Universidad de Buenos Aires

Facultad de Ciencias Exactas y Naturales

Departamento de Computación

Organizacion del Computador II 1C 2021

Trabajo Práctico 2: Filtros

Grupo 5

Integrante	LU	Correo electrónico
Cantini Budden, Sebastian	576/19	sebascantini@gmail.com
Seisdedos, Jose Luis	797/19	jlseisdedos11@gmail.com
Secin, Lucia Maria	748/19	luciamsecin@gmail.com

I. Introducción:

SIMD es un set de instrucciones diseñado para manipular muchos datos con instrucciones individuales (Single Instruction Multiple Data). Se utilizan los registros xmm de 128 bits, permitiendo almacenar dos quad words o, como es en nuestro caso, cuatro double words.

En este trabajo práctico vamos a experimentar con las instrucciones SIMD mediante la implementación de filtros en distintas imágenes y comparándolas con otra implementación en lenguaje C, con el objetivo de comprender como funcionan, evaluando ventajas y desventajas.

II. Desarrollo

A. *Filtro Max*

Este filtro recorre la imagen de a $4x4$ pixeles, formando una submatriz de $2x2$ centrada en la original en cada recorrido. En dicha submatriz se coloca el pixel tal que la sumatoria de sus componentes sea la máxima entre los dieciséis, y la misma se escribe en la imagen destino. Además agrega un marco blanco de un pixel de grosor alrededor de la imagen destino.

La parte principal de la función cuenta con tres ciclos. El primero se encarga de avanzar las filas. Dentro del mismo está el que se encarga de avanzar las columnas, y por último para cada matriz se corre el ciclo $buscarMaximo$ cuatro veces. Una vez que se encuentra el máximo, ponemos en destino la matriz de $2x2$. Finalmente, una vez que se recorrió toda la matriz, ponemos el marco blanco, usando dos ciclos, uno para el borde horizontal y otro para el vertical.

El primer ciclo, correspondiente a la altura ($cicloH$), va avanzando las filas de a dos cada vez que se termina de escribir una fila entera de matrices $2x2$ en la imagen destino y luego comienza el ciclo interior poniendo su contador en cero. El mismo se encarga de avanzar de a dos las columnas ($cicloW$) resetear los registros que van a contener el máximo de la matriz $4x4$ y el pixel al cual corresponde ese maximo, como asi tambien al contador de filas de la matriz en $buscarMaximo$.

El ciclo $buscarMaximo$ itera cuatro veces, en cada iteración encuentra el máximo de cada fila de la matriz y lo compara con el máximo anteriormente guardado, para obtener al final de su última iteración el pixel cuya suma de componentes es la máxima de la matriz. Para iniciar, ponemos en xmm0 los cuatro pixeles de la fila a analizar, habiendo calculado el offset previamente ($r10 * rowsize + r13 * 4$). Se copian estos cuatro pixeles en otros tres registros (xmm1, xmm2, xmm3) en los cuales, usando una instrucción shuffle y una máscara, aislamos en la parte menos significativa de cada double word del xmm los bytes pertenecientes a la parte azul, verde y roja de cada pixel. A continuación se suman los tres registros, y así en cada double word nos queda la suma de los componentes de cada pixel que luego vamos a comparar para encontar la suma maxima.

```
EJEMPLO DEL FUNCIONAMIENTO DE LA BUSQUEDA
DEL MAXIMO DE UNA FILA
suma2>suma1>suma3>suma4

xmm1 = |suma1|suma2|suma3|suma4|
pshufd xmm2, xmm1, 0b00111001
xmm2 = |suma2|suma3|suma4|suma1|
pmaxud xmm1, xmm2
xmm1 = |suma2|suma2|suma3|suma1|
pshufd xmm2, xmm1, 0b00111001
xmm2 = |suma2|suma3|suma1|suma2|
pmaxud xmm1, xmm2
xmm1 = |suma2|suma2|suma1|suma2|
pshufd xmm2, xmm1, 0b00111001
xmm2 = |suma2|suma1|suma2|suma2|
pmaxud xmm1, xmm2
xmm1 = |suma2|suma2|suma2|suma2|
```

Fig. 1. Ejemplo del procedimiento para encontrar el máximo.

