DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

16 de Mayo de 2021 — Organización del Computador II

Integrante	LU	Correo electrónico
Bulacio, Blas	284/19	blasbulacio@gmail.com
Ernst, Erik	111/19	erik_ernst@hotmail.com.ar
Salmun, Daniel	108/19	salmundani@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Introducción

Un filtro es un efecto generado en una imagen que resulta de modificar los valores de sus píxeles. En este trabajo, las imágenes a procesar vienen dadas en formato BMP y están almacenadas en memoria como una matríz de píxeles donde cada píxel ocupa 32 bits, 8 bits para cada uno de los elementos **R,G,B,A**, que son **enteros sin signo** que representan los colores y la transparencia (que en las imagenes otorgadas por la cátedra viene fijada en el valor 255 para todos los píxeles).

Implementaremos los filtros **Gamma**, **Max** y **Broken**, cada uno de ellos procesa de forma distinta los píxeles de la imagen de entrada e imprimien el resultado en una imagen de destino (siempre dejando el valor A en 255). Para ello utilizaremos la técnica **SIMD** mediante el set de instrucciones **SSE**, una extensión del x86 provista por primera vez por intel en 1999 para el Pentium III. SIMD es la sigla para Single Instruction, Multiple Data, en español: *una instrucción, múltiples datos*, esta técnica consiste en realizar operaciones sobre múltiples datos a la vez mediante una sola instrucción. El objetivo de SIMD es trabajar con datos en forma simultánea de manera eficiente. Por ejemplo, si dado un vector de tipo *int* queremos calcular el cuadrado de cada elemento, en vez de levantar uno por uno y realizar la operación cada vez, utilizando instrucciones SIMD podemos aplicar el cuadrado sobre múltiples elementos en simultáneo utilizando solo una instrucción.

Para cumplir su objetivo, el set SSE trabaja con los registros **XMM** que tienen un tamaño de 128 bits. Además, los registros XMM son utilizados por el CPU para trabajar con números en **punto flotante**, una forma de representar números reales en la computadora de manera eficiente. En los XMM podemos trabajar con los tres formatos de punto flotante, *single precision* (32 bits), *double precision* (64 bits) y *double extended precision* (128 bits).

Es esta posibilidad de ahorrar una significativa cantidad de instrucciones la motivación para utilizar esta técnica en el desarrollo de los filtros. Dado que las imágenes son matrices de píxeles donde cada píxel ocupa 32 bits, podemos procesar en un registro XMM hasta 4 píxeles a la vez. Explicaremos nuestras implementaciones SIMD y luego mostraremos mediante experimentaciones y análisis el rendimiento de las mismas.

Desarrollo

Filtro Gamma

El filtro Gamma consiste en aplicar sobre los elementos R, G, B de cada píxel de la imagen de entrada la operacion $255 * (\sqrt{\frac{src}{255}})$ e imprimir el resultado en la imagen destino.

Naturalmente, primero armaremos un ciclo que recorra todos los píxeles, y como en un registro XMM entran cuatro de ellos, voy a ir recorriendo la imagen de a 16 bytes, a medida que proceso cuatro píxeles en simultáneo. Las operaciones que se realizan en cada iteración son las siguientes:

1. Levanto con *movdqu* en un XMM los cuatro píxeles actuales, el registro quedará de

la forma XMM = $[p4|p3|p2|p1]$.

2. Ahora a cada componente de cada píxel A[1], R, G, B debo aplicarle la operación $255*(\sqrt{\frac{src}{255}})$, pero recordemos que dichos valores son enteros sin signo, por lo tanto no podemos aplicar esta cuentas directamente, necesitamos convertirlos a punto flotante para operar sobre ellos como números reales. En este caso utilizaremos el formato *single precision* ya que nos permite poner en un XMM un píxel entero, dado que se almacenaría de la forma XMM = $[A|R|G|B]$ donde cada componente del píxel nos ocupa 32 bits. Para tomar el píxel de la parte baja del XMM donde levantamos los de la imagen original, realizamos una extensión de los cuatro *bytes* más bajos a cuatro *doublewords* en otro XMM, utilizando la instrucción *pmovzxbd* (extendiendo con ceros dado que R, G, B, A son enteros sin signo). Luego convertimos las *packed doublewords* en *packed single precision* con la instrucción *cvtdq2ps*.

