DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Grupo Smooth criminal

Organización del Computador II

DC - UBA

Trabajo Práctico N°2

Integrante	LU	Correo electrónico
Horacio Avendaño-Lucas	129/14	horacio.avendanio.lucas@gmail.com
Brian Goldstein	27/14	brai.goldstein@gmail.com
Maximiliano Paz	251/14	m4xileon@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) 4576-3300
http://www.exactas.uba.ar

Índice

Resumen

En el presente trabajo, se lleva a cabo el estudio del conjunto de instrucciones *SSE* (*Streaming SIMD Extensions*), los cuales procesan múltiples datos con una sola instrucción. Para poder ver estas instrucciones en acción, se trabaja con filtros que procesan imágenes, a saber: *Blur Gaussiano* y *Diferecia de imágenes*. Para ello, se implementan estos algoritmos y, luego, se los útiliza para la realización de experimentos.

En la *sección 1*, se describen los aspectos teóricos relacionados con este trabajo. En la *sección 2*, se comenta en detalle cómo se implementaron los diferentes algoritmos descritos anteriormente y, además, se plantean las hipótesis que se contrastarán en la experimentación. En la *sección 3*, se detallan las hiótesis y los resultados obtenidos en la experimentación. Y, finalmente, en la *sección 4*, se da a conocer nuestras conclusiones.

1. Introducción

En esta sección, se proporciona una serie de definiciones que serán utlizados a lo largo de este trabajo.

1.1. Imagen

Se define una imagen como un conjunto ordenado de *píxeles* los cuales contienen la información sobre el color que tiene dicha imagen en una posición determinada. En función de esta representación, tenemos la representación RGB, o *formato RGB*, el cual consiste en una terna de valores que describen la intensidad de los colores rojo (R), verde (G) y azul (B). Cada valor de la terna, también denominado *canal*, esta representado por un rango de valores enteros que van de 0 a 255, generalmente.

1.2. Filtros

Diferencia de imágenes

Este filtro compara dos imágenes de igual tamaño y devuelve como resultado una tercera con la diferencia que estas dos imágenes presentan por cada pixel en escala de grises.

(a) Imagen 1 (b) Imagen 2 (c) Diferencia

Figura 1.1: Ejemplo de uso del filtro para detección de primer plano.

Sean I_1 e I_2 las imágenes que se comparan. Para obtener la tercera imagen, O, el cálculo que se debe realizar por pixel es el siguiente:

$$O[i,j,k] = \| I_1[i,j] - I_2[i,j] \|_\infty \quad \forall k = R,G,B$$

En otras palabras, se toma el componente de mayor valor, en valor absoluto, entre los canales del pixel resultante de la diferencia y asignarlo a cada canal del pixel de la imagen de salida.

Blur gaussiano

Este filtro aplica un efecto de desenfoque sobre la imagen. Para ello, realiza un cálculo sobre cada pixel, el cual requiere información de sus píxeles vecinos. La cantidad de vecinos que se debe considerar esta determinado por uno de los parámetros al que se denominará *radio*. Otro de los parámetros que este cálculo considera es denominado *desvío*, el cual se utiliza, junto con el radio, para la determinar los valores que se pueden obtener de una función gaussiana.

(a) Imagen original (b) Con Blur Gaussiano

Figura 1.2: Filtro Blur Gaussiano sobre la imagen de Lena. (σ= 5, r = 15)

Concretamente, sean I y O las imágenes de entrada y salida respectivamente y, además, sean r es el radio y σ es el desvío, entonces el cálculo a realizar por pixel es el siguiente:

$$O[i,j,k] = \sum_{x=-r}^{r} \sum_{y=-r}^{r} I[i+x, j+y, k]\ K[r-x, r-y] \quad \forall k = R,G,B$$

donde K, denominada matriz de convolución o *kernel*, se define como:

$$K[i,j] = G_\sigma(r-i, r-j)$$
$$G_\sigma(x,y) = \frac{1}{2\pi\sigma^2}\ e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Nótese que la matriz K contiene en cada una de sus posiciones un valor de la función gaussiana G_σ.

2. Desarrollo

En esta sección, se decriben los algoritmos de los filtros implementados. Estos filtros se implementarán en *assembler* y *C*.

Algo para tener en cuenta a lo largo de esta sección es que el pixel de la imagen que procesamos en cada filtro esta determinado por 4 enteros sin signo cuyo rango de valores esta comprendido en un byte (de 0 a 2^8-1). Como resultado, un pixel tiene un tamaño final de 4 bytes. Los primeros 3 componentes corresponden a los canales azul (B), verde (G) y rojo (R), en ese orden y el último componente de dicho pixel corresponde a la transparencia (A).