Ahora ponemos el máximo cuatro veces en xmm1 y lo comparamos con xmm3 que tiene las sumas de cada pixel para obtener la máscara que indicará cual pixel es el que necesitamos y se la aplicamos tanto a los pixeles como a los máximos. Comparamos el nuevo máximo con el anterior; si el nuevo es mayor entonces queda la máscara con todos sus bits en uno, caso contrario en cero. Esta máscara es aplicada al registro que contiene el pixel, y también al registro que tiene al máximo. Luego es invertida y aplicada al pixel y al máximo obtenidos de la fila anterior. Después, sumamos el máximo viejo con el nuevo y como tienen aplicados la máscara, en xmm6 queda el máximo entre los dos, este registro lo usaremos luego para cuando llegue el momento de escribir en el destino. Hacemos lo mismo para xmm7, en donde quedará el pixel máximo. Finalmente, comparamos el registro que tiene cuatro veces el máximo anterior con el que tiene su máximo actual, y nos quedamos con el máximo de los dos, aumentamos el contador de filas y seguimos el ciclo.

Una vez terminado el análisis de la matriz $4x4$ pasamos a $ponerEnDestino$ que escribe la matriz $2x2$ en la imagen destino. Primero agarramos lo que hay en xmm6 y ponemos $0xff$ en los bytes que tienen $0x00$ y luego calculamos el índice del mínimo horizontal del

registro y shifteamos xmm7 (que tiene el pixel) esa cantidad de double words para poder hacer una única extracción de la posición cero. Extraemos el pixel en r11 y lo ponemos en la matriz $2x2$ de la imagen destino y pasamos a la siguiente matriz $4x4$.

Una vez hecho esto para toda la imagen, agregamos los bordes blancos en la imagen destino con dos ciclos, uno para los bordes horizontales y otro para los verticales, y finalmente termina la función.

B. *Filtro Gamma*

El filtro gamma consiste en aplicar la siguiente ecuación a cada componente del pixel.

$$255 * \sqrt[A]{\frac{Input}{255}} \tag{1}$$

Input es el componente r, g o b del pixel a alterar. El A de la raíz, es el componente gamma del pixel, sin embargo, en el tp utilizaremos una versión simplificada de esta ecuación, que nos quedara de la siguiente forma:

$$255 * \sqrt{\frac{Input}{255}} \tag{2}$$

La implementación la resolvimos alterando cuatro pixeles por iteración del ciclo. Separamos cada componente r, g y b en distintos registros xmm, para poder alterar cada componente por separado usando una máscara para cada color y usando un shuffle para aplicarla (ej: alteramos los cuatro componentes azules, luego los cuatro componentes verdes y finalmente los cuatro componentes rojos).

Una vez que estos fueron separados en sus registros, lo primero que hacemos es convertirlos a packed single precision floating points para poder usar las instrucciones que necesitamos para hacer las cuentas, ya que las mismas operan con floats, luego dividimos por 255 a cada float del registro.

Como cada registro solo tiene un componente del pixel, tenemos que repetir este proceso tres veces, uno por cada componente. Después utilizamos la instrucción para hacer raiz cuadrada a cada componente, y volvemos a multiplicar por 255. Como dijimos antes cada una de estas instrucciones se utilizan tres veces, una para cada componente cromático del pixel. Los pasamos a packed double words de nuevo, y luego con una serie de instrucciones (packs y una máscara), los convertimos a bytes de nuevo, ya que cada color debe ocupar ese espacio. El proceso de conversión de double word a byte está ilustrado para un color en la siguiente imagen (fig.2.), esto lo hacemos tres veces, uno para cada color.

```
EJEMPLO PARA EL COLOR AZUL

xmm1 = | BLUE 1    | BLUE 2    | BLUE 3    | BLUE 4    |

packusdw xmm1, xmm1
xmm1 = | BL1 | BL2 | BL3 | BL4 | BL1 | BL2 | BL3 | BL4 |

packuswb xmm1, xmm1
xmm1 = |B1|B2|B3|B4|B1|B2|B3|B4|B1|B2|B3|B4|B1|B2|B3|B4|

movdqu xmm4, [reordenarB]
xmm1 = |B1|B2|B3|B4| 0000000000| 0000000000| 0000000000|
```

