DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Informe

Organización del Computador II

Primer Cuatrimestre de 2020

Integrante	LU	Correo electrónico
Patricio Sosa	218/16	patriciososa91@gmail.com
Martín Morán	650/17	martinmoran1994@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

Resumen

En el presente trabajo se describen las ventajas del modelo de ejecución SIMD. Mediante varias implementaciones de código en lenguaje Assembler que utilice dicho modelo, buscamos, mediante experimentación, compararlo con con las mismas implementaciones en un lenguaje de mayor nivel, en nuestro caso C. Demostraremos que las implementaciones en Assembler resultan ampliamente superiores a las C, incluso con su flag de optimización O3 en uso.

Índice

1. Introducción

El objetivo de este Trabajo Práctico es realizar un análisis obre la performance del modelo de procesamiento SIMD. Para el mismo, realizaremos la implementación de tres filtros sobre imágenes en lenguaje ensamblador utilizando dicho modelo. En la sección ”Desarrollo”, detallaremos como se llevaron a cabo las implementaciones, mostrando como fuimos resolviendo las problemáticas que fueron surgiendo, como manipulamos los registros y datos para extraer la información necesaria y consideraciones que tuvimos que tener en cuenta para casos particulares.

En la sección Çomparación”, como su nombre lo indica, compararemos las implementaciones en Assembler con las implementaciones de los filtros en lenguaje C. Mostraremos el impacto que tienen los flags de optimización de código en C (flags O) a la hora de usar implementaciones con código compilado usando estos flags. Tanto en esta sección como en la siguiente, usaremos una métrica de tiempo especial dada por un registro del procesador que lleva cuenta de los ciclos de clock.

En la sección .Experimentos”, realizaremos pruebas sobre las implementaciones, siempre poniendo en contraste el desempeño C con el de Assembler. Las mismas se llevarán a cabo buscando responder algunas de las preguntas brindadas por la cátedra. Se presentará una hipótesis formada por el grupo, realizaremos el experimentos y reflexionaremos sobre los resultados.

Para concluir, haremos un análisis general sobre lo aprendido durante la realización del presente Trabajo Práctico.

2. Desarrollo

En esta sección describiremos las implementaciones de los filtros. En primer instancia, explicaremos las operaciones realizadas previamente a la introducción al ciclo principal, luego un análisis de sus partes y, de ser pertinente, varias consideraciones tomadas en cuenta.

2.1. Ocultar

Inicialización de parámetros Primero calculamos la cantidad total de píxeles de la imagen. La misma se obtiene por medio de la fórmula #filas*#columnas. Como tenemos que realizar la misma operatoria sobre todos los píxeles de la imagen, utilizaremos este número como contador para saber cuántos píxeles quedan por procesar. También lo usaremos para obtener el puntero al src(mirror), en principio cuatro píxeles ante del final de la imagen. Además, inicializaremos un registro cuyo propósito será llevar cuenta del offset para obtener los píxeles actuales a procesar.

Ciclo Como mencionamos anteriormente, verificamos la cantidad de píxeles recorridos para saber cuando dejar de iterar. Procedemos a levantar los cuatro píxeles a procesar de la imagen a ocultar. Realizamos la conversión a escala de grises de los mismos de la siguiente manera:

- Usamos tres registros que representan a los componentes R, G y B.
- Por medio de máscaras, limpiamos las partes que no correspondan a los correspondientes pixeles.
- Shifteamos todo al word más bajo de cada pixel.
- Realizamos las sumas correspondientes trabajando en tamaño de word para no perder precición.
- Finalmente, shifteamos las word dos veces a derecha que corresponde con dividir por cuatro. El resultado de la conversión se encuentra en el byte más bajo de cada doubleword. Como en los bytes más significativos de los words que contienen lo resultados siempre resultan en 0 por la naturaleza de la fórmula, no es necesario realizar operaciones de empaquetado.

En XMM4 queda el resultado de la conversión. A continuación, obtenemos los cuatro píxeles correspondientes a src(mirror). Estos píxeles vienen en el orden inverso al que necesitamos para operar cuando los levantamos de memoria. Para lograr ver la operación de forma más clara mostramos (de forma simplificada) el contenido de los registros en la siguiente ilustración:

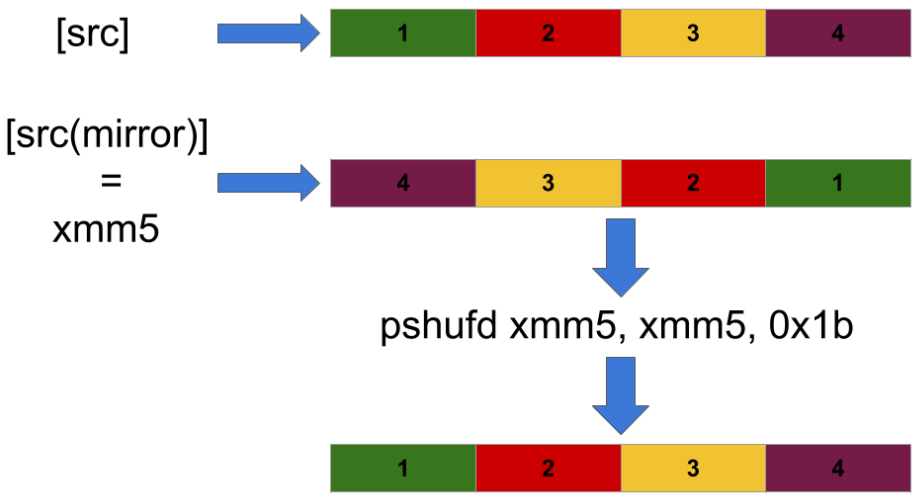

Figura 1: Reordenamiento de datos obtenidos de src(mirror)
Los píxeles en los dos registros son diferentes. Lo que indican los números y colores es que píxel hay que operar con cual.

Seguimos con la obtención de los bits para el procedimiento. Primero, hacemos un shuffle de a byte de modo que quede el resultado de la conversión a escala de grises en cada byte de su píxel correspondiente. Después de esto, dejamos en un registro los bits 3 y 2 de cada componente de src(mirror) con ayuda de una máscara. Los shifteamos dos lugares a derecha para dejarlos en posición a la hora de guardar el resultado del procedimiento en memoria. Luego, para cada componente, obtenemos los respectivos para el procedimiento. La idea es la siguiente:

- Obtener el primer bit empezando por la parte baja del byte con la conversión a gris correspondiente para realizar el XOR (por ejemplo, el primer bit rojo para el XOR sería el que se encuentra en el índice 2 del byte de gris). Lo shifteamos a derecha para que quede en su lugar respectivo de los dos bits más bajos.
- Realizamos el mismo procedimiento para el segundo bit.
- Juntamos los dos registros que contienen estos bits.

El resultado es que tenemos en los dos bits más bajos de cada componente los bits necesarios para realizar el XOR con los componentes de src(mirror). Finalmente, realizamos el PXOR del procedimiento, ajustamos lo valores de la transparencia a 255 y guardamos el resultado en la imagen destino, dejando intactos los 6 bits superiores de cada componente.

2.2. Descubrir

Inicialización de parámetros: Primero calculamos la cantidad total de píxeles de la imagen. La misma se obtiene por medio de la fórmula tot_pixeles=#filas*#columnas. Como tenemos que realizar la misma operatoria sobre todos los píxeles de la imagen, utilizaremos este número como contador para saber cuántos píxeles quedan por procesar. También lo usaremos para obtener el puntero al src(mirror), en principio cuatro píxeles ante del final de la imagen. Además, inicializaremos un registro cuyo propósito será llevar cuenta del offset para obtener los píxeles actuales a procesar.
Ciclo:

Obtención de Datos Ocultos Como mencionamos anteriormente, verificamos la cantidad de píxeles recorridos para saber cuando dejar de iterar. Procedemos a levantar los cuatro píxeles a procesar de la imagen que contienen lo datos para Descubrir la nueva imagen y a continuación los píxeles correspondientes srcMirror y para este ultimo se realiza el mismo procedimiento que en Descubrir, el cual se trata de reubicar los píxeles que obtuvimos con srcMirror tal y como se indica en la Figura 1. Una vez reubicados los datos de srcMirror procedemos a shiftearlos 2 lugares para ubicar los bits numero 3 y 2 en la parte menos significativa de cada componente como se indica en la Figura 2.