3. Divido a los elementos del píxel que fue extendido por la constante 255,0 (almacenada previamente en un XMM de forma empaquetada) usando *divps*, luego calculo la raíz de cada elemento empaquetado con *sqrtps* y vuelvo a multiplicar todo por 255,0 con *mulps*. Por último convierto los valores a enteros sin signo con la instrucción *cvtps*2*dq*.

4. Repito este proceso para cada píxel. Para ello voy a ir shifteándolos a la parte baja del XMM con un *psrldq* de 4 bytes e ir repitiendo el proceso.

5. Ahora tendré en cuatro XMMs distintos, el resultado de haber procesado cada píxel por separado, por lo tanto debo juntarlos en un último XMM de forma que queden como vinieron, es decir XMM = $[p4|p3|p2|p1]$. Esto lo puedo lograr mediante las instrucciones ***packssdw***, que junta dos registros de cuatro *packed signed doubleword integers* en uno sólo de ocho *packed signed word integers*, y por último ***packuswb*** que junta dos registros de ocho *packed signed word integers* en uno de dieciséis *packed unsigned byte integers*. En otras palabras, en tres pasos conseguimos juntar los píxeles que teníamos separados en cuatro registros distintos en uno solo. En el siguiente ejemplo, en XMMx se guarda el píxel px, en XMM1 queda la versión procesada de los píxeles originales.

```
packssdw xmm1, xmm2 ; xmm2 = [ p2 | p1 ]
packssdw xmm3, xmm4 ; xmm3 = [ p4 | p3 ]
packuswb xmm1, xmm3 ; xmm1 = [ p4 | p3 | p2 | p1 ]
```

6. Imprimimos en la imagen destino los resultados.

[1]Notemos que A = 255, y $255*(\sqrt{\frac{255}{255}}) = 255$. Por lo tanto podemos aplicar esta cuenta al componente de transparencia sin problema.

Filtro Max

Lo primero que hacemos en el filtro Max es pintar toda la imagen *dst* de blanco. Aunque el enunciado pide solo pintar los bordes nos pareció más importante simplificar el código y esa es la razón por la que pintamos todo en vez de solo los bordes al principio, aunque sea menos óptimo. Por lo tanto, luego de este ciclo tenemos *dst* todo blanco y listo para arrancar con el ciclo principal que recorre toda la matriz, va creando el kernel con los valores de *src* y luego lo aplica para pintar con los valores correspondientes el pixel en *dst*. Expliquemos más en detalle lo anterior: la parte principal del código consiste en un ciclo que itera por las filas y otro adentro del anterior que itera por las columnas. Ambos arrancan en 0 y van avanzando de a dos en dos, como especifica el enunciado. Obviamente, los ciclos terminan correspondientemente cuando alcanza o supera el ancho o el largo de la imagen, sin contar los bordes.

En el cuerpo del ciclo interno esta la parte principal de la lógica del filtro. Lo primero que hacemos es crear una matriz de 4×4 píxeles, arrancando con la posición i, j de la imagen, quedando cada fila de la matriz en un registro XMM distinto. Cada fila esta compuesta por cuatro píxeles ya que, como sabemos, cada píxel tiene un longitud de cuatro bytes y un registro XMM tiene una capacidad de 16 bytes. Una vez que tenemos la matriz, vamos a procesar cada fila por separado de la siguiente manera:

1. Movemos los primeros dos píxeles de la fila a un nuevo XMM extendiendo cada byte a word extendiendo con ceros

2. Shifteamos ocho bytes a la derecha la fila para que queden solo los otros dos píxeles

3. Movemos estos otros dos píxeles a otro registro XMM, nuevamente byte a word y extendiendo con ceros

4. Añadimos horizontalmente word a word en otro registro XMM los dos registros que contienen dos píxeles extendidos cada uno. Al haber extendido a word, no se provoca un overflow de byte a byte que es lo que queremos evitar. De esta forma en el registro resultante queda tal que por cada word se suman dos componentes de cada píxel.