2.1. Implementaciones

A continuación, se decriben las implementaciones de los filtros desarrollados en *assembler* y *C*. En los filtros desarrollados en *assembler* la premisa es utlizar al máximo las instrucciones *SSE*. Como aclaración previa, se debe mencionar que en todos los casos se determina el valor de la transparencia de un pixel (el cuarto componente) con el valor 255.

2.1.1. Diferencia de imágenes

Implementación en *C*

Esta implementación es bien sencilla. Se recorre simultáneamente por fila cada imagen de entrada y, en cada fila, se recorre de a un pixel para realizar el cálculo correpondiente. De modo que, el proceso realiza 2 ciclos anidados. El cálculo que se realiza consiste en obtener la resta vectorial de ambos píxeles y, acto seguido, tomar la norma infinito del vector resultante. Este valor será, entonces, la intensidad en escala de grises de la imagen resultante.

Para obtener la norma infinito, se determina el máximo valor entre los componententes del vector resultante, en valor absoluto. Con este fin, para obtener el máximo entre 3 valores, se hace uso de saltos condicionales y comparaciones. Para obtener el valor absoluto, se utiliza una función `abs` de la librería `<stdlib.h>` de *C*, la cual hace uso, también, de saltos condicionales y comparaciones. El pseudocódigo de ambos procesos es el siguiente:

Data: enteros n
Result: entero m

if $n < 0$ **then**
  $m \leftarrow n$
else
  $m \leftarrow -n$
end
return m

Algoritmo 1: Algoritmo para determinar el valor absoluto.

Data: enteros *red*, *green*, *blue*
Result: entero *max*

if $red \geq green$ **then**
  if $red \geq blue$ **then**
    $max \leftarrow red$
  else
    $max \leftarrow blue$
  end
else
  if $green \geq blue$ **then**
    $max \leftarrow green$
  else
    $max \leftarrow blue$
  end
end
return max

Algoritmo 2: Algoritmo para determinar el máximo entre 3 componente.

Implementación en *ASM*

En esta implementación el procedimiento es de algún modo análogo al anterior, con la salvedad de que se trabaja con cuatro pixeles en paralelo haciendo uso del modelo *SIMD*. Como consecuencia, se recorren las imágenes de entrada leyendo de a cuatro píxeles y guardando esta información en un registro XMM. También, se procesa todo dentro de un ciclo, a diferencia de la implementación en *C* donde se realiza en 2 ciclos. lectura por fila y luego por pixel. Además, no se hace uso de saltos condicionales en los proceso donde obtenemos el valor absoluto y el máximo.

Además, el cálculo de la norma infinito también es diferente. Se hacen uso de varias instrucciones *SSE*. Entre las más destacadas en este algoritmo se encuentra la instrucción PMAXUSB, el cual determina el máximo valor entre los correspondientes bytes sin signo de dos registros XMM, es decir: dados dos registros XMM se compara el byte 0 de ambos y se obtiene el máximo para depositarlo en el byte 0 del registro XMM destino, repitiendo lo mismo para los pares de bytes restantes. Además, se hace uso de la instrucción PSUBUSB, resta con saturación por byte sin signo, para realizar la resta vectorial. La manera en que se obtiene la norma infinito es entonces:

1. Dados dos registros XMM1 y XMM2 donde se hallan los 4 píxeles de las imágenes de entrada, la primera y la segunda respectivamente, se obtiene el máximo por pixel, por medio de la instrucción PMAXUSB, guardando esa información en 2 registros XMM, por ejemplo: XMM3 y XMM4.
2. Se hacen las siguientes restas por medio de la instrucción PSUBUSB: XMM3-XMM1 y XMM4-XMM2. Se registran ammbos resultados en XMM3 y XMM4 respectivamente.
3. Se realiza la operación lógica *OR* entre los registros XMM3 y XMM4 y se guarda este resultado en otro resgitro, por ejemplo: XMM0. En este punto ya tenemos la resta vectorial con los componentes en valor absoluto.
4. Se copia lo almacenado en XMM0 en tres registros similares, por ejemplo: XMM1, XMM2 y XMM3. En esos 3 registros rotamos los bytes correspondientes a los canales rojo, verde y azul de cada pixel. Es decir, si se tiene el siguiente formato en XMM0:

   ```
   XMMO = | A | R | G | B |
   ```

   se tendrá en los otros registros, por ejemplo, el siguiente formato:

```
XMM1 = | A | R | G | B |
XMM2 = | A | G | B | R |
XMM3 = | A | B | R | G |
```

5. Se determina el máximo entre los registros XMM1, XMM2 y XMM3 por medio de la instrución PMAXUB guardando el resultado en XMM0, por ejemplo. Así, obtenemos el valor que se quería en XMM0 en los primeros 3 bytes de cada *double word* de ese registro.