Fig. 2. Ejemplo del procedimiento para pasar de double word a byte.

Finalmente juntamos los 3 componentes, un componente en cada double word y mediante el uso de una máscara los reordenamos para que formen sus respectivos pixeles y les cambiamos la transparencia a $0xFF$. Una vez listos, guardamos los pixeles nuevos en la imagen destino y continuamos con los próximos cuatro.

C. *Filtro Funny*

Este filtro combina la imagen de entrada con una nueva imagen, calculando cada componente independientemente, dividirlo por 2 y luego sumándole el valor original dividido 2. El cálculo de la parte independiente de cada componente es el siguiente:

- **Componente rojo**

$$100 * \sqrt{|j - i|} \tag{3}$$

- **Componente verde**

$$\frac{|i - j| * 10}{\frac{j+i+1}{100}} \tag{4}$$

- **Componente azul**

$$10 * \sqrt{(i * 2 + 100) * (j * 2 + 100)} \tag{5}$$

La implementación de Funny la resolvimos recorriendo cuatro pixeles a la vez y usando dos ciclos, uno maneja las filas ($cicloH$) y otro las columnas ($cicloW$). Dentro del segundo es donde se realizan las operaciones mencionadas arriba.

Comenzamos calculando las partes independientes de cada componente. Para hacer esto, los pasamos de packed double words a packed single precision floating points (PS), para poder usar las instrucciones específicas para estas cuentas y reduciendo así errores de precisión en las operaciones. Para el rojo y el verde no

hubo ninguna otra modificación, en cambio para el azul se necesitó pasarlo a packed double precision floating points y operar con la parte baja y alta por separado, pues encontramos que para algunos casos las operaciones se pasaban de la representación PS.

Una vez terminado esto, nos quedan cuatro registros (uno para el rojo, uno para el verde y dos para el azul), los cuales convertimos de nuevo a packed double words. Luego, truncamos con una máscara; nos quedamos solamente con el byte menos significativo de cada double word, y unimos los dos registros del componente azul. Luego, los dividimos por dos y le hacemos una suma saturada con sus respectivos componentes de la imagen original, los cuales también están divididos por dos. Finalmente, shifteamos los registros para reordenar los componenetes, ponemos la transparencia en $0xFF$, y unimos los componentes para después ubicarlos en destino y continuar el ciclo.

III. Comparación C VS ASM

Decidimos hacer cincuenta mediciones de ciclos de clock para cada implementación, en el caso de la implementación de C también lo realizamos para sus varias optimizaciones, con el propósito de hacer un promedio más exacto y reducir el efecto del ruido del sistema sobre los resultados. Para las mediciones nos aseguramos que el compilador sea el único programa corriendo, ya que un sistema operativo hace procesos por encima de los que la computadora recibe como input y los procesos dictados por el usuario, algo que puede afectar seriamente el resultado de las mediciones y además hacerlas variar, haciendo que no sean del todo confiables. Luego con el uso de Python generamos las figuras que se ven en cada análisis de abajo.

A. *Filtro Max*

De los tres filtros, Max probó ser el más desafiante ya que nos costó mucho encontrar un grupo de instrucciones SIMD que nos permitiera operar con los datos de la manera que queríamos. Sin embargo, luego de incontables intentos llegamos a una versión de Max que no solo satisfacía nuestros requerimientos en cuanto a manejo simultáneo de datos, sino que también logramos que fuera más eficiente que la versión en C.

