DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Nº 2

Segundo cuatrimestre de 2015 — Organización del Computador II

Grupo: Smelly Cat

Integrante	LU	Correo electrónico
Frizzo, Franco	013/14	francofrizzo@gmail.com
Martínez, Manuela	160/14	martinez.manuela.22@gmail.com
Rabinowicz, Lucía	105/14	lu.rabinowicz@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, se aplica el modelo de programación vectorial SIMD (*Single Instruction, Multiple Data*) para la implementación de filtros para el procesamiento de imágenes. Más precisamente, se lleva a cabo la implementación de los siguientes dos filtros:

- *Diferencia* (*diff*), que recibe como entrada dos imágenes y devuelve como resultado otra imagen que indica dónde difieren las dos primeras.
- *Blur gaussiano* (*blur*), que suaviza la imagen reemplazando cada píxel por un promedio de los píxeles circundantes, ponderado según una función gaussiana.

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 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.

2. Desarrollo

2.1. Consideraciones generales

Las imagenes que se utilizan como entrada y salida de los algoritmos a implementar son matrices de píxeles. Cada uno de estos píxeles está representado por cuatro enteros sin signo de 8 bits de profundidad (es decir, en el rango $[0, 256)$), que contienen, respectivamente, los valores de los colores azul (b), verde (g) y rojo (r), y la transparencia (a).

Se usará la notación $\mathbf{I}_{x,y}$ para referirse al píxel ubicado en la fila x y la columna y de la imagen $\mathbf{I}$, y la notación $\mathbf{I}^k_{x,y}$ para hacer referencia al valor de la componente k de este píxel, donde $k \in \{\mathsf{b}, \mathsf{g}, \mathsf{r}, \mathsf{a}\}$.

2.2. Diferencia de imágenes

Este filtro recibe dos imágenes como entrada y devuelve como salida una tercera imagen que muestra, en cada píxel, la diferencia entre los píxeles correspondientes de las imágenes de entrada, ignorando la componente a. Más especificamente, si $\mathbf{I_1}$ e $\mathbf{I_2}$ son las imágenes de entrada y $\mathbf{O}$ es la imagen de salida, entonces:

$$\mathbf{O}^k_{x,y} = \begin{cases} \max\limits_{k \in \{\mathsf{b},\mathsf{g},\mathsf{r}\}} \left(\left| \mathbf{I_1}^k_{x,y} - \mathbf{I_2}^k_{x,y} \right| \right) & \text{si } k \in \{\mathsf{b}, \mathsf{g}, \mathsf{r}\} \\ 255 & \text{si } k = \mathsf{a} \end{cases}$$

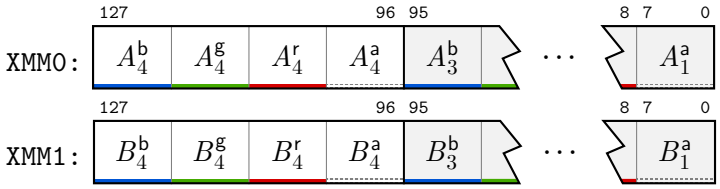
2.2.1. Implementación en lenguaje C

La implementación de este filtro en lenguaje C es sumamente sencilla. Ambas imágenes de entrada se recorren simultáneamente mediante dos ciclos anidados, que iteran sobre sus filas y sus columnas, respectivamente. Para las componentes `b`, `g` y `r` de cada píxel, se calcula el valor absoluto de la diferencia entre los valores correspondientes a ambas imágenes. Luego, se computa el máximo entre estos tres valores, que se guarda como el valor de las componentes `b`, `g` y `r` para el píxel correspondiente en la imagen de salida. Por último, la componente `a` de dicho píxel se define como 255.

2.2.2. Implementación en lenguaje ensamblador

Al implementar el filtro en lenguaje ensamblador, es posible aprovechar las ventajas que brinda el modelo SIMD. En particular, dado que los registros `XMM` son de 16 bytes, se los puede utilizar para procesar 4 píxeles de las imágenes en paralelo, reduciendo la cantidad de iteraciones del algoritmo y, particularmente, de accesos a memoria necesarios para completar el mismo.

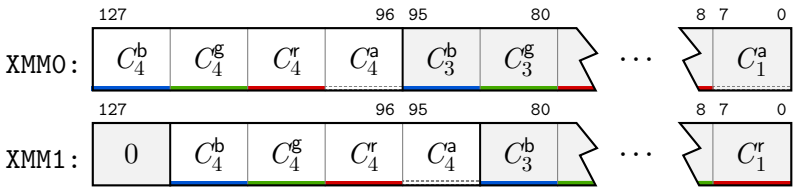
La implementación en este lenguaje del filtro consiste principalmente de un ciclo que itera sobre la imagen. Al comienzo de cada ejecución, se copian 4 píxeles de $\mathbf{I_1}$ al registro `XMM0`, y los correspondientes 4 píxeles de $\mathbf{I_2}$ a `XMM1`.