Luego, por cada dos filas que aplicamos el proceso de arriba, podemos nuevamente añadir horizontalmente word a word para que quede en cada word del registro resultante la suma de los cuatro componentes del píxel. Como a es siempre 255, basta ver el máximo de la suma de los cuatro componentes para ver el máximo de la suma de b, g, r

Por lo tanto, nos quedan dos registros XMM de ocho words cada uno, donde uno contiene en cada word la suma de los componentes de un píxel de las primeras dos filas mientras que el otro contiene el de las últimas dos filas. Ahora es momento de buscar el máximo word de estos dos registros. Para eso primero aplicamos la instrucción *pmaxuw* entre los dos registros que mencionamos anteriormente para ir buscando el máximo. Después, hay que ir shifteando los bytes correspondientes y seguir aplicando esta instrucción.

Primero shifteamos ocho bytes, luego cuatro y finalmente dos, intercalando con la instrucción de máximo en el medio. En el último *pmaxuw* que aplicamos va a quedar el máximo de entre los 16 píxeles en el primer word del registro XMM.

Tenemos la máxima suma de los componentes b, g, r, a de un píxel alguno de la matriz de 4×4, ¿pero ahora cómo sabemos cuál de todos los píxeles es el de estos componentes? Primero, broadcasteamos este máximo para que quede en los 16 words del XMM. Después, aplicamos *pcmpeqw* entre el max broadcasteado y ambos XMM que contenían la suma acumulada de los píxeles, una de las primeras, y otro de las últimas filas. Finalmente, unificamos ambas comparaciones en otro registro XMM haciendo *packuswb*. Como el pack es de signado a unsigned, primero le calculamos el valor absoluto a ambas comparaciones para que quede 1 en vez de -1 en el pack. De esta forma, nos quedó un XMM como "índice" donde el byte que está marcado como 1 corresponde al píxel máximo. Por lo tanto, tenemos que armar un ciclo nuevo para encontrar los valores de la componente de este píxel, aprovechando este índice. Para eso, en cada iteración me fijo si el byte menos significativo del XMM "índice" es 1. Si no lo es, shifteo 1 byte a la derecha el XMM "índice" y aumento correspondiente el índice de la fila y la columna de nuestra búsqueda. Una vez que lo encuentra, tenemos el índice de la columna y fila de nuestra matriz 4×4 de donde se encuentra nuestro píxel máximo.

Finalmente conseguimos nuestro píxel máximo, ahora solo nos queda moverlo y extenderlo a la parte baja de un XMM e ir guardándolo correspondientemente en *dst* de la manera que especifica el pseudo código, usando *movq* en este caso. Inicialmente lo hacíamos en un registro general esta parte, pero como el enunciado especificaba que todo el manejo de píxeles tenía que ser en registros XMM decidimos modificarlo para que use un registro XMM.

Filtro Broken

En este filtro no se realizan operaciones con los píxeles del *src* sino que se reescriben los píxeles con un cierto offset para cada componente (excepto para la componente alfa que permanece siempre en su valor máximo). Los offsets de cada componente dependen de la fila, por eso, para mayor eficiencia decidimos precalcular los offsets y almacenar esos valores en una tabla. La tabla tiene tamaño 16 bytes * *height* y en cada fila tiene 4 enteros de 32 bits: el primero es el offset para la componente r, el segundo para la componente g, el tercero para la b y el cuarto es un cero para mantener alineación a 16 bytes.

Luego de precalcular los offsets y antes de entrar en los ciclos, broadcasteamos el width en un registro xmm para luego calcular los jr, jg y jb. Posteriormente se entra en el cicloFila donde simplemente se levantan los offsets correspondientes (de memoria alineada) y se pasa a recorrer todos los píxeles de a cuatro en esa fila en el cicloColumna.

En el cicloColumna, primero se broadcastea el j (posición de la columna) en un registro xmm y se realiza una suma vertical con los offsets de la fila actual. Así ya se tienen los jr, jg y jb aunque falta tomar módulo *width*. Como los offsets son entre -16 y 32 basta con sumar *width* en donde haya quedado una posición negativa y restar *width* donde una

posición sea mayor o igual a *width*. Esto se hace generando máscaras con comparaciones de enteros por mayor, comparando cada entero con cero y con *width*. Luego en las posiciones donde se detectan negativos o enteros mayores a *width*, se coloca *width* en esa posición de la máscara y se resta o suma correspondientemente.

