DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Organización del Computador II

Segundo Cuatrimestre de 2014

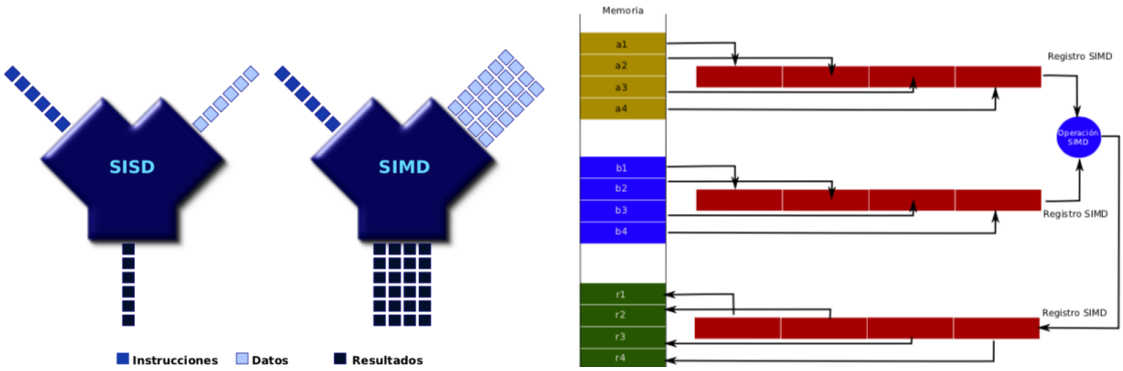
Integrante	LU	Correo electrónico
Joaquin Romera	183/16	joakromera@gmail.com
Santiago Tucci	682/16	stucci4@gmail.com
Carlos Soliz	406/12	rcarlos.cs@gmail.com

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

Índice

1. Introducción

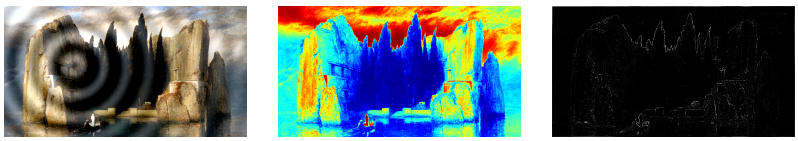
En el presente trabajo hicimos un primer acercamiento al modelo de ejecución SIMD (Single instruction Multiple Data). Diseñado como una extensión al set de instrucciones de la arquitectura x86 e introducido por Intel en 1999, esta técnica de paralelización puede mejorar notablemente la performance de un sistema en contextos de programación donde se deben aplicar algoritmos repetitivos sobre un mismo conjunto de datos. Esto lo hace particularmente útil para procesamientos multimedia -audio, video e imágenes- donde la reproducción en tiempo real es crítica. Gracias a las instrucciones SIMD podemos ejecutar una misma operación sobre muchos datos al mismo tiempo en una sola instrucción, contrario a realizarlo en un modelo exclusivamente SISD (single instruction single data). Como veremos a lo largo del trabajo su utilización introducirá notables mejoras en performance y eficiencia de nuestros algoritmos siempre que sea posible su aplicación.



1.1. Objetivos Generales

Además de la exploración del modelo SIMD se repasaron conceptos y técnicas de programación vectorizada en C y ASM dentro del campo de aplicación del procesamiento de imagenes. Para esto se llevó a cabo la implementación de los siguientes filtros para imágenes:

Imagen original | Blit | Monocromatizar



Ondas | Temperature | Edge

1.2. Metodología

La elaboración del trabajo se dividió en dos etapas. En primer lugar, se implementaron ambos filtros tanto en lenguaje C como en lenguaje ensamblador para la arquitectura x86-64 de Intel. En este último caso, se utilizaron las instrucciones SSE de dicha arquitectura, que aprovechan el ya mencionado modelo SIMD para procesar datos en forma paralela.

Una vez realizadas estas implementaciones, fueron sometidas a un proceso de comparación para extraer conclusiones acerca de su rendimiento. Con este fin, se experimentó con variaciones tanto en los datos de entrada como en detalles implementativos de los mismos algoritmos. De esta manera, se pudo recopilar datos sobre el comportamiento de cada implementación, y contrastar estos resultados con diversas hipótesis previamente elaboradas.