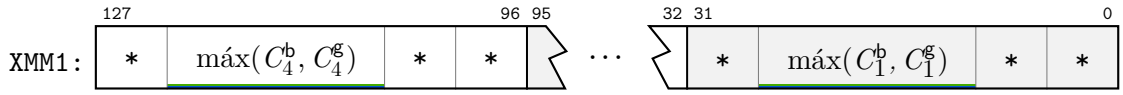
El paso siguiente consiste en calcular, para cada una de las componentes de estos píxeles, el valor absoluto de la diferencia entre ambas imágenes. Para realizar esto, se realiza la resta de las dos maneras posibles, obteniendo `XMM0` $=$ `XMM0` $-$ `XMM1` y `XMM1` $=$ `XMM1` $-$ `XMM0`. En las posiciones donde el valor contenido en `XMM0` sea mayor que el de `XMM1`, será válido el resultado de la primera operación, mientras que en las demás posiciones se deberá tener en cuenta el segundo resultado.

Para seleccionar cuál de los dos resultados es el correcto, se utiliza una máscara que se obtiene comparando los valores de `XMM0` y `XMM1`. Aquí aparece un problema, ya que debemos comparar enteros sin signo, y SSE no brinda instrucciones para hacer esto. Es por eso que se recurre a desempaquetar los números y considerarlos como enteros con signo de dos bytes, que sí se pueden comparar. Empaquetando luego el resultado obtenido, se logra la máscara buscada. Esta se aplica mediante `PAND` al valor contenido en `XMM0` y mediante `PANDN` al valor presente en `XMM1`. Posteriormente se computa un `POR` entre los valores recién calculados, llegando al valor deseado, que se almacena en `XMM0`.

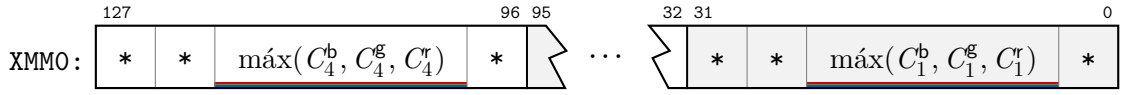
Por último, es necesario calcular la norma infinito de las componentes `b`, `g` y `r` obtenidas. Para hacer esto, se copia en `XMM1` el último resultado y se lo desplaza un byte hacia la izquierda.



A continuación puede usarse la instrucción `PMAXUB XMM1, XMM0` para calcular el máximo entre estos dos registros, obteniendo



Repitiendo el proceso anterior, pero esta vez almacenando el resultado en `XMM0`, se obtiene



A partir de aquí, utilizando la instrucción `PSHUFB` para replicar el valor calculado, se almacena este máximo en las componentes `r`, `g` y `b` de los píxeles correspondientes de la imagen de destino, y utilizando una máscara adecuada, se define la componente `a` de todos ellos como 255.

2.3. Blur gaussiano

Este filtro recibe una imagen como entrada y la devuelve con un efecto de desenfoque; esto se logra mediante la aplicación de una convolución[1] con una función gaussiana, que dependerá de un parámetro σ que podrá ser modificado y regulará la intensidad del efecto. Dada la naturaleza del problema, trabajaremos con una convolución discreta en dos dimensiones, y como nuestro poder de cómputo es limitado, procesaremos solo una vecindad acotada de cada píxel, cuyo radio quedará determinado por un parámetro configurable r. En definitiva, el resultado del filtro será

$$\mathbf{O}^k_{x,y} = \begin{cases} \displaystyle\sum_{i=-r}^{r}\sum_{j=-r}^{r} \mathbf{I}^k_{x+i,y+j}\mathbf{K}_{r-i,r-j} & \text{si } k \in \{\mathsf{b},\mathsf{g},\mathsf{r}\} \\ 255 & \text{si } k = \mathsf{a} \end{cases}$$

donde $\mathbf{K}$ es la matriz o *kernel* de la convolución, con

$$\mathbf{K}_{i,j} = \frac{1}{2\pi\sigma^2} e^{-\frac{(r-i)^2+(r-j)^2}{2\sigma^2}} \qquad \text{para todo } 0 \leq i,j \leq 2r$$

2.3.1. Implementación en lenguaje C

Para aplicar el filtro, es necesario que el parámetro r sea menor que la mitad de la cantidad de filas y menor que la mitad de la cantidad de columnas. Por esto, verificamos que el parámetro cumpla con estas condiciones. El siguiente paso consiste en crear la matriz de convolución del filtro llamando a una función auxiliar, explicada más adelante.

Luego, mediante dos ciclos, se recorre toda la parte de la imagen original a la que es posible aplicarle el filtro. Esta parte es la que contempla las filas desde el valor del radio hasta $filas - (r+1)$, y las columnas desde el valor del radio hasta $columnas - (r+1)$. El resto de la imagen no será afectada.

En cada paso de esta iteración, se llama a la función `afectarPixel`, que se ocupa de modificar cada píxel correctamente, utilizando la matriz de convolución creada anteriormente.

[1] Dadas dos funciones f y g, una *convolución* $f * g$ es una operación que las transforma en una tercera función: $(f*g)(t) = \int_{-\infty}^{+\infty} f(\tau)g(t-\tau)\,d\tau$ en el caso continuo, $(f*g)_n = \sum_{k=-\infty}^{+\infty} f_k g_{n-k}$ $(n,k \in \mathbb{Z})$ en el caso discreto.