Hecho ésto, ya tenemos en un registro xmm las posiciones a leer. Las colocamos en registros de uso general shifteando y moviendo doublewords. Luego, se levantan 4 píxeles desde las posiciones src_matrix[i][jr], src_matrix[i][jg], src_matrix[i][jb]. Ésto es útil y correcto ya que el jX en la posición $j+1$ será el mismo corrido una posición (recordemos que el offset es el mismo para cada fila).

Una vez que ya tenemos en tres registros xmm los píxeles que necesitábamos ahora hay que quedarse con las componentes (bytes) que queremos. Ésto se realiza con una máscara precalculada y una instrucción blend. Finalmente, se escriben los 4 píxeles nuevos en la posición dst_matrix[i][j].

Comparación

En esta sección compararemos nuestras implementaciones de los filtros realizadas en assembler con las implementaciones en C provistas por la cátedra. Analizaremos cuáles versiones de los algoritmos corren más rápido y por qué, ayudándonos con gráficos para cada una de ellas y explicaremos los resultados. Además para las implementaciones en C utilizaremos distintas optimizaciones del compilador *gcc*. Estas optimizaciones son **O0, O1, O2** y **O3**, nos brindan distintas mejoras a cambio de ciertos recortes de performance.

Correremos 1000 veces las versiones de los filtros ASM, C_O0, C_O1, C_O2 y C_O3, e iremos guardando la cantidad de ciclos de clock del procesador que tomó cada una de ellas. Para mostrar estos resultados haremos uso de **boxplots**, ya que el sistema operativo puede generar **ruido** en nuestras mediciones provocando *outliers*, que se pueden ver con claridad en estos gráficos. También mostraremos con barras el promedio de los resultados para una comparación más general.

La imagen de entrada para estos experimentos fue **Labyrinth.bmp** (provista por la cátedra), con 1280 píxeles de ancho y 720 de alto.

Filtro Gamma

Podemos observar en la **Figura 1** una clara ventaja de la implementación en assembler respecto a las de C. Ninguna de las optimizaciones de gcc termina el algoritmo en una cantidad de clicks tan rápida como ASM, además de que esta última presenta una dispersión muy reducida.

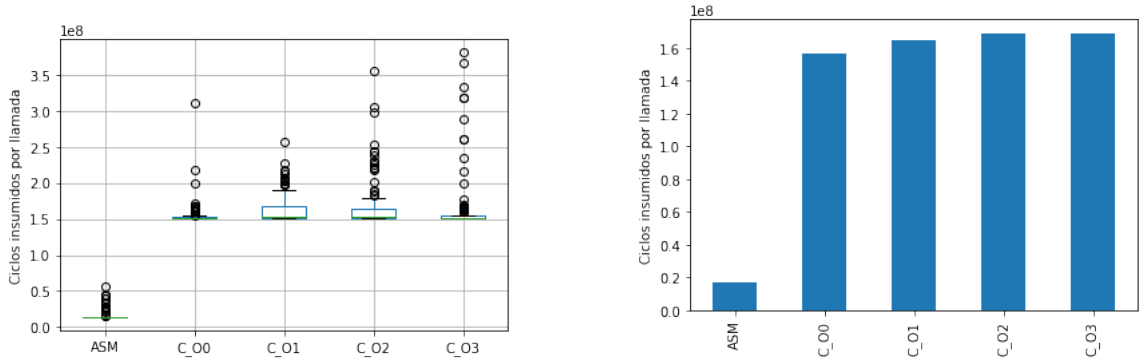

Figura 1
A la izquierda, los boxplots de la cantidad de ciclos que tomó cada ejecución. A la derecha, el promedio de las mismas.