A continuación introducimos los filtros y sus respectivas implementaciones, luego describimos los tests realizados y los resultados obtenidos, y finalmente a partir de estos datos otorgamos algunas conclusiones sobre la experiencia realizada.

2. Filtros

2.1. Blit

2.1.1. Descripción

Esta operación recibe dos imagenes -original y blit- y genera una nueva combinando ambas. La combinación se realiza considerando un determinado color del blit como transparente para que en el resultado final algunos pixeles del blit queden por encima de la imagen original. En el lugar de aquellos que no son interpretados como transparentes quedan los pixeles del original. La imagen obtenida es una superposición de ambas, teniendo transparencias en el blit siempre que el color del pixel sea igual a (255,0,255).

A pedido de la cátedra la imagen **blit** será una foto de Perón. La misma será ubicada en la esquina superior derecha de la imagen, además la imagen original deberá tener un tamaño mayor a 89 píxeles de ancho por 128 de alto para poder ubicar el blit. Como resultado tendremos una imagen *Peronizada* cuando se aplique la siguiente función para cada píxel p en la imagen de Perón (blit):

$$dst(p) = \begin{cases} src(p) & \text{si } blit(p) \text{ es de color magenta, es decir sus colores son } (255, 0, 255) \\ blit(p) & \text{si no} \end{cases}$$

2.1.2. Implementación C

Nuestra implementación en C consiste en dos etapas: primero copiamos los píxeles la imagen original en la imagen destino, luego copiamos los píxeles del blit siempre que los mismos no tengan el color (255,0,255). Como Perón debe estar arriba a la derecha de la imagen final, restamos el tamaño de su imagen al de la original en las dimensiones correspondientes. A continuación el pseudocódigo describe el algoritmo:

Algorithm 1 $blit(I_{src}, I_{dst}, I_{blit})$

for all y:=0 **to** Height(I_{src}) **do**
  for all x:=0 **to** Width(I_{src}) **do**
    $pixel \leftarrow I_{src}(x, y)$
    $I_{dst}(x, y) \leftarrow pixel$
  end for
end for
$Int\ bh \leftarrow$ Height(I_{src}) $-$ Height(I_{blit})
$Int\ bw \leftarrow$ Width(I_{src}) $-$ Width(I_{blit})
for all y:=0 **to** Height(I_{blit}) **do**
  for all x:=0 **to** Width(I_{blit}) **do**
    $pixel \leftarrow I_{blit}(x, y)$
    $Int\ r \leftarrow Red(pixel)$
    $Int\ g \leftarrow Green(pixel)$
    $Int\ b \leftarrow Blue(pixel)$
    if $\neg(r = 255$ **and** $g = 0$ **and** $b = 255)$ **then**
      $I_{dst}(x + bw, y + bh) = I_{blit}(x + bw, y + bh)$
    end if
  end for
end for

2.1.3. Implementación ASM

En este Filtro Blit procesamos de a 4 pixeles, Tenemos dos ciclos en assembler, uno que recorre las columnas de la imagen y otra q recorre las fila. Cuando itera la fila es en ese momento q toman los 4 pixeles de I_{src} y lo copiamos en I. Hay caso especial cuando nos encontramos en la fila "Height(I_{src}) − Height(I_{blit})ż columna "Width(I_{src}) − Width(I_{blit})", es en ese caso que tenemos o no aplicar el blit segun la formula explicada arriba. Abajo detallaremos esa parte importante del código ASSEMBLER q desarrolla el blit con un ejemplo.

