciclos

NOTA: las líneas pequeñas azules en el tope de las barras representan el desvío estándar del promedio de las mediciones que estamos mostrando.



Se observa en este experimento una notoria ventaja (en eficiencia temporal) del procesamiento vectorial de datos sobre el secuencial, incluso luego de haber aplicado las optimizaciones sobre el secuencial (que redujeron mucho el tiempo de procesamiento) la implementación de assembler pudo aplicar el filtro mucho más rápido y de manera más efectiva, mientras que entre las optimizaciones O2-O3 no hubo diferencia casi, indicando que no parece ser posible optimizar mucho más la implementación en C.

### Experimento 1.5) Cropflip - cpu vs bus de memoria

blablalbla pregrafico

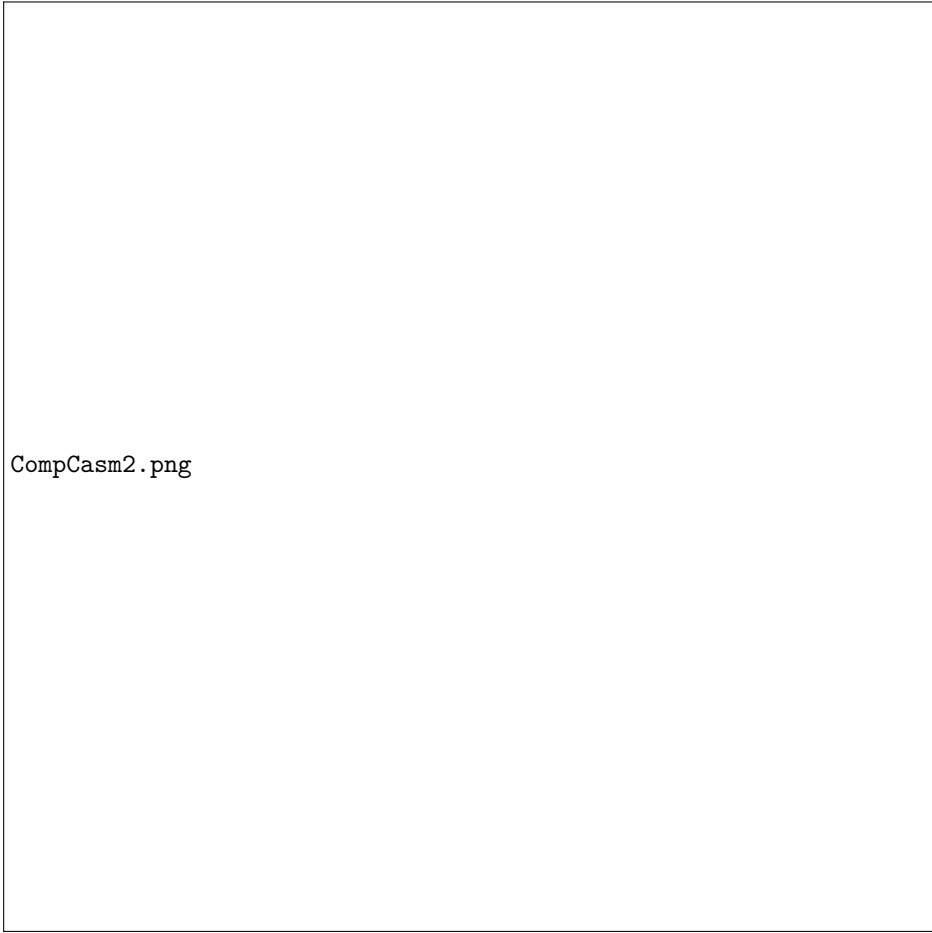
blablalbla postgrafico

### Experimento 2.1) Sierpinski - secuencial vs vectorial

Este experimento comparará la performance de las implementaciones C y assembler que aplican el filtro sierpinski (siguiendo el modelo de procesamiento secuencial y vectorial, respectivamente), probamos la implementación en C con 4 distintos flags de optimización.