A continuación se expone un ejemplo de aplicación paso a paso. Supongamos, para hacer mas sencillo la explicación que nos interesa el primer pixel de cada resgistro XMM. Además, no es de interés la transparencia. Supongamos, entonces, que tenemos los siguientes 4 pixeles leídos de las imágenes de entrada:

```
XMM1 = |...| - | 9 | 5 | 3 |
XMM2 = |...| - | 1 | 7 | 8 |
```

Entonces, aplicando el procedimiento antes explicado se obtiene que:

1. `PMAXUB(XMM1,XMM2) = |...| - | 9 | 7 | 8 |`

   `entonces:`

   ```
   XMM3 = |...| - | 9 | 7 | 8 |
   XMM4 = |...| - | 9 | 7 | 8 |
   ```

2. ```
   XMM3 = XMM3 - XMM1 = |...| - | 0 | 2 | 5 |
   XMM4 = XMM4 - XMM2 = |...| - | 8 | 0 | 0 |
   ```

3. `XMM0 = POR(XMM3,XMM4) = |...| - | 8 | 2 | 5 |`

4. ```
   XMM1 = |...| - | 8 | 2 | 5 |
   XMM2 = |...| - | 2 | 5 | 8 |
   XMM3 = |...| - | 5 | 8 | 2 |
   ```

5. `XMM0 = PMAXUB(XMM1,XMM2,XMM3) = |...| - | 8 | 8 | 8 |`

   `Por lo tanto, la norma infinito del primer pixel es 8.`

2.1.2. Blur gaussiano

Antes de describir cada implementación, se aclarará un par de cuestiones. La primera es el tamaño de la matriz K. Esta matriz tiene una dimensión de $2 \times radio + 1$ píxeles de ancho y alto. Por otra parte, se debe mencionar que no se hace tratamiento de los casos borde en ninguna de estas implementaciones.

Implementación en *C*

Esta implementación, se divide en 2 etapas:

- Se crea la matriz K teniendo en cuenta los parámetros dados, de la manera descripta en el enunciado. Es decir, primero se calcula la función G_σ, lo cual implica simplemente aplicar algunas operaciones aritméticas con los parámetros (sumas, potencias, etc), alguna de ellas provenientes de la librería `<math.h>` de *C*, y, después, con esa función ya calculada, se obtiene cada posición de la matriz K.

- Luego, se usa la matriz *K* para calcular por cada pixel de la imagen de entrada un promedio ponderado, lo cual se registrará en la imagen de salida. Los valores involucrados en este promedio serán de aquellos pixeles que se encuentren en el vecindario del pixel que se esta tratando y éste mismo. Ese vecindario tiene las mismas dimensiones que las matriz de convolución. En este proceso, se recorre la imagen de entrada por fila y luego por pixel. Y por cada pixel, se recorren los elementos de su vecindario por filas y luego por pixel en simultáneo con la matriz *K*. Se ve, entonces, que este proceso se resuelve en 4 ciclos anidados.

Implementación en *ASM*

En esta implementación, se construye la matriz *K* llamando a la función de la implementación anterior que realiza dicho proceso. También, el recorrido realizado sobre la imagen de entrada es similar así como el recorrido sobre el vecindario del pixel sobre el cual se aplica Blur. Aún así, la manera en que se desarrolla el cálculo para determinar el valor que se guardará en la imagen destino es diferente pues se hace uso de las instrucciones *SSE*. Para realizar este cálculo, se realizan los siguiente pasos. Identificado el pixel sobre el cual se quiere aplicar Blur y su correspondiente vecindario, entonces:

1. Para cada pixel de la imagen de entrada que se encuentre en el vecindario, se realiza la transferencia de éste a un registro XMM, supóngase XMM0.
2. Luego, se utiliza la instrucción PSHUFB, con ayuda de una máscara bien definida, con el fin de desempaquetar de *byte* a *double word* los canales de ese pixel sobre el registro donde se encuentra, es decir, el registro XMM0.
3. Después, se hace uso de la instrucción CVTDQ2PS sobre el registro XMM0 para realizar la conversión de entero a número de punto flotante, dejando así, en cada uno de sus *double word*, el valor de cada canal de ese pixel en ese formato.
4. Paso siguiente, se transfiere el valor de la matriz *K*, correspondiente al pixel del vecindario que se este tratando, a un registroXMM, supóngase XMM1. Una vez realizado esta transferencia, se replica este valor en cada *double word* de ese registro utilizando la instrucción PSHUFD.
5. Luego, se utilizan operaciones aritméticas empaquetadas en el entorno de los números de punto flotante para realizar:
   a) el producto entre el registro XMM0, que contiene los canales del pixel en cuestión en formato punto flotante, y el registro XMM1, la que contiene el valor que se tomó de *K* replicado en cada uno de sus *double words*;
   b) la acumulación de este producto en otro registro XMM, supóngase XMM2.
6. Se repiten los pasos anteriores hasta terminar el recorrido por el vecindario. Al finalizar el recorrido, se tendrá en el registro XMM2 los valores de cada canal del pixel que se guardará en la imagen destino en formato punto flotante, en otras palabras, el pixel con blur aplicado en ese formato.
7. Una vez finalizado ese recorrido, se utiliza la instrucción CVTPS2DQ sobre el registro XMM2 para realizar la conversión de número punto flotante a entero, dejando así, en cada uno de sus *double words*, el valor cada canal de ese pixel en ese formato.
8. Luego, se utiliza la instrucción PSHUFB, con ayuda de una máscara bien definida, con el fin de empaquetar de *double word* a *byte* los canales de ese pixel sobre el primer *double word* del registro donde se encuentra, es decir, el registro XMM2. Se garantiza en este paso que los valores empaquetados estarán comprendidos en el rango de valores enteros sin signo que determina un byte, es decir, entre 0 y 255. Esto es así ya que la suma de los productos realizados, al recorrer el vecindario, es un promedio ponderado (gracias a la función G_σ).

9. Finalmente, se realiza la transferencia del primer *double word* del registro XMM2 al pixel de la imagen destino.

En base a este proceso, se desarrollaron dos implementaciones en *assembler*. La primera, a la que se denomina como *ASM1*, procesa un pixel por ciclo. En cambio, la segunda, a la que se denomina como *ASM2*, procesa cuatro píxeles por ciclo. Sobre la implementación *ASM1*, no tiene que aclararse más de lo que ya se ha mencionado en esta sección. Pero sobre la implementación *ASM2*, se deben hacer algunas aclaraciones.

Particularidades de la implementación en *ASM* 2

En esta implementación multiplicó los 4 pixeles por el mismo k_i para luego guardar la sumatoria en su respectivos registros XMM. Los píxeles a levantar por columna estan determinados por la siguiente fórmula: $4m-2r$, donde $4m$ es la cantidad de pixeles por columna y r, el radio. Entonces, cuando r es impar, quedan dos píxeles que debo tratar como caso aparte. El efecto esperado al comparar esta implementación con *ASM1* sería un aumento en la velocidad de ejecución por el simple hecho de procesar más píxeles por ciclo.

3. Experimentación

3.1. Hipótesis y Experimentos

- *Hipótesis 1*: La implementación en *ASM* con *SSE*, es linealmente mas eficiente que su contraparte en *C*. Más precisamente, existe una constante de proporcionalidad k entre 1 y 16 tal que si T_c es el tiempo de procesamiento un filtro en *C* y T_{asm}, el de ese filtro en *ASM*, entonces:

$$T_c(n) = k\ T_{asm}(n)\ \ \forall n = \text{tamaño de la entrada}$$

Proponemos esta hipotesis ya que esperamos logicamente que el tiempo de la version en ASM sea mas rapida, pero no mas de aproximadamente 16 veces mas rapida(pues 16B es la cota maxima de capicidad de procesamiento en paralelo, por ser el tamaño de los registros XMM) y el algoritmo a ejecutar en ambas implementaciones es similar en cuanto a el costo (en tiempo) de ejecucion. Es decir la unica gran ventaja del de ASM es el procesamiento en paralelo(El cual esta limitado entre 1 y 16).
Experimento: Para verificar la hipótesis, tomaremos muestras del tiempo de procesamiento (medido con ticks del reloj). Dadas las funciones $T_c(n)$ y $T_{asm}(n)$ respectivamente, entonces, se graficará en función del tiempo la función

$$k(n) = \frac{T_c(n)}{T_{asm}(n)}$$

y se espera observar que $k(n)$ se comporta como una constante. Más aún, esta constante estará entre 1 y 16 es la k de la hipotesis que queremos probar.