- En los registro **xmm0,xnm14** tenemos la copia de los cuatro píxeles q levantamos de memoria, uno es I_{src} y el otro I_{blit} respectivamente. Y en **xnm2, xmm15** las mascaras q utilizamos para la comparación y operaciones lógicas.

a	b	g	r	a	b	g	r	a	b	g	r	a	b	g	r

xmm0 ← I_{src}

A	B	G	R	A	B	G	R	A	B	G	R	A	B	G	R

xmm14 ← I_{blit}

255	255	0	255	255	255	0	255	255	255	0	255	255	255	0	255

Mascara Magenta xmm15 (maskMagenta)

00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h	00h

Mascara en xmm2 (maskCeros)

- Mascara para filtrar píxeles valor magenta. A suponemos que hay dos píxeles color magenta en I_{blit} a modo de ejemplo en los píxeles 1 y 3(siguiendo el order de pixel_15,...,pixel_0). A esta mascara la guardamos en xmm12 y xmm15.

0xFF	0xFF	0xFF	0xFF	0x00	0x00	0x00	0x00	0xFF	0xFF	0xFF	0xFF	0x00	0x00	0x00

xmm12,xmm15 ← pcmpeqd xmm15, xmm14

- Mascara para filtrar píxeles que No son magenta.

0x00	0x00	0x00	0x00	0xFF	0xFF	0xFF	0xFF	0x00	0x00	0x00	0x00	0xFF	0xFF	0xFF

xmm15 ← pcmpeqd xmm15, xmm13

- Me quedo con los valores de I_{src} que tengo q poner en la imagen I_{dst}.

A	B	G	R	0x00	0x00	0x00	0x00	A	B	G	R	0x00	0x00	0x00	0x00

xmm12 ← pand xmm12, xmm0

- Me quedo con los vamlores de I_{blit} que tengo q poner en la imagen I_{dst}.

0x00	0x00	0x00	0x00	a	b	g	r	0x00	0x00	0x00	0x00	a	b	g	r

xmm15 ← pand xmm15, xmm14

- Junto todos los valores en xmm15.

A	B	G	R	a	b	g	r	A	B	G	R	a	b	g	r

xmm15 ← por xmm15, xmm12

- Por ultimo resta guardar estos 4 pixeles(xmm15) en I_{dst}.

Para mas detalles dejamos un extracto del código ASM:

```
maskMagenta: db 255  ,0, 255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 255, 255
maskCero: db 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

section .text

blit_asm:
...
movdqu xmm0, [rdi]; xmm0=|A B G R|A B G R|A B G R|A B G R|; img
movdqu xmm14, [r15]; xmm14=|a b g r|a b g r|a b g r|a b g r|; blit
   ; filtro los valores color magenta
pcmpeqd xmm15, xmm14   ; xmm15=|FF FF FF FF|00 00 00 00|FF FF FF FF|00 00 00 00|
movdqu xmm12, xmm15    ; xmm12= xmm15 paso la mascara a xmm12
   ;filtro los valores que no son magenta
pcmpeqd xmm15, xmm13   ; xmm15=|00 00 00 00|FF FF FF FF|00 00 00 00|FF FF FF FF|
   ;Me quedo con los velores de xmm0 que tengo que poner en la imagen
pand xmm12, xmm0       ; xmm2=|A B G R|00 00 00 00|A B G R|00 00 00 00|
   ;Me quedo con los valores de blit que tengo que poner en la imagen
pand xmm15, xmm14      ; xmm15=|00 00 00 00|a b g r|00 00 00 00|a b g r|
   ; Junto los dos valores en xmm15
por xmm15, xmm12     ;xmm15=|A B G R|a b g r|A B G R|a b g r|
movdqu [rsi], xmm15    ; Copio todo a dst
movdqu xmm15, [maskMagenta]
movdqu xmm13, [maskCero]
.....
```

2.2. Monocromatizar

2.2.1. Descripción

El objetivo de este filtro es convertir una imagen color a una escala de grises. A diferencia de otros filtros, los pixeles de la imagen de salida estarán compuestos por un solo byte, que representará la intensidad de la luz. El criterio para obtener este valor es tomar el máximo entre los componentes R, G, B originales de cada pixel y aplicarlo al correspondiente en la imagen destino. Para esto aplicamos, a todos los pixeles de la imagen original, la siguiente función:

$$I_{out}(p) = max(R, G, B)$$

2.2.2. Implementación C

A continuación el pseudocódigo de la implementación en C:

Algorithm 2 $monocromatizar(I_{src}, I_{dst})$

```
1: for all y:=0 to Height(I_src) do
2:   for all x:=0 to Width(I_src) do
3:     pixel ← I_src(x, y)
4:     Int r ← Red(pixel)
5:     Int g ← Green(pixel)
6:     Int b ← Blue(pixel)
7:     Int max ← max(r, max(g, b))
8:     pixel ← DevolverPixel(r, g, b)
9:     I_dst(x, y) ← pixel
10:  end for
11: end for
```

2.2.3. Implementación ASM

En este filtro procesamos de a 4 píxeles en cada iteración usando las instrucciones **SSE** de la arquitectura. Por cada iteración se realizaron los calculos del máximo de cada píxel y este se guardó en la imagen de salida.

- En los registro **xmm0,xnm4, xmm5** tenemos la copia de los cuatro pixeles que levantamos de memoria. Y en **xnm1, xmm2, xmm3** las máscaras q utilizamos para los shuffles que utilizaremos para permutar los componentes.

a	b	g	r	a	b	g	r	a	b	g	r	a	b	g	r

xmm0, xmm4, xmm5

15	13	13	13	11	9	9	9	7	5	5	5	3	1	1	1

Mascara en xmm1 (mask1)

15	14	14	14	11	10	10	10	7	6	6	6	3	2	2	2

Mascara en xmm2 (mask2)

15	12	12	12	11	8	8	8	7	4	4	4	3	0	0	0

Mascara en xmm3(mask3)

- Realizamos el **shuffle xmm4, xmm1** que nos coloca el componente **g** en las posiciones que podemos observar en el registro xmm4. Y luego calculamos el máximo con la instrucción **pmaxub**.

a	g	g	g	a	g	g	g	a	g	g	g	a	g	g	g

xmm4 ← pshufb xmm4, xmm1

a	.	.	max(g,r)	a	.	.	max(g,r)	a	.	.	max(g,r)	a	.	.	max(g,r)

xmm4 ← pmaxub xmm4, xmm0

- Realizamos el mismo procedimiento en este paso. Luego nos quedaría el valor del Máximo (ver imagen).

a	b	b	b	a	b	b	b	a	b	b	b	a	b	b	b

xmm5 ← pshufb xmm5, xmm2

a	.	.	max(g,r,b)	a	.	.	max(g,r,b)	a	.	.	max(g,r,b)	a	.	.	max(g,r,b)

xmm4 ← pmaxub xmm4, xmm5

- Solo resta copiar los máximos. Para esto utilizaremos el Shuffle junto con la máscara contenida en xmm3.

a	max	max	max	a	max	max	max	a	max	max	max	a	max	max	max

xmm5 ← pshufb xmm4, xmm3

Por último resta copiar esto a memoria, brindamos el código ASM de este paso.

```
section .data
mask1: db 1, 1, 1, 3, 5, 5, 5, 7, 9, 9, 9, 11, 13, 13, 13, 15
mask2: db 2, 2, 2, 3, 6, 6, 6, 7, 10, 10, 10, 11, 14, 14, 14, 15
mask3: db 0, 0, 0, 3, 4, 4, 4, 7, 8, 8, 8, 11, 12, 12, 12, 15

section .text

monocromatizar_inf_asm:
.........
movdqu xmm0, [rdi]; xmm0=|a b g r|a b g r|a b g r|a b g r|
movdqu xmm4, xmm0; xmm4= xmm0
movdqu xmm5, xmm0; xmm5= xmm0

pshufb xmm4, xmm1; xmm4=|a g g g|a g g g|a g g g|a g g g|
pmaxub xmm4, xmm0; xmm4=|a . . max(g,r)|a . . max(g,r)|a . . max(g,r)|a . . max(g,r)|;primer maximo

pshufb xmm5, xmm2; xmm5=|a b b b|a b b b|a b b b|a b b b|
pmaxub xmm4, xmm5; xmm4=|a . . max(g,r,b)|a . . max(g,r,b)|a . . max(g,r,b)|;maximo de maximos
 ; xmm4=|a . . max|a . . max|a . . max|a . . max|
pshufb xmm4, xmm3; xmm4=|a max max max|a max max max|a max max max|a max max max|

movdqu [rsi], xmm4; [rsi]= xmm4
......
```