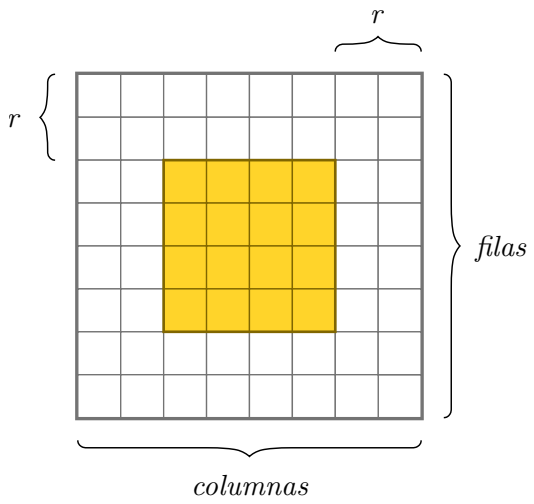


Figura 1: Sector de la imagen afectado por el filtro.

Función `matrizDeConvolucion`

En esta parte del algoritmo se calcula la matriz de la convolución. Para ello, en primer lugar, se piden $4 \times (2r+1)^2$ bytes de memoria, lugar que va a ocupar la matriz ya que su altura es $2r+1$ y su ancho, $4 \times (2r+1)$ (porque cada píxel ocupa 4 bytes).

Se utilizan dos ciclos; ambos van desde 0 hasta $2r+1$.

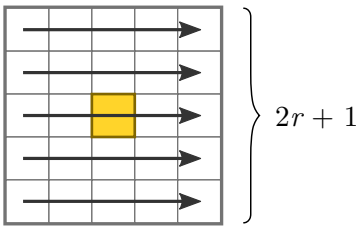


Figura 2: Ejemplo de funcionamiento de la implementación en C de `matrizDeConvolucion` con $r = 2$.

En cada paso, se calcula la función gaussiana con el σ, r, i y j correspondientes, donde i representa la fila y j la columna. Luego, se coloca el resultado en la fila i, columna j de la matriz. Finalmente, se devuelve un puntero a la primera posición de la matriz.

Función `afectarPixel`

Primero se inicializan 3 variables donde luego se van a almacenar las sumas que corresponden a cada componente del píxel a afectar (`b`, `g` y `r`).

Seguidamente, se recorren la matriz de convolución y la parte de la imagen correspondiente a los vecinos del píxel a afectar mediante dos ciclos anidados. Estos van desde 0 hasta $2r$, para recorrer las filas, y desde 0 hasta $(4 \times 2r)$ para recorrer las columnas. En cada paso se multiplica el valor de las componentes del píxel observado por el valor de la matriz de convolución. El resultado de esta multiplicación se suma en las 3 variables creadas en el principio.

Finalmente, se copia el valor de cada variable en cada componente del píxel en la imagen destino. Luego, en la componente `a` se coloca el valor 255.

2.3.2. Implementación en lenguaje ensamblador

Este algoritmo se ocupa de recorrer toda la porción de la imagen a la que es posible aplicarle el filtro.

Al igual que en la implementación en C, primero se hace una comparación para revisar si el radio es válido.

Luego, utilizando la instrucción `CALL`, se hace un llamado a la función `matrizDeConvolución` (implementada en C), la cual devuelve un puntero a la matriz de convolución creada.

Posteriormente, se utilizan dos ciclos para recorrer la porción a modificar de la imagen. Estos utilizan los registros `R8` para recorrer las columnas y `R9` para recorrer las filas.

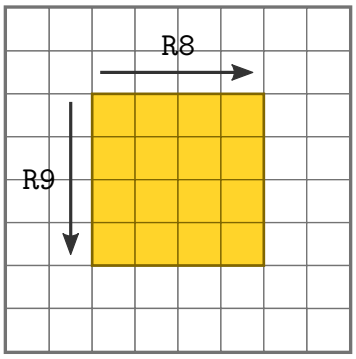


Figura 3: Aplicación del filtro en la implementación en lenguaje ensamblador, con $r = 2$.

En cada paso del ciclo, utilizando nuevamente la instrucción `CALL`, se realiza un llamado a la función `afectarPixel` (implementada en ensamblador) que se ocupa de modificar el píxel correspondiente.

Función `afectarPixel`

El primer paso de este algoritmo es encontrar el puntero al píxel que se debe afectar en la imagen original y otro puntero al mismo píxel pero en la imagen destino. Estos punteros son guardados en los registros `R12` y `R14`, respectivamente.

Llamaremos *submatriz imagen* a la porción de la imagen original que se debe utilizar para que, junto con la matriz de convolución, se obtenga el nuevo valor del píxel. Esta submatriz es la que contiene al píxel a modificar en el centro y su tamaño es el mismo que el de la matriz de convolución.

Luego, se debe encontrar el puntero al primer píxel de la submatriz imagen. Esto se calcula de la siguiente manera: $\texttt{R12} - 4 \times (r - r \times \mathit{columnas})$. Este puntero se encuentra en el registro r13 y es utilizado para recorrer la submatriz.