Gráficos e interpretación:

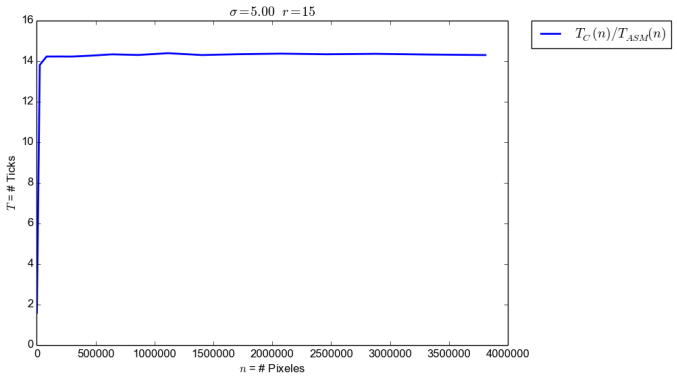


Figura 3.1: Gráfico donde se divide $T_c(n)$ por $T_{asm}(n)$ para el filtro Blur.

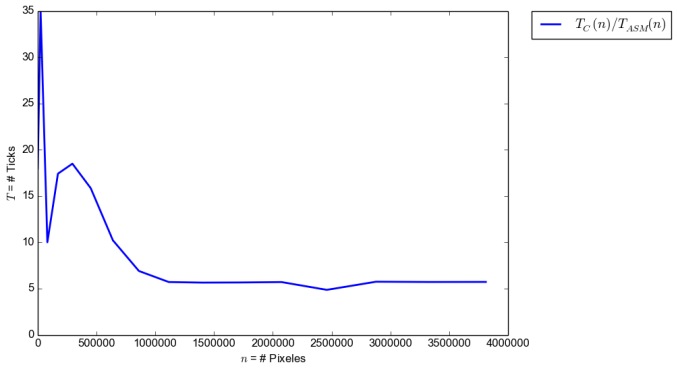


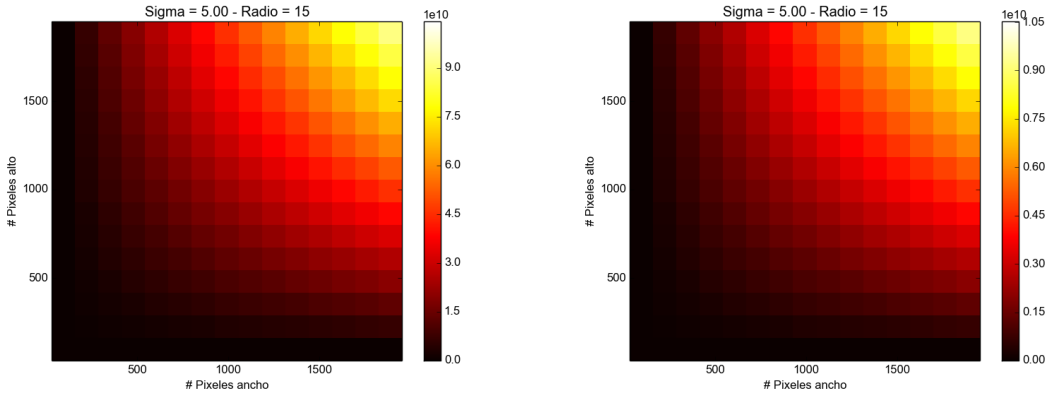
Figura 3.2: Gráfico donde se divide $T_c(n)$ por $T_{asm}(n)$ para el filtro Diferencia.

Se observa en estos gráficos, como esta propuesto en el planteamiento del experimento, que la función $k(n)$ se aproxima a una función constante (exeptuando quizas las imagenes pequeñas que presentan anomalias para imágenes mas chicas). En particular, para el filtro Diferencia el valor hacia donde tiende la constante k es 5 y para el filtro Blur, tiende a 14. Ambas constantes se encuentran, entonces, entre 1 y 16.

- *Hipótesis 2*: Para imágenes suficientemente grandes, el ancho y el alto influyen de manera diferente en el tiempo de procesamiento de los filtros.
  Proponemos esta hipotesis ya que como la imagen se guarda en memoria por filas, tendria sentido esperar que el tiempo de proceso cambie de manera despareja cuando el tamaño de una imagen se da en ancho o en alto.