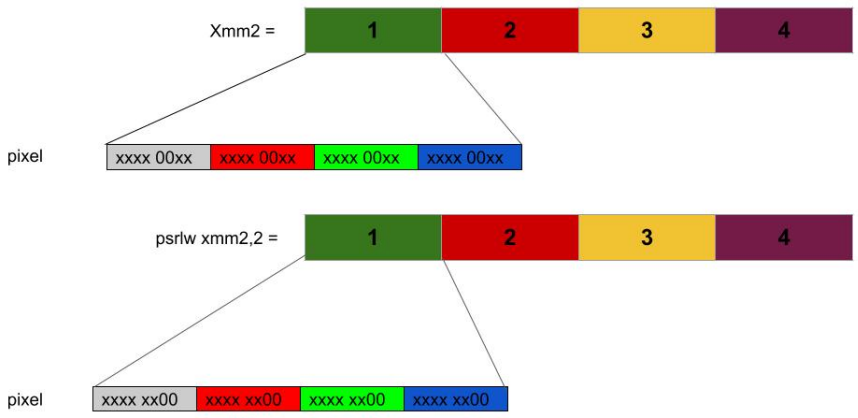

Figura 2: Reordenamiento de bits para obtener la información de la imagen

Luego se realiza una operación lógica xor, para poder finalmente obtener los bits guardados en la imagen en los bits menos significativos de cada componente, una vez hecho esto, utilizamos una mascara para poder extraer únicamente los bits que contienen la información de la imagen oculta.

Procedimiento para recomponer la imagen en escala de grises

Bits del componente Blue:

- Se crea una mascara y se realiza una operación lógica "and" para obtener el bit correspondiente.
- Se reubica el bit utilizando la operación de shifteo.

Una vez realizado el este procedimiento para cada bit de la componente blue, se realiza una operación lógica "or" para poder ubicarlos en un mismo registro

Bits del componente Green:

- Se crea una mascara y se realiza una operación lógica "and" para obtener el bit correspondiente.
- Se reubica la componente green,para ello utilizamos una mascara y la operación shuffle, y la reubicamos en el byte menos significativo de cada píxel.
- Se reubica el bit utilizando la operación de shifteo.

Una vez realizado el este procedimiento para cada bit de la componente green, se realiza una operación lógica "or" para poder ubicarlos en un mismo registro.

Bits del componente Red:

Se realiza el mismo procedimiento que se utilizo en la componente green.

Una vez culminado el proceso para cada componente R,G,B se utiliza la operación "or" para terminar de unir todos los bits obtenidos de cada uno de ellos, todas estas operaciones para extraer cada de bit de cada componente se fueron almacenando en el byte menos significativo de cada píxel. Luego seteamos con ceros, los bits que no fueron ocultados utilizando una mascara y la operación lógica "and".Por ultimo replicamos cada byte, obtenido en estas operatorias sobre los componentes R,G,B y cargamos el valor de la transparencia para poder almacenarlos en dst.

2.3. Zig-Zag

Inicialización de parámetros Para este filtro deberemos realizar operaciones diferentes dependiendo del número de fila, utilizaremos el parámetro de altura (el cual indica la cantidad de filas de la matriz imagen) a modo de contador para saber cuando dejar de iterar. Le restamos 4 a este contador, número que corresponde a las 2 filas superiores e inferiores que hay dejar pintadas de blanco.

Ya que el módulo de la fila al iniciar el procedimiento siempre es 2 y avanzamos de fila en fila de forma unitaria, decidimos inicializar un registro que lleve cuenta del módulo de la filas. El mismo empieza en dos, aumentará en uno al final de cada fila y lo reiniciaremos a cero cada vez que se termine una fila de módulo 3.

Conservaremos en un registro el puntero original a la imagen destino para más adelante realizar el pintado de bordes.

El procedimiento principal se realiza dejando un margen de 2 píxeles por lo que avanzamos los punteros de la imagen fuente y la de destino para que apunten a la segunda fila luego de avanzar 2 píxeles.

En cada iteración recorremos una fila. Para saber cuando dejar de iterar obtenemos un puntero que indique el final de la fila (sin contar los últimos 2 píxeles sobre los que no hay que operar).

Dejamos un registro XMM en cero que utilizaremos más adelante para las operaciones de desempaquetado.