En el registro `R11` se guarda la cantidad total de píxeles que componen la submatriz imagen.

$$\texttt{R11} = (2 \times r + 1)^2$$

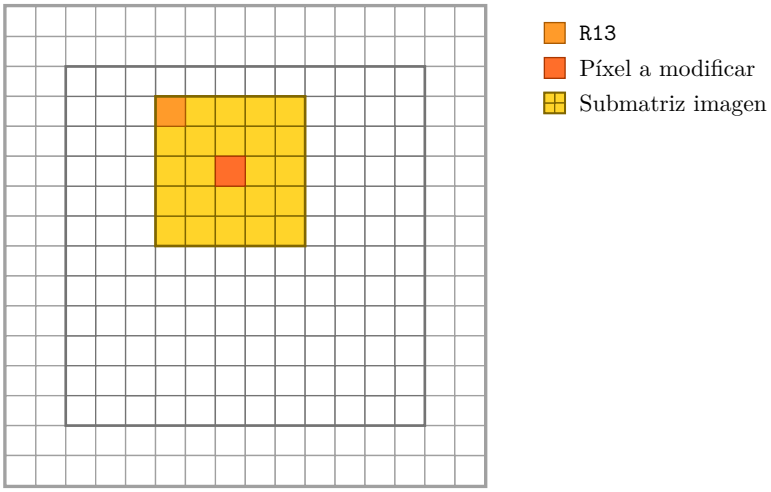


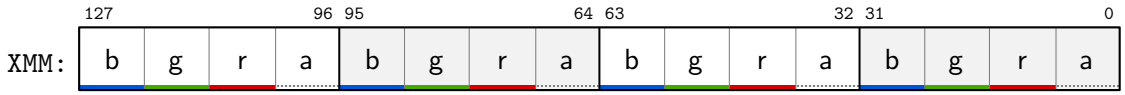
Figura 4: Implementación en ensamblador de `afectarPixel` con $r = 2$.

El registro `R15` contiene el tamaño de las filas.

$$\texttt{R15} = (2 \times r + 1)$$

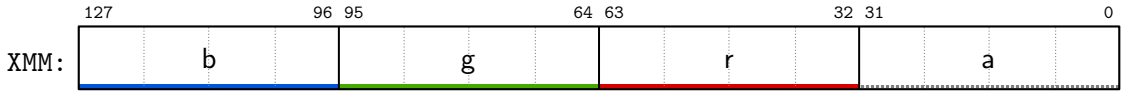
Estos últimos dos se utilizan como registros contadores.

A continuación, se recorren por filas la submatriz imagen y la matriz de convolución a la vez. `R11` se utiliza para saber cuando finalizar el bucle, es decir, cuando ya se observaron todos los píxeles. Estos son procesados utilizando instrucciones SSE y registros del tipo `XMM`. En cada uno de estos registros es posible guardar 4 píxeles, ya que cada píxel ocupa 4 bytes.

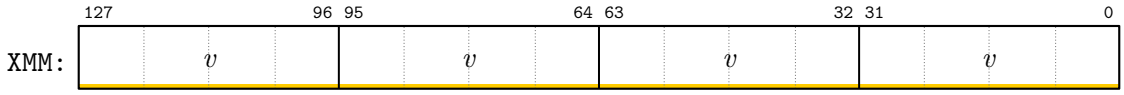


En cada paso de este ciclo se considera una fila, utilizando `R15` para saber cuándo termina. Se toman 4 píxeles de la submatriz imagen y los correspondientes 4 de la matriz de convolución. Luego de desempaquetar los píxeles se realiza la multiplicación de cada componente (`b`, `g` o `r`) con el valor de la matriz de convolución que le corresponde. El resultado de cada producto se suma al registro `XMM6`.

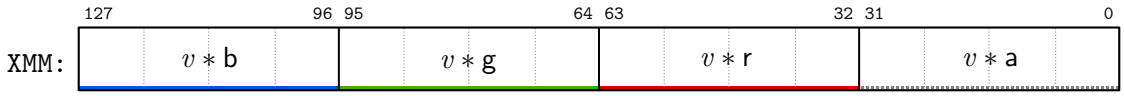
1. Píxel desempaquetado



2. Valor en la matriz de convolución



3. Resultado



Se puede notar que el tamaño de las filas es congruente a 1 o 3 módulo 4. Por lo tanto, se procesan de a 4 píxeles hasta llegar a alguno de los dos casos borde posibles: cuando queda 1 píxel por computar o cuando quedan 3.

La solución a este problema es tomar y desempaquetar los píxeles de la submatriz imagen que todavía no fueron procesados junto con sus siguientes, hasta completar 4. Esto es así en todos los casos, excepto en el que los píxeles restantes son los últimos de la submatriz imagen (es decir, los que se corresponden con la última fila). En este caso, se toman los píxeles anteriores, ya que de otra forma, si el píxel a modificar se encontrase en alguna de las 3 últimas posiciones se accedería a posiciones de memoria fuera de la imagen. Estos píxeles no se tienen en cuenta a la hora de realizar el cálculo.

Esto mismo se realiza para la matriz de convolución.