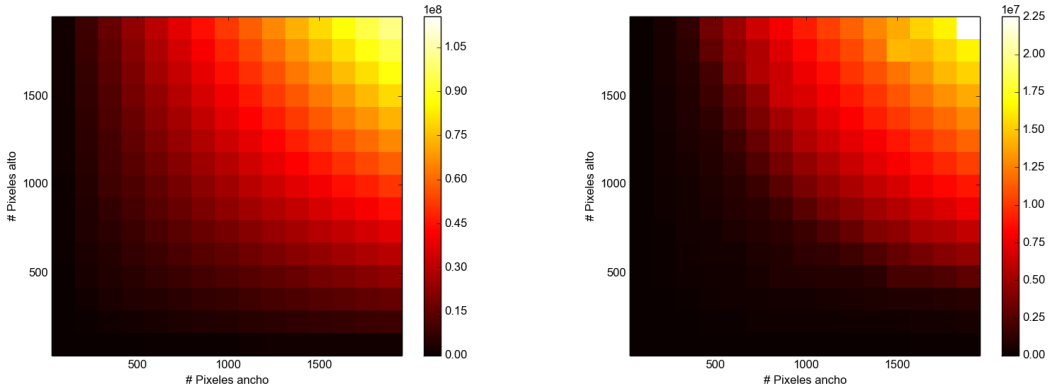
  Experimento: Para verificar esta hipótesis, se tomarán muestras de imágenes de distintos anchos y altos para luego mostrar los resultados en un gráfico de colores donde las dimensiones laterales serán el alto y el ancho y la dimensión del color representará el tiempo tardado para procesar una imagen de tales dimensiones. Una vez realizado este experimento, se analizará la distribución de los colores en el gráfico (mas tiempo de procesamiento). Si esa distribución se realiza de manera simétrica entonces quedará refutada la hipótesis, caso contrario, quedara verificada.

  Gráficos e interpretación:



(a) C (b) ASM

Figura 3.3: Gráfico de tiempo de ejecución por ancho y alto de la imagen para la implementación *C*, figura (a), y *ASM*, figura (b), del filtro Blur.



(a) C (b) ASM

Figura 3.4: Gráfico de tiempo de ejecución por ancho y alto de la imagen para la implementación *C*, figura (a), y *ASM*, figura (b), del filtro Diferencia.

No se observan cambios asimetricos en los tiempos de procesamiento respecto a si el tamaño esta dado por acho o por alto. Es decir, observamos que en todos estos casos las matrices son simetricas.

- *Hipótesis 3*: Para imágenes lo suficientemente grandes (tanto que no entre en cache) al tiempo total de procesamiento se le agregará una constante multiplicada por el tamaño de la fila. Es decir, si a la función que determina el tiempo para imágenes pequeñas se la define como

$$T_p(n)$$

Con n el tamano de la entrada, entonces la función que describe el tiempo en imágenes grandes será

$$T_g(m) = c * \sum_{1}^{k} T_p(n)$$

con c una constante mayor a 1, m tamaño de la imagen grande, y $n = m/k$ de con k lo suficientemente grande para que n se considere de tamano pequeno.

Proponemos este experimento porque esperamos que cuando la imagen sea lo suficientemente grande como para no entrar en cache, se vea afectada la velocidad de proceso por pixel (ya que la tira de pixeles que entran en cache habra que ir a buscarlas a memoria principal y eso significaria un costo en tiempo).

Experimento 3: Para verificar esta hipótesis lo que hará es plasmar las mediciones en un gráfico de tiempo (Ticks de reloj) por pixel, es decir, cuanto tarda cada pixel en ejecutarse. Se debe observar que, a partir de cierto tamaño de imagen, el tiempo tardado por pixel aumenta en una constante.

Gráficos e interpretación:

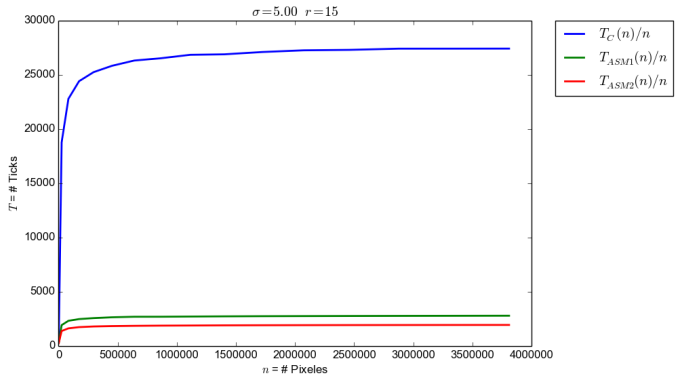


Figura 3.5: Gráfico donde se dividen los tiempos en *C* y en *ASM* por el tamaño de la imagen para el filtro Blur.