Tal como podemos ver en el gráfico de promedios de Max (fig.3.), nuestra función en ASM tarda el 26% de los ciclos que tarda la optimización cero en C en aplicar el filtro a una imagen, este es un aumento significativo de la eficiencia en cuanto a tiempo que tarda en correr la función. Comparada una vez mas con las optimizaciones

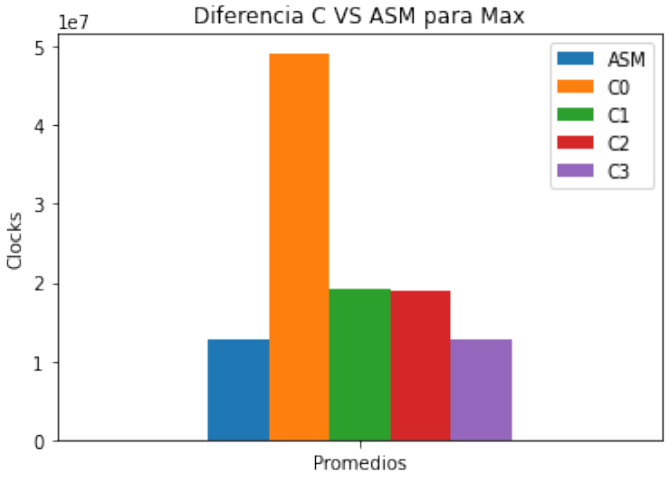

Fig. 3. Comparación entre la implementacion de Max en ASM y distintas optimizaciones de C.

uno y dos, podemos ver que el ASM sigue siendo más rápido, sin embargo, al llegar a la optimización tres, la diferencia en cantidad de ciclos es casi nula.

Esto se puede deber a que la complejidad de la función Max es alta y para poder hacer todas las operaciones se necesitan muchas instrucciones, además, al trabajar con matrices, el ciclo interior para encontrar el máximo y el que pone dicho máximo en el destino deben correr varias veces, haciendo cada vez una comparación y un salto condicional. También, al necesitar el índice de fila y el de columnas, fue necesario agregar dos ciclos externos que los controlen. Esto pudo haber afectado a la rapidez y a la eficiencia de la función.

B. *Filtro Gamma*

Totalmente contrario a Max, Gamma fue el filtro que mas rápido logramos implementar ya que solo fue necesario elegir las instrucciones para cada operación, sin uso de ciclos internos y sin mucho manejo de registros y memoria. Como se ve en el gráfico de promedios de ciclos de Gamma (fig.4.), la función ASM es significativamente más eficiente que todas las distintas implementaciones en C, demorando un 5% de la optimzación cero y un 20% de la optimizacion tres. En el caso de Gamma creemos que esto se debe a que solo usamos un ciclo que además para avanzar lo único que hace es sumarle a los respectivos punteros 16 bytes y aumentar el contador, evitando saltos innecesarios y haciendo el acceso a memoria muy simple pues no se requieren cuentas para saber a que dirección debemos ingresar para buscar o escribir los datos.

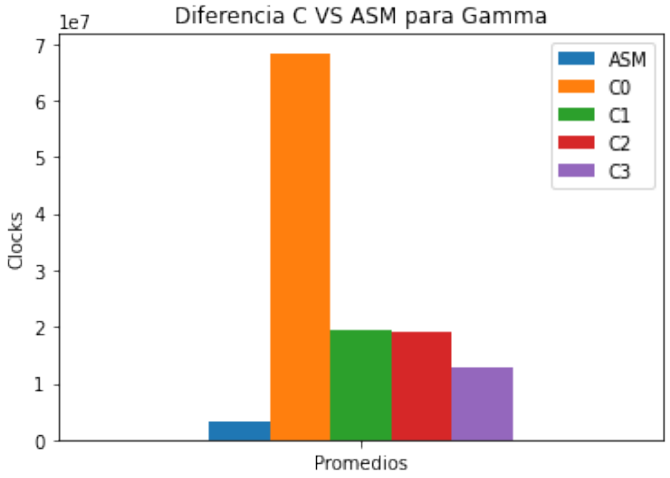

Fig. 4. Comparación entre las implementaciones de Gamma en ASM y distintas optimizaciones de C.