Principalmente esto se debe al uso de SIMD. La implementación en C procesa un píxel por vez teniendo que realizar $1280 * 720 = 921600$ iteraciones, mientras que ASM procesa de a cuatro píxeles a lo ancho, reduciendo esta cantidad a $(1280/4) * 720 = 320 * 720 = 230400$ iteraciones. Además, en el cuerpo del ciclo de C debemos multiplicar, dividir y aplicar raíz tres veces, una por cada valor RGB, mientras que (como vimos en el desarrollo) SIMD permite realizar esto para cada componente de color con una sola instrucción.

Filtro Max

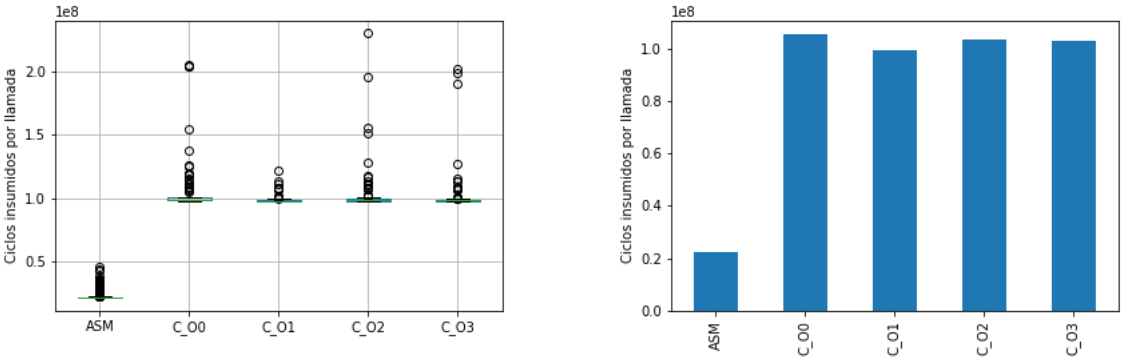

Figura 2
A la izquierda, los boxplots de la cantidad de ciclos que tomó cada ejecución. A la derecha, el promedio de las mismas.

Para el caso del filtro Max, vemos que su implementación en C es aproximadamente cinco veces más lenta que su implementación en ASM. La razón principal por la que creemos que esto es así es que vemos que para armar la matriz de 4×4 en C tiene que

iterar individualmente por cada píxel, accediendo a memoria por cada uno, o sea 16 veces, mientras que en ASM, que tiene un enfoque SIMD, se llama solo cuatro veces a memoria para armar la matriz de 4×4, una por cada fila. Pasa algo similar al escribir en la imagen destino, se itera píxel por píxel de la matriz 2×2 a escribir, accediendo y escribiendo a memoria mientras que en SIMD solo se accede y se escribe dos veces en vez de cuatro. Combinando ambas cosas, vemos que nos ahorramos una cantidad considerable de saltos en el código y accesos a memoria, que debe ser la principal causa de la diferencia de performance entra una implementación y otra.

La implementación de SIMD sería aún más rápida si en vez de pintar toda la imagen destino en blanco al principio, pintáramos solo los bordes como pide el enunciado, pero decidimos optimizar lineas de código en vez de tiempo de ejecución en este caso.

Filtro Broken

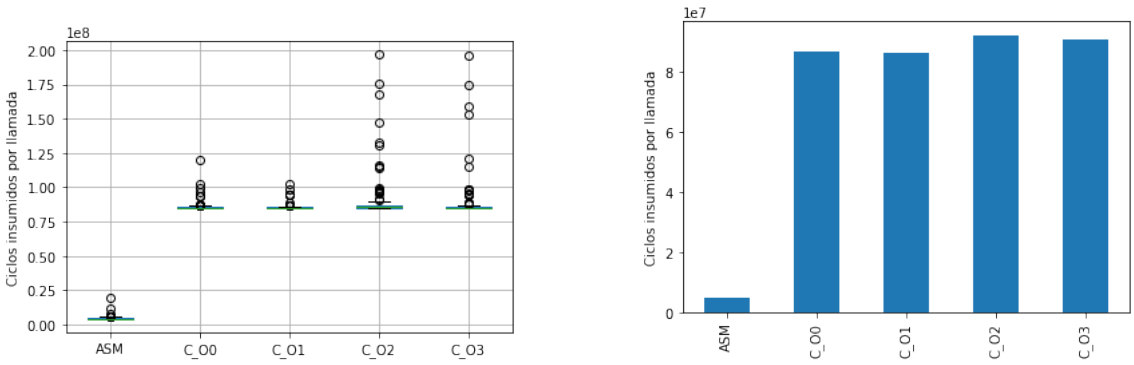

Figura 3
A la izquierda, los boxplots de la cantidad de ciclos que tomó cada ejecución. A la derecha, el promedio de las mismas.