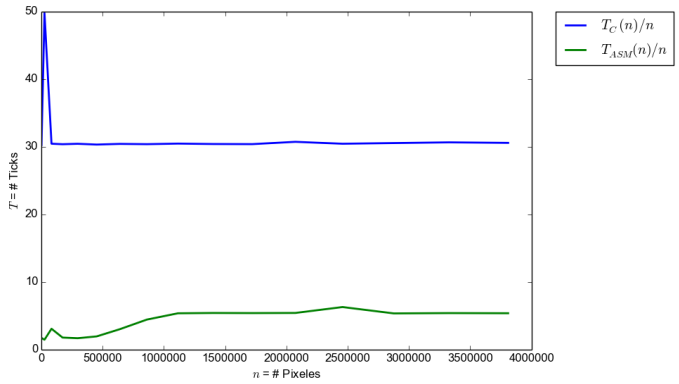


Figura 3.6: Gráfico donde se dividen los tiempos en *C* y en *ASM* por el tamaño de la imagen para el filtro Diferencia.

Contrario al planteamiento del experimento: no se observa ningún cambio significativo en el tiempo de procesamiento por pixel para imágenes grandes, lo observamos viendo que no hay un cambio significativo en la pendiente en niguna de las dos implementaciones.

- *Hipótesis 4*: El tiempo de desarrollo de las implementaciones en *ASM* con *SSE* es mucho mayor que el de desarrollo en *C* (mientras que, si se verifica la primer hipótesis, la velocidad ganada es a lo sumo 16).

  Experimento: Para verificar esta hipótesis, presentamos una aproximación de cuanto tiempo se llevó cabo para implementar cada filtro.

Implementación	Tiempo
Filtro Diferencia en C	40 min
Filtro Diferencia en ASM	5 horas
Filtro Blur en C	2 horas
Filtro Blur en ASM1	7 horas
Filtro Blur en ASM2	12 horas

  Se observa, entonces, que se tarda aproximadamente 5 veces mas el desarrollo de cada función en *ASM* (con *SSE*) que el desarrollo en *C*.

- *Hipótesis 5*: La función que representa el tiempo de procesamiento del filtro diferencia(diff), en función del tamaño de la imagen, se aproxima a una función lineal mientras que la del filtro blur se encontrará entre una lineal y una cuádratica.

  Proponemos esta hipotesis ya que observamos que en el filtro diff, nunca se lee de memoria de memoria mas de una vez cada pixel, a su vez la cantidad de operaciones echa con cada pixel es un numero constante de operaciones(restar, tomar norma y demas), entonces por esto y por el echo de que se leen todos los pixeles esperamos que se comporte de manera lineal.
  Por otro lado en el Blur, procesamos tomando todos los pixeles y para cada pixel tomamos como maximo una vez mas cada pixel de la imagen, eso y el echo que todo el resto de las operaciones son de tiempo constante, nos hace esperar que como mucho tarde un tiempo cuadratico el filtro blur. A su vez se toma cada pixel al menos una vez por lo que el tiempo sera superior o igual a uno lineal.
  Experimento: Para verificar la hipotesis de que el tiempo del diff(tiempo en funcion de cantidad de pixeles de la imagen) aproxima a una funcion lineal, graficaremos el tiempo(en tics) que tarda por pixel en funcion del tamaño de la imagen y observaremos que esta es una funcion constante, es decir independiente del tamaño, osea lineal. Por otro lado, para verificar que el tiempo en función del tamaño de la imagen en el filtro Blur esta acotada inferiormente por una función lineal y superiormente por una cuadrática, si $T(n)$ es el tiempo en función de la cantidad de píxeles, realizaremos los siguientes gráficos:

  - *(a)* $\frac{T(n)}{n}$: si el gráfico de este cociente resulta ser una función constante o si tiende a infinito significará que $T(n)$ es lineal o de algún orden superior al lineal respectivamente.
  - *(b)* $\frac{T(n)}{n^2}$: por el contrario, si en el gráfico de este este otro cociente se observa que la función resultante tiende a 0 significará, entonces, que es de algún orden inferior al cuadrático.

  Si se observan *(a)* y *(b)* entonces se puede concluir que la hipótesis es correcta.

  Gráficos e interpretación:
  -El graficos de $\frac{T_{diff}(n)}{n}$ no lo repetiremos ya que esta mostrado en la experimentacion de la hipotesis 3 (figura 3.6) se observa de este grafico que el cociente del tiempo y la cantidad de pixeles de la imagen es contante para ambas implementaciones, se interpreta entonces que el tiempo de procesamiento del dif es del orden lineal.

  -El grafico $\frac{T_{blur}(n)}{n}$ tambien esta mostrado en la experimentacion de la misma hipotesis (hipotesis 3, figura 3.5). Se observa de este, que el tiempo del blur dividido el tamaño de la imagen(cantidad de pixeles) es al menos constante, es decir mayor o igual a una funcion constante tanto para la implementacion en C como para las de ASM, se interpreta entonces que el tiempo de procesamiento del blur es de al menor orden lineal.