C. *Filtro Funny*

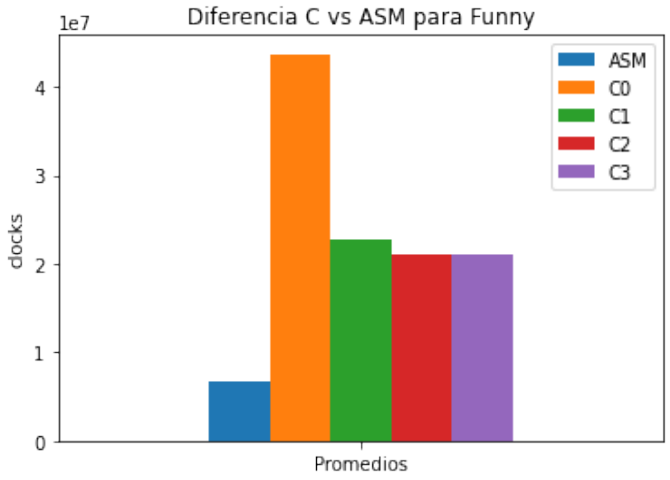

Fig. 5. Comparación entre las implementaciones de Funny en ASM y distintas optimizaciones de C.

Tal como es el caso de Gamma, Funny es simple en términos de manejo de datos y memoria, pues también consistió casi completamente de operaciones para hacer cuentas. Sin embargo, esta vez tuvimos que hacer una conversión extra para el color azul, algo que puede haber afectado la performance de la función al momento de aplicar el filtro. Este además, si bien usa ciclos pues necesitamos los índices de columnas y filas para las cuentas principales, al momento de manejar la memoria lo unico que debe hacerse es sumarle dieciseis a los respectivos punteros.

Más allá de todo ésto, Funny en ASM igual prueba ser más eficiente que sus contrapartes en C, incluso para la mejor optimización. Al compararlo con la implementación en C (fig.5.), descubrimos que demora solo un 15% de la optimización cero y un 32% de la optimizacion tres, comprobando una vez mas la eficiencia de las instruccciones SIMD y su posibilidad de manejar datos simultáneamente.

IV. Experimentación

A. *Experimento 1 : Max*

En este primer experimento vamos a comparar dos implementaciones distintas para el filtro Max. La primera es la versión final que presentamos anteriormente, mientras que la segunda es una versión anterior no optimizada para SIMD, presentada en el archivo adjunto $referencias.asm$. El objetivo de esto es observar las ventajas que tiene una implementación que aprovecha el manejo simultáneo de datos que permiten las instrucciones SIMD sobre una implementación que no lo hace. Esperamos confirmar que la primera implementación es más eficiente, en cuanto a ciclos de clock que necesita para aplicar el filtro sobre la imagen, que la segunda.

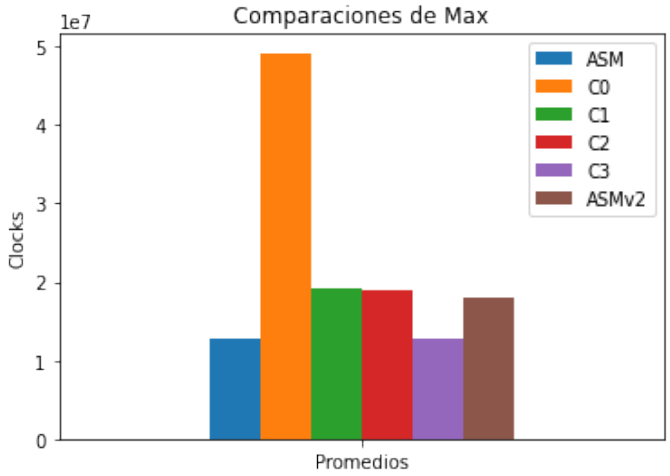

Fig. 6. Comparación entre las implementaciones de Max en ASM, ASMv2 y distintas optimizaciones de C.