- **asm-** 1618682 ciclos
- **C O3-** 4114385 ciclos
- **C O2-** 10599598 ciclos
- **C O1-** 17026398 ciclos
- **C O0-** 24125694 ciclos





CompCasm2.png

En este experimento la ventaja de usar SIMD siguió notándose con fuerza, mostrando una gran diferencia cuando es contrastado su tiempo de ejecución con el de las distintas optimizaciones de la implementación en C, si bien en este caso se notó que la optimización O3 tiene un gran impacto comparada con la O2.

### **Experimento 2.1) Sierpinski - cpu vs bus de memoria**

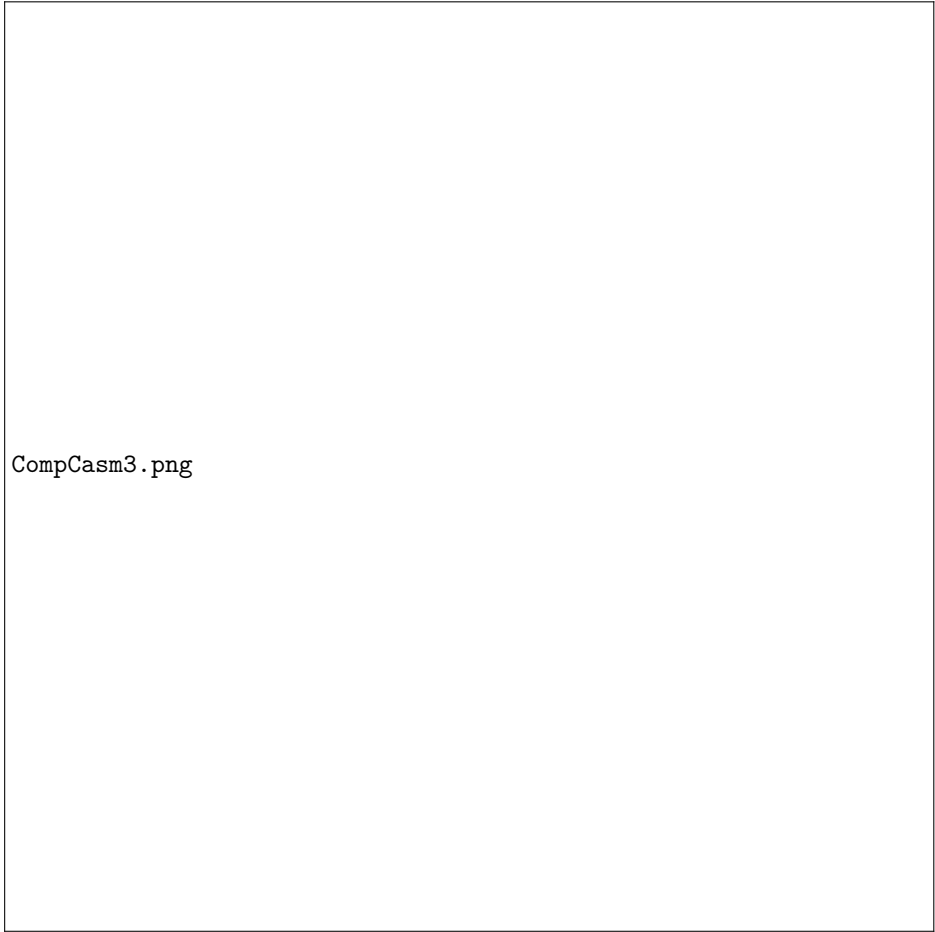
**Experimento 3.1) Bandas - saltos condicionales**

cmpIFS.png

### Experimento 3.2) Bandas - secuencial vs vectorial

Este experimento comparará la performance de las implementaciones C y assembler que aplican el filtro bandas (siguiendo el modelo de procesamiento secuencial y vectorial, respectivamente), probamos la implementación en C con 4 distintos flags de optimización.