En primer lugar, se puede observar que la cantidad de ciclos insumidos para las diferentes optimizaciones no difieren significativamente con respecto a lo que le insume a la versión en assembler. Sí difieren en sus respectivos desvíos que son muy marcados en los casos de O2 y O3.

En cuanto la versión en assembler obtuvimos resultados muchísimo menores y con un desvío aceptable. Estas grandes diferencias, más allá de las generales entre C y assembler ya mencionadas, se deben a que en ésta implementación se realizan la menor cantidad de operaciones posible. Interpretando algebraicamente el pseudocódigo provisto pudimos obviar operaciones tales como sumarle $8 * width$ a los jX y hacer operaciones equivalentes menos costosas. Un ejemplo de ésto último es cuando hay que calcular módulo, se intenta hacer operaciones solo cuando se requiera y en lo posible reemplazarlo por sumas y restas.

Experimentos y resultados

Los experimentos fueron corridos sobre la virtual machine de la cátedra en una computadora con un procesador Intel Core i3-3220 CPU @ 3.30Ghz.

Experimento 1: impacto de las instrucciones SIMD en la eficiencia

En este experimento exploraremos el impacto del paralelismo sobre el filtro Broken. Para ésto implementamos en assembler de una manera alternativa una parte del filtro mencionado. La parte que se modificó es la primera parte del ciclo, en vez de utilizar un registro xmm para almacenar los offsets se los guarda en registros de uso general. Posteriormente, en el cicloColuma se opera con tres registros de uso general donde a cada offset de la fila se le suma el j. A cada uno de estos registros hay que calcularle el módulo *width*, para ésto se le suma o resta *width* si el jX es menor a cero o, mayor o igual a *width* respectivamente. En lo siguiente el algoritmo no es modificado.

En este caso, cuando hay que sumar o restar se utilizan etiquetas y saltos condicionales, lo cual esperamos que tenga una influencia negativa en el pipeline que llevará a mayores tiempos de ejecución. También, cabe destacar que para no leer de memoria de más se necesitan tres registros de 32 bits en el cicloFila y otros tres registros en el cicloColumna. Con ésto ya nos encontramos con una escasez de registros de uso general (que puede ser resuelto de varias formas).

De esta manera corrimos las dos implementaciones con la misma entrada y obtuvimos los resultados que se observan en el siguiente gráfico:

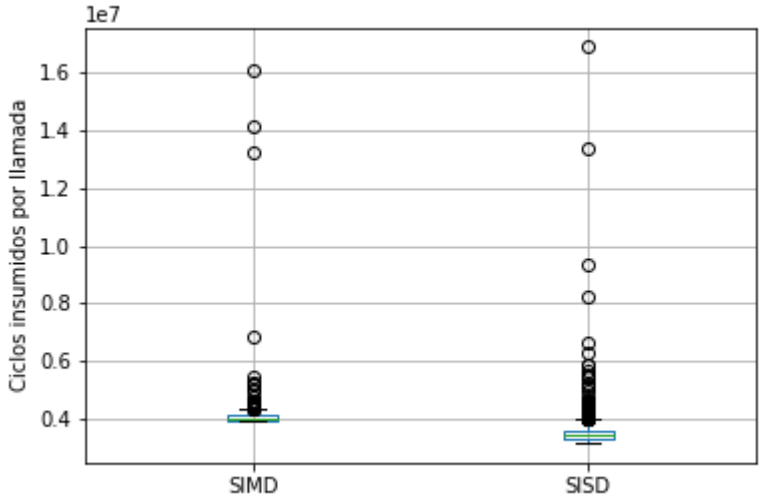

Figura 4
Notamos SIMD a la implementación que cumple enteramente el modelo de ejecución SIMD y SISD a la implementación alternativa recién explicada.