Adjuntamos un gráfico (fig.6.) que muestra los promedios de los ciclos de clock para la implementación ASM orginal, la versión ASM contra la cual la vamos a comparar (ASMv2), y para todas las distintas optimizaciones de la implementacion en C. Analizando dicho gráfico podemos ver, que si bien corre en menos de la mitad del tiempo que la optimización cero, el tiempo de ejecución de ASMv2 es muy similar a los de las optimizaciones uno y dos, y no compite con el de la

optimizacion tres, ya que la misma demora un 70% de lo que demora ASMv2.

Esto comprueba que esta segunda versión de ASM, no cumple completamente con el objetivo de usar el set SIMD y los registros SSE para aumentar la eficiencia de la implementación de una función. Sin embargo, esto no quita que igual es notoriamente más eficiente que la implementacion C0, por lo que también confirma que, si bien no es una función del todo óptima, sigue siendo más eficaz que la implementación básica en C.

Finalmente, al momento de compararla con nuestra versión original de ASM, ASMv2 demora un 39% más en aplicar el filtro, esto resalta la importancia de usar de manera inteligente y correcta las instrucciones de SIMD para poder realmente aprovechar los beneficios que tienen para ofrecer. A su vez, también maneja los registros eficientemente para evitar el uso de memoria innecesario y poder procesar la mayor cantidad de datos en simultáneo.

B. *Experimento 2 : Gamma*

Nuestro segundo experimento consiste en comparar nuestra implementación de Gamma contra una en la que no convertirmos ningún valor a floating point antes de realizar las operaciones, presentada en el archivo adjunto $referencias.asm$. Queremos ver cuanto afecta a la precisión del dato final convertir o no los datos a float. Nuestra hipótesis es que va a haber una pérdida de precisión mayor usando la segunda version de gamma y que además tarde menos tiempo en correr dado que tiene que hacer menos instrucciones por ciclo.

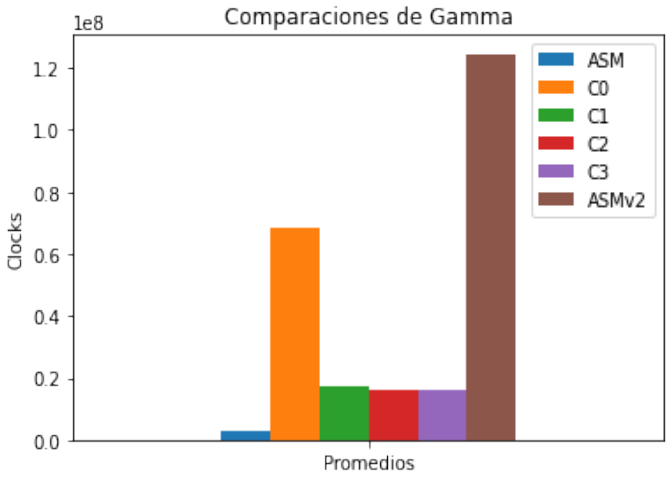

Fig. 7. Comparación entre las implementaciones de Gamma en ASM, ASMv2 y distintas optimizaciones de C.

Lo primero que analizamos luego de aplicar los filtros a la misma imagen fue la diferencia en precisión una vez obtenido el resultado. Si bien ambas pasaron los tests de la cátedra, al usar bmpdiff para compararla con la implementación en C con un error de 0 en vez de 1, nuestra versión pasaba pero ASMv2 no lo hacía para todos los pixeles. Esto confirma nuestra hipótesis de que no manejar a mano las conversiones a float afecta la presición final de la función, pues la mayoría de los pixeles de ASMv2 tenían una diferencia de 1 bit por color con los pixeles de C. Suponemos que este hecho se debe a que al hacer todas las operaciones con enteros, para cada cuenta se tuvo que redondear el resultado y eso terminó afectando a los pixeles finales.