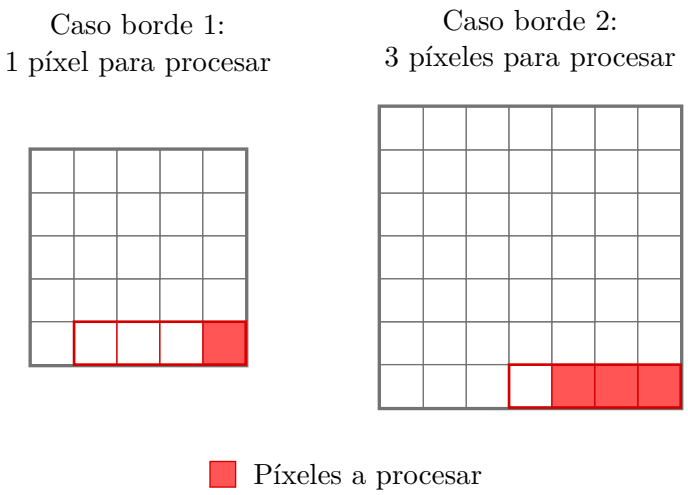


Figura 5: Tratamiento de casos borde en la implementación en ensamblador de `afectarPixel`.

Al finalizar el ciclo, se tiene en `XMM6` el valor esperado para cada componente (`b`, `g` y `r`) del píxel a afectar, en punto flotante de 32 bits. Primero se pasan estos valores a enteros (con la instrucción `CVTPS2DQ`); después, en la componente `a` se coloca un 255, y luego se empaqueta `XMM6` para copiarlo en la posición del píxel que se quiere modificar en la imagen destino.

3. Experimentación

En esta sección, se presentan las pruebas experimentales realizadas sobre ambas implementaciones de los algoritmos, como así también los resultados obtenidos en cada una de ellas y una discusión de los mismos. El objetivo detrás de la realización de estos experimentos fue evaluar y comparar el rendimiento de las diferentes implementaciones, para extraer conclusiones acerca de las características del modelo de programación SIMD.

Todas las pruebas, como así también los gráficos que se incluyen en este informe, pueden ser recreadas mediante la ejecución de una serie de *scripts* incluidos con los archivos fuentes del trabajo práctico. Estos *scripts*, programados en Bash, se encuentran en el directorio `exp`, y llevan por nombre `exp{i}`, donde `i` es el número del experimento. El parámetro opcional `-n <cant>` permite elegir la cantidad de veces que se repetirá cada medición.

3.1. Experimento 1: Comparación de rendimiento

Presentación

En este experimento, se buscó comparar el rendimiento de las implementaciones en lenguaje C de ambos filtros, con el obtenido con las respectivas implementaciones en lenguaje ensamblador utilizando el paradigma SIMD. Las pruebas se realizaron con diferentes tamaños de imágenes, con el objetivo de lograr una comparación independiente de dicho parámetro y, al mismo tiempo, adquirir una idea del comportamiento empírico de cada implementación con respecto al tamaño de la entrada.

Metodología, datos y parámetros del experimento

El experimento consistió en la aplicación de las dos implementaciones de ambos filtros a una serie de imágenes de diferentes tamaños; en todos los casos se midió el tiempo de ejecución, utilizando la instrucción `RDTSC` de la arquitectura *x86-64*. Para mejorar la calidad de los resultados cada medición se repitió un total de 20 veces, calculando luego el tiempo promedio insumido por iteración.

Como dato de entrada, se consideró una imagen de 1800×1200 píxeles (2160000 píxeles en total), que puede encontrarse bajo el nombre `img/phoebe1.bmp`; para utilizar con el filtro *diff*, además, se utilizó una versión modificada de la misma, `img/phoebe2.bmp`. Redimensionando ambas imágenes se crearon versiones de los siguientes tamaños en píxeles: {1801824, 1500000, 1024224, 960000, 726624, 540000, 369024, 240000, 1536000, 60000, 38400, 21600, 9600, 6144, 2400, 1536, 864, 384}.

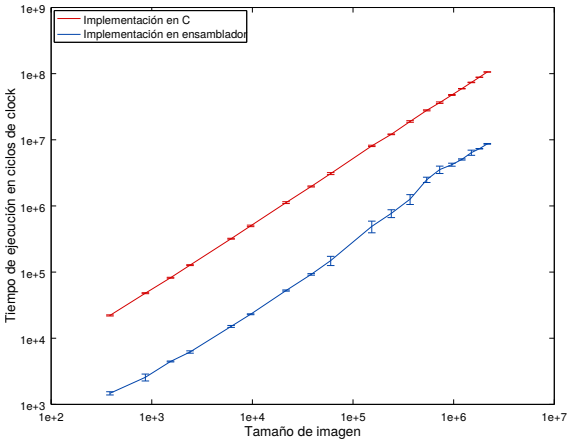
Por otra parte, como parámetros del filtro *blur* se seleccionaron arbitrariamente los valores $\sigma = 5$ y $r = 15$, manteniendo los mismos constantes a lo largo de todo el experimento.