Como podemos notar la implementación alternativa tiene una media sensiblemente menor sin embargo el desvío es mayor y son más frecuentes los resultados alejados de la media. Nuestra interpretación de los resultados es que al ser una sección pequeña del código que solo calcula nuevas posiciones haciendo lecturas, una suma y módulo entendemos que las operaciones SIMD pueden ser demasiado costosa para operaciones tan simples, lo que hace que pese más ese costo que el beneficio, resultando en una media mayor. Más aún, en imágenes no chicas las veces que hay que hacer módulo son pocas debido a que los offsets son entre -16 y 32, solamente habrá que calcular el módulo para tener una posición válida en posiciones muy cercanas a los bordes.

En cuanto al desvío y los outliers, la implementación SIMD, como esperábamos, dio resultados más consistentes y con menor ruido. Es por el pararelismo a nivel de datos que el pipeline favorece estas implementaciones y se muestra en la forma de un menor desvío. Por el contrario, la implementación SISD tiene un ruido muy marcado. Los saltos condicionales limpian el pipeline y el ruido pasa a ser mucho mayor por todos los procesos que la computadora corre en simultáneo.

En conclusión, gracias a este experimento pudimos notar que la utilización de instrucciones SIMD pueden no ser eficientes para casos en los que sean pocas operaciones y que se necesiten en pocos casos (como dijimos, pocas veces necesitaremos calcular el módulo *width*). Por lo que es viable hacer implementaciones mixtas (con el modelo SIMD y SISD) en algunos casos y lograr mejores resultados. Es claro que una implementación puramente SISD sería muy ineficiente y por su parte, una puramente SIMD puede ser ligeramente ineficiente en algunos casos.

Experimento 2: filtro Max con kernel vertical

Como vimos en *Organización del Computador I*, cuando se pide un dato a memoria y no está en la cache, además de cachearse ese dato especifico también se cachean los datos vecinos contiguos horizontalmente. Por lo tanto, nuestro grupo cree que si vamos moviéndonos verticalmente en vez de horizontalmente, va a haber más cache miss y eso va a causar que la función sea considerablemente más lenta. Para probar esto, creamos un nueva función implementada tanto en C como en ASM, llamada Max_v. Este es bastante parecido al filtro Max ya explicado, la principal diferencia es que el ciclo principal ahora itera por las columnas y después por las filas mientras que antes hacía lo contrario. Veamos ahora la diferencia de performance entre el uno y el otro, primero en ASM:

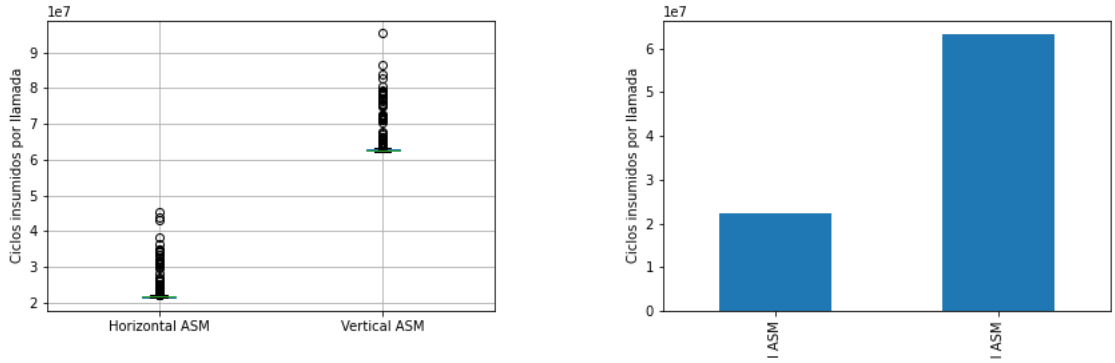

Figura 5: A la izquierda, los boxplots de la cantidad de ciclos que tomó cada ejecución. A la derecha, el promedio de las mismas.