El segundo aspecto que analizamos fue el tiempo de ejecución de cada implementación, lo cual nos tomó bastante por sorpresa y rechazó por completo nuestra hipótesis sobre el tema. Los resultados obtenidos son presentados en el gráfico anterior (fig.7.). Se puede observar a simple vista como ASMv2 tarda significativamente más en correr que todas las otras implementaciones e incluso casi el doble que C0. No estamos seguros de las razones por las cuales ocurre ésto; los resultados nos hacen pensar que se debe a que no convertir los datos previo a las operaciones (como éstas son para datos en punto flotante), el procesador debe convertir por su cuenta los registros a punto flotante, y luego de nuevo a enteros cada vez que tiene que llevar a cabo una instrucción.

Al comparar los tiempos de ASM y ASMv2 podemos ver que ASM tarda solo un 2% de lo que tarda ASMv2, una diferencia notoria. Si bien no tenemos confirmación de porque exactamente pasa ésto, una cosa que queda clara es que si se van a usar instrucciones que son específicas a un tipo de dato, siempre debemos convertir primero y operar después.

V. Desvíos

En esta sección mostraremos los desvíos de las muestras utilizadas en este trabajo práctico para la comparación entre C y ASM y la experimentación. Para cada muestra se puede notar que el desvío es significativamente menor al promedio obtenido previamente. Son notables los desvíos que se observan en Gamma ASMv2 y C0, sin embargo, estos desvíos no constituyen ni el 2% del total del promedio de las muestras respectivas.

Implementación	Desvío
C0	995107
C1	736279
C2	220857
C3	310163
ASM	242206

Fig. 8. Desvíos de Funny.

Implementación	Desvío
C0	1082998
C1	353184
C2	135809
C3	180258
ASMv2	1930980
ASM	104400

Fig. 9. Desvíos de Gamma.

Implementación	Desvío
C0	782770
C1	1167629
C2	144936
C3	173501
ASMv2	341827
ASM	120608

Fig. 10. Desvíos de Max.

VI. Conclusión

El principal aprendizaje que nos llevamos de este trabajo práctico son los beneficios que tienen las instrucciones SIMD.

Al ver todos nuestros resultados, podemos confirmar que todas las versiones bien optimizadas para SIMD en ASM tienen un menor tiempo de ejecución que hasta la mejor optimización en C. Esto se refleja especialmente en la eficiencia del manejo de datos simultáneo para operar sobre grandes conjuntos de datos.

Además, pudimos ver cómo en algunos filtros, manejar la matriz de datos como un gran vector con todos los pixeles almacenados consecutivamente en memoria, puede beneficiar al programa al momento de ingresarla, ya sea para escribir o para obtener datos. Nuestra función Gamma fue la más eficiente de todas y la que más redujo el tiempo de ejecución. Casualmente es la que más trata a la matriz como un vector, solo sumando dieciséis bytes en cada ciclo para avanzar por ella y no haciendo cuentas con los índices de filas y columas para saber a que posición acceder.

Finalmente, comprendimos como aprovechar el potencial de este tipo de instrucciones, ya que un mal uso de las mismas puede afectar significativamente la manera en la que corre nuestro programa. Como vimos en los experimentos, puede ser afectado no sólo el tiempo de ejecución, sino también el resultado final. Esto puede traer varios problemas pues a mayor escala la pérdida de presición puede ser mayor, y al hacer que el procesador necesite convertir más datos o por momentos no procesarlos de manera simultánea, podemos hacer que el programa pase demasiado tiempo en ejecución, y así no obtener una version más eficiente que la que obtendríamos programándolo en C.