2.3. Ondas

Este filtro combina la imagen original con una imagen de ondas, dando tonos más oscuros y mas claros en forma de onda. Estas se centran en un punto de la imagen -pasado como parámetro- y se expanden hacia sus bordes de manera concéntrica. Para ello aplicamos implementamos el pseudocódigo del algoritmo otorgado por la cátedra, en el cual a cada pixel de la imagen se le suma un valor de profundidad basado en su distancia al centro de las ondas.

2.3.1. Implementación C

Nuestra implementación en C es una traducción iterativa del mencionado algoritmo propuesto por la cátedra en el siguiente pseudocódigo:

Algorithm 3 $ondas(I_{src}, I_{dst}, x_0, y_0)$

for all pixel ubicado en la posici'on $(\mathbf{x}, \mathbf{y})$ **do**

$d_x \leftarrow x - x_0$

$d_y \leftarrow y - y_0$

$d_{xy} \leftarrow \sqrt{d_x^2 + d_y^2}$

$r \leftarrow \frac{(d_{xy} - RADIUS)}{WAVELENGTH}$

$a \leftarrow \frac{1}{1+(\frac{r}{TRAINWIDTH})^2}$

$t \leftarrow (r - floor(r)) \cdot 2 \cdot \pi - \pi$

$prof \leftarrow a \cdot (t - \frac{t^3}{6} + \frac{t^5}{120} - \frac{t^7}{5040})$

$pixel = prof \cdot 64 + I_{src}(x, y)$

$I_{dst}(x, y) = saturar(pixel)$

end for

- x_0 e y_0 representan la posici'on donde est'a centrada la onda,
- $RADIUS$, $WAVELENGTH$ y $TRAINWIDTH$ son constantes que definen la forma de la onda y
- $saturar(x)$ es una funci'on que retorna 0 si x es menor 0, 255 si es mayor a 255 y x en cualquier otro caso.

2.3.2. Implementación ASM

2.4. Temperature

El filtro temperatura toma una imagen fuente y genera un efecto que simula un mapa de calor. Dicho efecto lo consigue tomando los tres componentes de color de cada pixel y promediándolos. Luego le asigna un valor que depende de este promedio *t*.

$$\mathsf{t}_{(i,j)} = \lfloor (\mathsf{src}.r_{(i,j)} + \mathsf{src}.g_{(i,j)} + \mathsf{src}.b_{(i,j)})/3 \rfloor$$

Finalmente el color en la imagen destino se determina en función de la temperatura *t* conforme al siguiente mapeo:

$$\mathsf{dst}_{(i,j)} < r, g, b >= \begin{cases} < 0, 0, 128 + t \cdot 4 > & \text{si } t < 32 \\ < 0, (t-32) \cdot 4, 255 > & \text{si } 32 \leq t < 96 \\ < (t-96) \cdot 4, 255, 255 - (t-96) \cdot 4 > & \text{si } 96 \leq t < 160 \\ < 255, 255 - (t-160) \cdot 4, 0 > & \text{si } 160 \leq t < 224 \\ < 255 - (t-224) \cdot 4, 0, 0 > & \text{si no} \end{cases}$$

2.4.1. Implementación C

El algoritmo implementado en lenguaje C recorre la imagen iterativamente. Por cada pixel en la imagen original calcula el promedio de sus componentes RGB (línea 4) y según este valor se guarda en la imagen destino el calculo correspondiente a la temperatura (líneas 5 a 15). A continuación el pseudocódigo:

Algorithm 4 $temperature(I_{src}, I_{dst})$

```
1:  for all y:=0 to Height(I_src) do
2:    for all x:=0 to Width(I_src) do
3:      pixel ← I_src(x, y)
4:      Nat t ← ⌊(Red(pixel)+Green(pixel)+Blue(pixel))/3⌋
5:      if t < 32 then
6:        I_dst(x, y) ← DevolverPixel(0, 0, 128 + t · 4)
7:      else if 32 ≤ t < 96 then
8:        I_dst(x, y) ← DevolverPixel(0, (t − 32) · 4, 255)
9:      else if 96 ≤ t < 160 then
10:       I_dst(x, y) ← DevolverPixel((t − 96) · 4, 255, 255 − (t − 96) · 4)
11:     else if 160 ≤ t < 224 then
12:       I_dst(x, y) ← DevolverPixel(255, 255 − (t − 160) · 4, 0)
13:     else
14:       I_dst(x, y) ← DevolverPixel(255 − (t − 224) · 4, 0, 0)
15:     end if
16:   end for
17: end for
```

2.4.2. Implementación ASM

2.5. Edge

El filtro EDGE devuelve una imagen con los bordes de otra imagen original. Esto se logra observando los pixeles donde la intensidad de la imagen cambia de forma abrupta, lo cual puede ser logrado buscando saltos en una función de intensidades. Esta idea de detectar los bordes fue implementada a través del operador de Laplace, cuya matriz es:

$$M = \begin{pmatrix} 0{,}5 & 1 & 0{,}5 \\ 1 & -6 & 1 \\ 0{,}5 & 1 & 0{,}5 \end{pmatrix}$$

La forma de utilizarla es posicionando uno a uno cada pixel en el centro de la matriz y realizando el siguiente cálculo:

$$dst(x,y) = \sum_{k=0}^{2}\sum_{l=0}^{2} src(x+k-1, y+l-1) * M(k,l)$$

Al igual que monocromatizar, EDGE opera sobre imágenes en escala de grises (1 componente de color por píxel). Como no se pueden procesar los bordes con la función anterior se aplica la siguiente función: $dst(x,y) = src(x,y)$.

2.5.1. Implementación C

Ignorando la primera y la última fila al igual que el primer y el último pixel de cada fila, recorrimos iterativamente todos los pixeles de la imagen realizando el siguiente proceso: por cada pixel calculamos todas las sumas parciales de la función que resulta de la aplicación de la matriz de Laplace, luego todas estas sumas fueron almacenadas en una variable auxiliar (*edge*). Si el valor de *edge* requería ser saturado se lo restringió a los valores 0/255 y finalmente se guardó el resultado en el pixel correspondiente de la imagen destino.

Algorithm 5 $edge(I_{src}, I_{dst})$

1: **for all** y:=1 **to** Height($I_{src}-1$) **do**
2: **for all** x:=1 **to** Width(I_{src})-1 **do**
3: $m_{0,0} \leftarrow I_{src}(x-1, y-1) * 0{,}5$
4: $m_{0,1} \leftarrow I_{src}(x-1, y) * 1$
5: $m_{0,2} \leftarrow I_{src}(x-1, y+1) * 0{,}5$
6: $m_{1,0} \leftarrow I_{src}(x, y-1) * 1$
7: $m_{1,1} \leftarrow I_{src}(x, y) * (-6)$
8: $m_{1,2} \leftarrow I_{src}(x, y+1) * 1$
9: $m_{2,0} \leftarrow I_{src}(x+1, y-1) * 0{,}5$
10: $m_{2,1} \leftarrow I_{src}(x+1, y) * 1$
11: $m_{2,2} \leftarrow I_{src}(x+1, y+1) * 0{,}5$
12: $edge \leftarrow m_{0,0} + m_{0,1} + m_{0,2} + m_{1,0} + m_{1,1} + m_{1,2} + m_{2,0} + m_{2,1} + m_{2,2}$
13: $I_{dst}(x,y) \leftarrow Saturar(edge)$
14: **end for**
15: **end for**

2.5.2. Implementación ASM

3. Mediciones y rendimiento

La forma de medir el rendimiento de nuestras implementaciones se realizará por medio de la toma de tiempos de ejecución del algoritmo (sea este el codigo version asembler o el codigo C). Como los tiempos de ejecución son muy pequeños, se utilizará uno de los contadores de performance que posee el procesador. La instrucción de assembler rdtsc permite obtener el valor del Time Stamp Counter (TSC) del procesador. Este registro se incrementa en uno con cada ciclo del procesador. Obteniendo la diferencia entre los contadores antes y después de la llamada a la función, podemos obtener la cantidad de ciclos de esa ejecución. Esta cantidad de ciclos no es siempre igual entre invocaciones de la función, ya que este registro es global del procesador y se ve afectado por una serie de factores.