- **asm-** 516916 ciclos
- **C O3-** 2205562 ciclos
- **C O2-** 2165530 ciclos
- **C O1-** 2214220 ciclos
- **C O0-** 6931576 ciclos

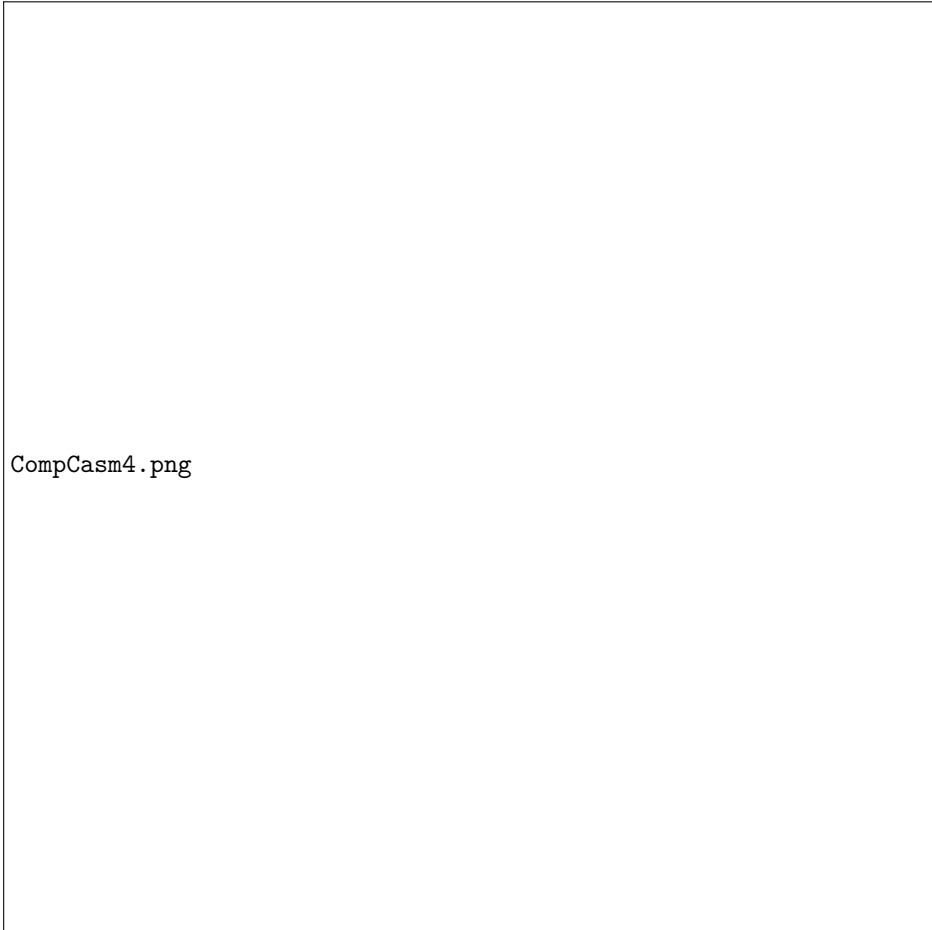


CompCasm3.png

### Experimento 4.1) Motion Blur - secuencial vs vectorial

Este experimento comparará la performance de las implementaciones C y assembler que aplican el filtro motion blur (siguiendo el modelo de procesamiento secuencial y vectorial, respectivamente), probamos la implementación en C con 4 distintos flags de optimización.

- **asm-** 2030949 ciclos
- **C O3-** 4698057 ciclos
- **C O2-** 4889967 ciclos
- **C O1-** 5513945 ciclos
- **C O0-** 17694922 ciclos



CompCasm4.png

En este experimento, nuevamente, la implementación de assembler mostró una clara diferencia (positiva) con las distintas optimizaciones de la de C, mientras que en C no se pudo optimizar el código mucho más allá de la primera optimización.

**Título del párrafo**    Bla bla bla bla. Esto se muestra en la figura ??.

## 3. Enunciado y solución

### 3.1. 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.

### 3.2. 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 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>1</sup>agregando este flag a `CCFLAGS64` en el `makefile`

<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 -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

## **4. Conclusiones y trabajo futuro**