Como podemos ver, al recorrer verticalmente el filtro tardamos aproximadamente tres veces más que si lo hubiéramos recorrido horizontalmente. Como no hay ninguna otra diferencia significativa en el código fuera de lo ya mencionado, eso significa que el acceso a memoria en uno es considerablemente más rápido que en el otro, avalando nuestra teoría inicial de que hay una cantidad significativa de caches miss en uno comparado al otro. Esto es bastante interesante, ya que muestra que si utilizamos de forma errónea dos loops anidados podemos ralentizar nuestro código por un 66 %.

Aún en la implementación de C se ven diferencias considerables de tiempo:

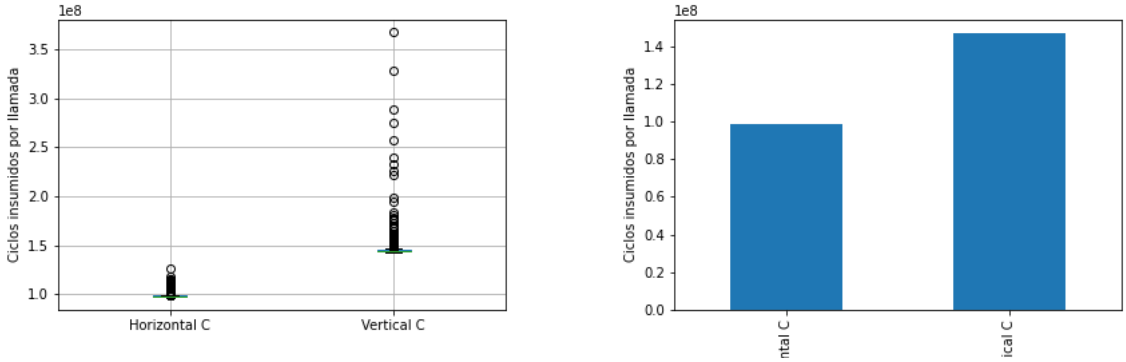

Figura 6: A la izquierda, los boxplots de la cantidad de ciclos que tomó cada ejecución. A la derecha, el promedio de las mismas.

En la implementación de C, el caso vertical es 40 % más lento que el horizontal. No es tan notorio como el 300 % de diferencia que hay en el caso de ASM, pero aún así, esto demuestra que ni las optimizaciones del compilador GCC te salvan del todo si cambias el orden de los loops, va a seguir habiendo cache misses y una baja considerable de la performance.

Para el futuro, sería interesante experimentar que pasa si exploramos nuevamente

horizontal y verticalmente la imagen, solo que ahora en vez de ir del comienzo de la imagen al final, habría que probar que vaya del final de la imagen al principio. Es especialmente interesante para el caso horizontal esto, ya que al pedir a memoria: ¿cachea los que están anteriores en memoria con la misma eficacia que cachea a los posteriores?

Conclusión

En este trabajo, pudimos apreciar la ventaja que supone SIMD a la hora de resolver problemas que tienen como característica aplicar operaciones similares a múltiples datos. En nuestro caso vimos que las imágenes venían almacenadas de forma matricial donde los elementos eran píxeles de 32 bits, por lo que pudimos aprovechar los registros XMM y el set de instrucciones SSE para procesar más de uno a la vez.

Nuestra hipótesis era que una implementación SIMD para los filtros iba a funcionar de forma bastante más rápida que procesar píxel por píxel, por eso realizamos un análisis del desempeño comparando nuestros códigos ASM con los de C provistos por la cátedra. Como fue explicado en la sección de **Comparación** estas hipótesis fueron comprobadas, lo que nos llevó a realizar experimentaciones para aprender más sobre la eficacia de nuestros algoritmos ASM. Una con el filtro **Broken**, donde profundizamos en la comparativa *procesar un dato* vs *procesar múltiples datos* por iteración del algoritmo, y otra con el filtro **Max** donde exploramos los efectos positivos y negativos de la caché en nuestro código.

En conclusión, SIMD es una técnica que para cierto tipo de problemas resulta muy favorable, en este informe respondimos algunas preguntas acerca de cuándo es conveniente aplicarla y de qué forma encarar nuestras implementaciones para aprovecharla al máximo.