Hipótesis

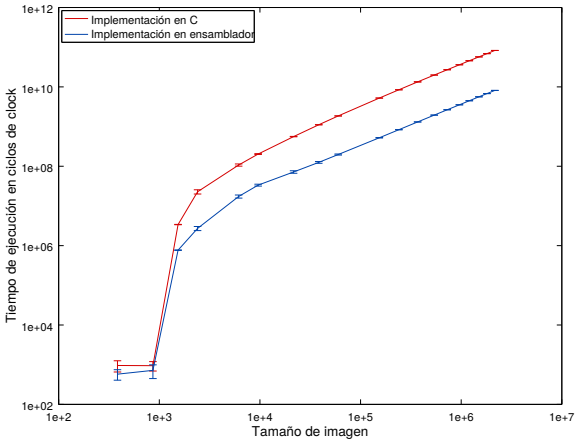
Se espera observar que la implementación en lenguaje ensamblador de ambos filtros sea más eficiente, independientemente del tamaño de la imagen. Esto se debe a que hacen uso del modelo SIMD, con todas las ventajas ya mencionadas que esto tiene sobre el rendimiento del código, a diferencia de las implementaciones de los algoritmos en C, que procesan cada píxel de manera

independiente. Esto último puede inferirse no solo de la estructura propia del código, cuyos ciclos iteran sobre un único píxel a la vez, sino también de la ausencia de instrucciones SEE para procesamiento de valores empaquetados que se observa al desensamblar los objetos compilados a partir de este código.

Resultados obtenidos y discusión

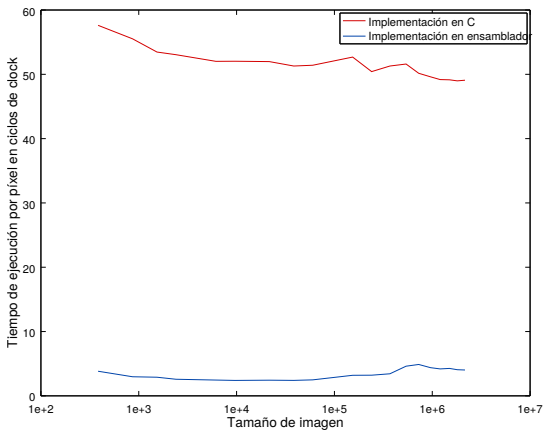


Filtro *diff*

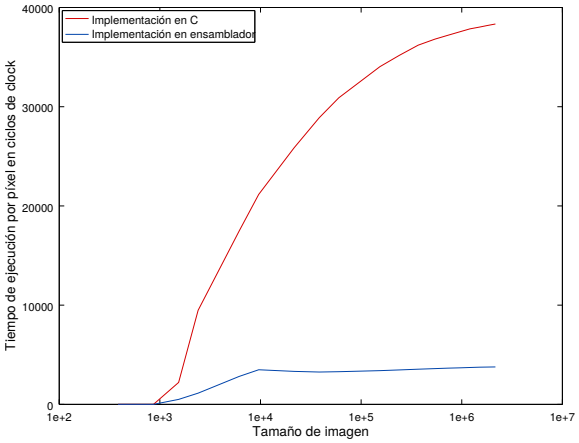


Filtro *blur*

Resultados arrojados por el experimento 1. Los gráficos muestran, para ambas implementaciones, el tiempo de ejecución según el tamaño en píxeles de la imagen.



Filtro *diff*



Filtro *blur*

Resultados arrojados por el experimento 1. Aquí los resultados se muestran normalizados, mostrando el tiempo insumido por el procesamiento de cada píxel.

Como se observa en los resultados, se pudo confirmar la hipótesis planteada: la implementación en lenguaje ensamblador resultó más rápida que la implementación en C para todos los tamaños de imagen.

En *blur*, cuando llamamos a la función con un valor de r mayor a la mitad de la altura o a la mitad del ancho de la imagen, no se producen cambios. Dado que en el experimento el valor de r se mantiene constante, las dos imágenes más pequeñas no se ven afectadas por el filtro, lo

cual se ve reflejado en los resultados, ya que para estas dos imágenes el tiempo de ejecución es notablemente menor.

3.2. Experimento 2: Rendimiento según parámetros del filtro *blur*

Presentación

Este experimento se llevó a cabo con el objetivo de analizar el impacto de los diferentes parámetros del filtro *blur* en el rendimiento temporal de ambas implementaciones del mismo.

Metodología, datos y parámetros del experimento

Para la realización del experimento se consideró una única imagen de 400×600 píxeles, es decir, un total de 240000 píxeles. Luego, se ejecutaron múltiples instancias de ambas implementaciones; esto se hizo en dos etapas, en las que se evaluó por separado el impacto de cada parámetro en el rendimiento general:

(a) Se mantuvo constante el parámetro $\sigma = 5$, y se varió r, tomando los valores {1, 2, 3, 4, 5, 8, 11, 14, 17, 20, 23, 26, 29, 32, 35, 38}.

(b) Se mantuvo constante el parámetro $r = 10$, y se varió σ, tomando los valores {0.5, 1, 1.5, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 50}.