  Por ultimo mostraremos el grafico del tiempo del blur dividido el tamaño de la imagen:

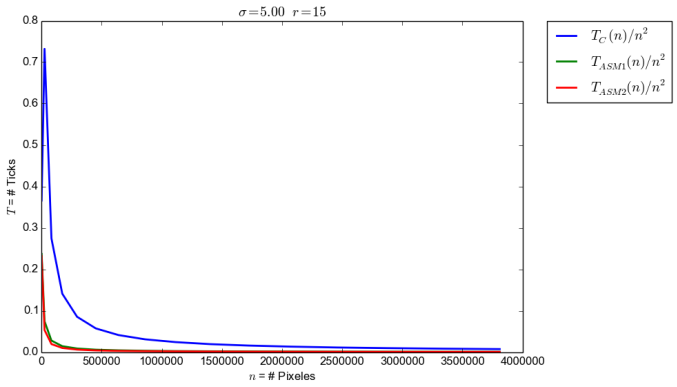


Figura 3.7: Gráfico donde se dividen los tiempos en *C* y en *ASM* por el tamaño de la imagen al cuadrado para el filtro Blur.

En este otro se observa que tiende a 0 el cociente con una función cuadrática, por lo tanto es de orden menor que esta.

4. Conclusión

- *Del experimento 1*: Se concluye que la hipótesis 1 planteada es correcta. El beneficio de la programación en *ASM* con *SSE*, es proporcional al de la programación en *C* si el beneficio lo medimos en tiempo de ejecución (en ticks de reloj), más precesisamente podemos decir que es cerca de 5 veces mejor en la funcion del filtro Diferencia y 10 veces mejor en la funcion del filtro Blur.
- *Del experimento 2*: Al no observarse asimetrías en el gráfico se entiende que el factor que influye al tiempo de procesamiento es la cantidad de pixeles y es indistinto como éstos estan distribuidos en la imagen, incluso en imágenes grandes. Se concluye, así, que la hipotesis planteada se verifica como invalida. Se argumenta respecto a esto que, probablemente, las optimizaciones del uso de la memoria cache de nuestros computadores y, más que nada, la optimización de búsqueda en memoria principal hacen inperceptibles el tiempo de procesamiento y el hecho de que la matriz este guardada por filas.Un posible experimento para corroborar o refutar definitivamente esto podria ser realizar un codigo similar pero cambiando la estructura de la imagen guardandola por columnas.
- *Del experimento 3*: En los gráficos planteados en este experimento, no se nota un incremento considerable en el costo de cómputo cuando las imágenes comienzan a ser grandes. Se concluye, entonces, que, por las muchas optimizaciones de búsqueda en memoria principal, ésta se hace casi tan eficiente como la cache de mas alto nivel y, por lo tanto, se hace imperceptible la diferencia en el costo por pixel entre imágenes pequeñas o grandes. Un posible experimento a futuro para corroborar esto podria ser correr las mismas pruebas deshabilitando el uso de cache y compararlo con los resultados ya obtenidos.
- *Del experimento 4*: Se concluye de lo observado que el tiempo de desarrollo en *assembler* es aproximadamente 5 veces mas grande que el desarrollo en *C*. Además, se observa que el incremento en el tiempo de desarrollo se ve trasladado a la mejora de la performance de manera casi idéntica. Para el caso del desarrollo del filtro Diferencia en *assembler*, se tarda 5 veces más el desarrollo y hace la ejecucion cerca de 5 veces más rápida. Para el caso del filtro Blur, el resultado resulta ser más contundente ya que mejora la perfomance drásticamente. Otro aspecto interesante es que refinar una aplicación en *assembler* (como se realizó en la implmentacion del filtro Blur), tratando de paralelizar lo maximo posible, optimizando el acceso a memoria, usar la mayor cantidad de registros y demás, no produce una mejora significativa en el tiempo de ejecucion pero si aumenta mucho el tiempo de desarrollo.
- *Del experimento 5*: Se concluye que la hipótesis 5 es correcta, efectivamente se observa en ambas implementaciones un comportamiento lineal en el filtro Diferencia y entre lineal y cuadrática en el filtro Blur.

5. Apéndice

5.1. Entorno de desarrollo

Los experimentos fueron realizados en una computadora de las siguientes características:

- Procesador: Intel(R) Core(TM) i5-4570 3.20GHz
- Memoria: 8 GB
- Placa de video: ATI Radeon HD 7970/8970 R9 280X