Existen principalmente dos problemáticas a solucionar: 1). La ejecución puede ser interrumpida por el scheduler para realizar un cambio de contexto, esto implicará contar muchos más ciclos (outliers) que si nuestra función se ejecutara sin interrupciones.

2). Los procesadores modernos varían su frecuencia de reloj, por lo que la forma de medir ciclos cambiará dependiendo del estado del procesador.

solucion 1): Para evitar el problema de los ciclos outliers lo que hicimos fue,

Paso 1: En nuestro caso hicimos 100 veces la medición de tiempo de nuestro algoritmo y guardarlo en un contenedor(podría ser un arreglo,lista, conjunto, diccionario,.., etc).

Paso 2: Sacar la media, tambien conocido como promedio, ejemplito:

$$Prom = \frac{x_1+x_2+...+x_{100}}{100}$$

Donde x_i es la medición de tiempo de la medición número i, con $1 \leq i \leq 100$

Paso 3: Calculamos la varianza:

$$Varianza = \sigma^2 = \frac{(x_1-Prom)+(x_2-prom)+...+(x_{100}-prom)}{100}$$

Paso 4: Calculamos el desvio estandar, $\sigma = \sqrt{Varianza}$

Paso 5: Utilizando el desvio estandar y el promedio, puedo ver que medición es ”buena” y cual no. Más formalmente una medicion es ”buena”si cumple:

$$Prom - \sigma \leq x_i \leq Prom + \sigma.$$

Luego sumando las todas las mediciones ”buenas”
y dividiendalas por la cantidad de mediciones buenas, obtengo el ”promedio bueno”. Con esto amortiguaria la cantidad de outliers de mis mediciones.

Observar: Todo lo anterior sirve también para mas de 100 mediciones.

Solucion 2): La solución que planteamos para esto fue, ejecutar sólamente el algoritmo. Con esto queremos decir que la ejecución estara en el nivel más alto de privilegio de ejecución. Esto lo hacemos metiendonos en el sistema operativo (en este caso ubuntu 14.04), tocando el monitor de sistema para darle privilegio a la ejecución. Tambien evitamos interumpir la maquina de forma mecanica(osea humana).

4. Conclusiones y trabajo futuro

A partir de los experimentos realizados en este trabajo, se pudo llegar a la conclusión de que las ventajas que brinda el paradigma **SIMD** a la hora de implementar programas que realicen operaciones altamente paralelizables, como el procesamiento de imágenes, son verdaderamente significativas. Esto queda reflejado en la gran brecha de rendimiento que se observa entre las implementaciones realizadas con dicho paradigma y las que utilizan el lenguaje de programación C.

Esto siempre debe contraponerse a otro hecho que se hizo presente durante el proceso de implementación: realizar un programa en lenguaje ensamblador resulta, por lo general, considerablemente más difícil que hacerlo en un lenguaje de más alto nivel. El código resultante es menos legible, es más sencillo cometer errores y el proceso de *debugging* se vuelve considerablemente más arduo. Por eso es importante analizar de antemano las características del contexto particular de aplicación, para poder decidir si este esfuerzo adicional realmente vale la pena.

Ahondando en los detalles más técnicos de la implementación, se intentó la realización de una optimización manual dentro del código ensamblador, sin obtener resultados destacables. La razón de esto es que estructura del código dificultaba ampliamente la realización de dicha modificación. Realizar la optimización de manera efectiva habría implicado modificar gran parte del código ya armado, con un costo casi equivalente al de empezarlo de nuevo con un enfoque distinto; esto es consecuencia de la ya mencionada dificultad que presenta mantener el código programado en este lenguaje.