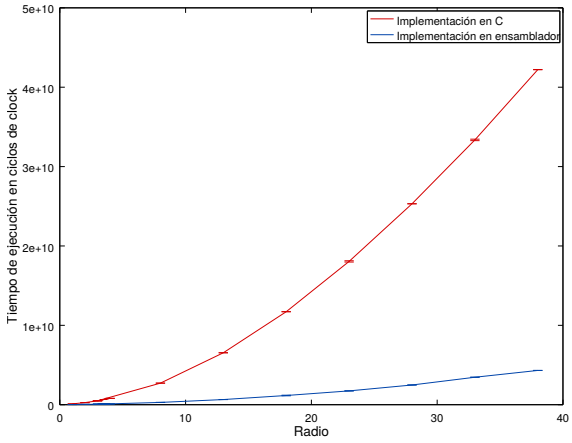
En cada una de las instancias consideradas, se midió el tiempo de ejecución. Esto se hizo, al igual que en el experimento anterior, mediante la instrucción `RDTSC` de la arquitectura *x86-64*, repitiendo 20 veces cada medición y calculando luego el tiempo promedio entre estos resultados.

Hipótesis

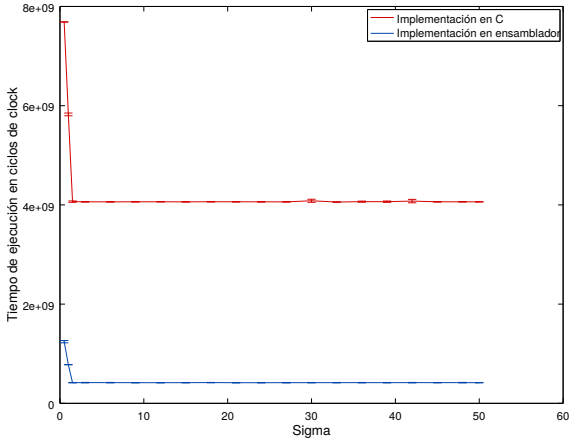
Se conjetura que, a medida que el valor del radio r se incrementa, el tiempo de ejecución en las dos implementaciones aumentará, y que lo hará de manera cuadrática con respecto al incremento en r. Esto se debe a que la complejidad temporal de cada ejecución del ciclo principal del algoritmo depende del tamaño de la matriz de convolución, que es $(2r + 1) \times (2r + 1) \times 4$, es decir, es cuadrático en el valor de r.

Debido a que el valor del sigma es utilizado solamente para realizar un cálculo por cada posición de la matriz de convolución, se estima que modificar este valor no alterará el tiempo de ejecución.

Resultados obtenidos y discusión



Rendimiento según valor de r



Rendimiento según valor de σ

Resultados arrojados por el experimento 2.

Se puede observar en los gráficos que a medida que los r aumenta, también lo hace el tiempo de ejecución. En este sentido, se pudo confirmar la hipótesis. Sin embargo, si se dividen los valores del tiempo de ejecución por su correspondiente r^2, se comprueba que la relación no es lineal; es decir, el tiempo de ejecución no varía cuadráticamente con el valor de r.

Como se había previsto en la hipótesis, la variación del σ no afecta el tiempo de ejecución del algoritmo, tanto en lenguaje ensamblador como en C. Puede observarse que, para valores de σ menores que 1, el tiempo de ejecución es notoriamente mayor. Este es un comportamiento inesperado, que sería interesante estudiar posteriormente.

3.3. Experimento 3: Peso de llamados a función

Presentación

Al realizar este experimento, se buscó descubrir cuál es el peso que tienen en el rendimiento de los diversos algoritmos los llamados a función. Para este fin se utilizó la técnica de *inlining*, que consiste en tomar el código principal y reemplazar los llamados a función por el mismo código de la función. De esta manera, se buscó evaluar la utilidad y el beneficio de estos llamados a funciones auxiliares.

Metodología, datos y parámetros del experimento

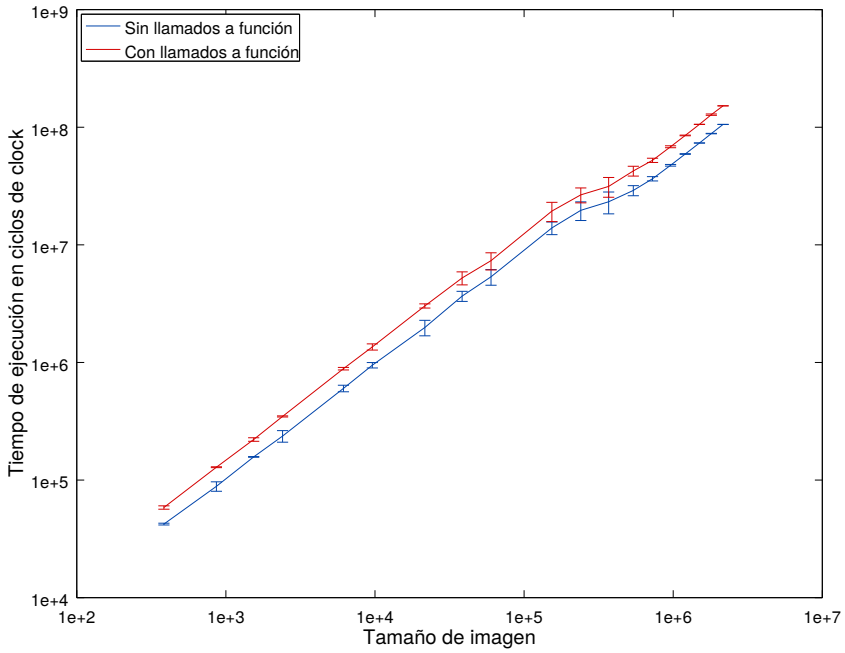
Al igual que en los dos experimentos anteriores, el tiempo de ejecución se tomó utilizando la instrucción `RDTSC` de la arquitectura *x86-64*, repitiendo 20 veces cada medición y calculando luego el tiempo promedio. En particular, se consideraron la versión en lenguaje C de *diff*, a la que se le reemplazaron macros de preprocesador utilizadas para realizar operaciones aritméticas por llamados a función, y la implementación en ensamblador de *blur*, con la que se hizo el proceso opuesto, eliminando los llamados a funciones auxiliares y colocando todo el código directamente en el cuerpo de la función principal. En este experimento el ancho de las imágenes utilizadas como parámetro se encuentran en un rango entre 24 y 1800 píxeles. Además, para el filtro *blur*,

se utilizó $r = 15$ y $\sigma = 5$.

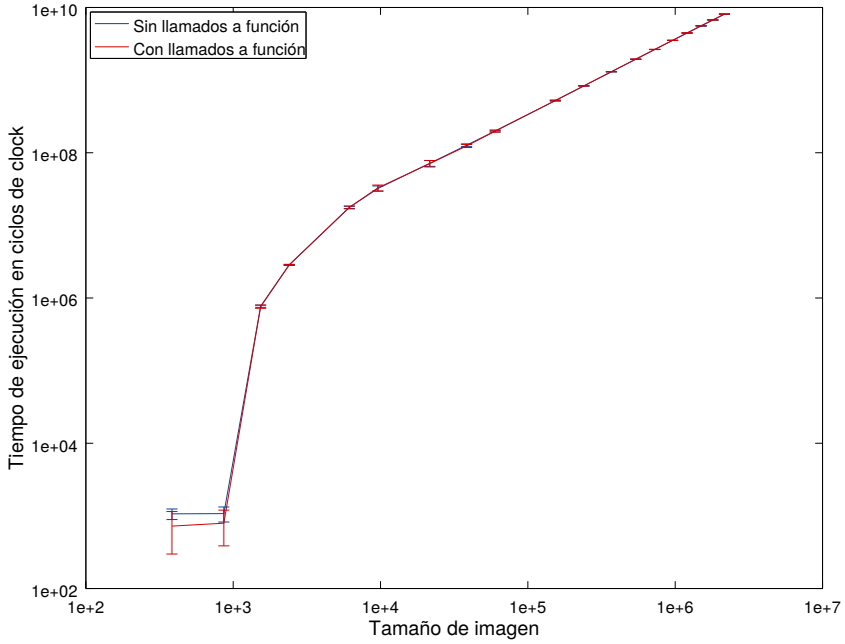
Hipótesis

Creemos que la versión del código implementada que no realiza llamados a funciones va a tener un mejor rendimiento, ya que se evita el overhead que producen estos llamados.

Resultados obtenidos y discusión



Resultados arrojados por el experimento 3. Comparación entre el rendimiento de *diff* en C con y sin llamadas a función.



Resultados arrojados por el experimento 3. Comparación entre el rendimiento de *blur* en ensamblador con y sin llamadas a función.

Como muestran los gráficos de la implementación en C, los algoritmos que hacen llamados a otras funciones tienen un mayor tiempo de ejecución que las que no los hacen. Esto se debe a que cada vez que se hace un llamado a función, en necesario modificar la pila, manteniéndola alineada y guardando los registros que se deben preservar según la convención C y fueron utilizados a lo largo de la función.

En lenguaje ensamblador, cuando implementamos el algoritmo sin el llamado a la función auxiliar, sigue siendo necesario acceder a la pila ya que hay que reutilizar registros que se tienen que mantener para la convención C. Estos registros son R11 y R13. Eliminarlos implicaría que los cálculos necesarios para poder realizar el filtro no se puedan llevar a cabo. Por esto, seguimos haciendo accesos a memoria. Una vez que la accedemos es muy posible que en la caché se encuentren los siguientes accesos a realizar. Entonces, la diferencia entre accederla pocas veces o algunas más es muy pequeña, ya que el acceso a memoria caché no es muy caro.

Posiblemente, si se reformulara por completo la estructura del algoritmo y la manera en que se utilizan los registros, podría lograrse una implementación en lenguaje ensamblador que saque una ventaja más considerable. Sin embargo, esto representaría una labor muy costosa, y hay que tener en cuenta que un código que no realiza llamadas a funciones auxiliares es más difícil de mantener y menos legible, por lo que la ganancia obtenida en rendimiento sería probablemente muy pequeña en relación con las desventajas que se ocasionarían.

4. Conclusiones

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.