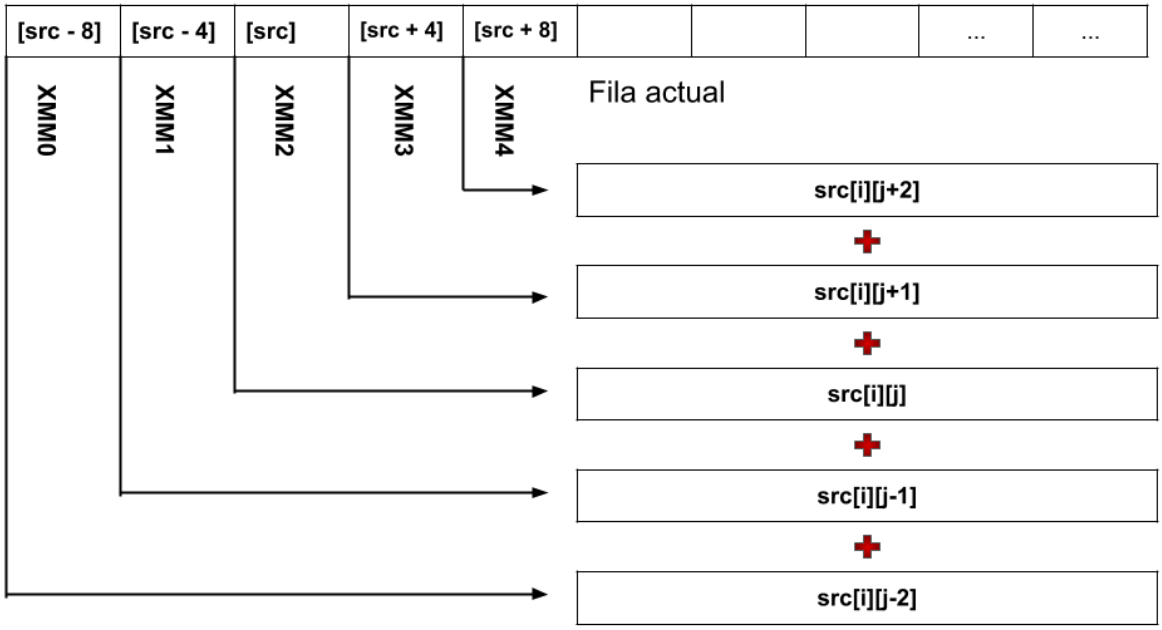

Figura 3: Obtención de los elementos a sumar

Ciclo principal Este ciclo esta dividido en tres secciones, correspondientes a las operaciones a realizar dependiendo del módulo de la fila. Tanto módulo 1 como módulo 3 son simples y autoexplicativos por lo que nos concentraremos en describir el caso módulo 0 y 2.

En la figura 3 presentamos un esquema. El mismo busca mostrar de manera simplificada como buscamos obtener los datos para realizar las operaciones. La idea es considerar a los cinco registros como las cinco partes de la fórmula a aplicarle a los datos. A modo más detallado, los cuatro píxeles levantados de src quedan alineados con los cuatro píxeles de cada uno de los otros registros, lo que nos permite realizar operaciones verticales.

Para evitar perder precisión a la hora de sumar, debemos dividir el procedimiento en dos partes: primero realizaremos las cuentas sobre los dos píxeles bajos, luego sobre los dos alto y, finalmente, uniremos los resultados en un único registro para guardarlo en memoria. En la figura 4 se muestra, de forma esquemática, como buscamos operar durante una iteración del ciclo. En resumen:

1. Tomamos los dos píxeles bajos levantados de src y sus dos pares aledaños y desempaquetamos sus bytes a words.
2. Realizamos la suma verticalmente.
3. Como no existe una instrucción que permita la división de números enteros empaquetados, decidimos utilizar la instrucción DIVPS. Para utilizar la misma tenemos que convertir las words actuales que tenemos a punto flotante. Una vez más, vamos a tener que partir el proceso en dos partes:
   - Pasar tanto los words bajos como los altos de la sumatoria a doublewords por medio de instrucciones de desempaquetado.
   - Convertirlos a punto flotante.
   - Dividirlos por cinco.
   - Convertirlos devuelta a doublewords (aquí para la conversión realizada utilizamos la opción de truncado. Esto se debe a que la comparación la hacemos con imágenes filtradas en lenguaje C. En este lenguaje la división entera se resuelve por medio del truncado de datos, tendiendo siempre el resultado hacia la parte entera más cercana a 0).

- Empaquetamos los resultados de doublewords a words, poniendo la parte alta y baja donde corresponden.

4. Repetimos los pasos 1, 2 y 3 para los dos píxeles altos de src.
5. Finalmente, empaquetamos los dos píxeles bajos y los dos altos ya procesados de vuelta a byte.
6. Procedemos a guardar en memoria.

Pintado de bordes Una vez que terminamos de procesar la parte media de la imagen, procedemos con el pintado de los márgenes. Como siempre, realizamos la explicación de a pasos:

1. Colocamos el valor 0xFF correspondiente con el color blanco en todos los bytes de las dos primeras filas de la imagen. Iteramos hasta el final de la segunda fila/principio de la tercera.
2. A partir del principio de la tercer fila y hasta el final de la antepenúltima, levantamos la primer double quadword de la fila. Con ayuda de máscaras modificamos los datos para que queden Fs en la parte baja del registro utilizado, mientras que la parte alta quede con los datos originales.
3. Avanzamos el puntero al final de la fila y realizamos el mismo proceso, esta vez modificando los datos de forma inversa a la descripta.
4. Para las dos últimas filas, realizamos el mismo procedimiento que en el paso 1 hasta llegar al final de la imagen.

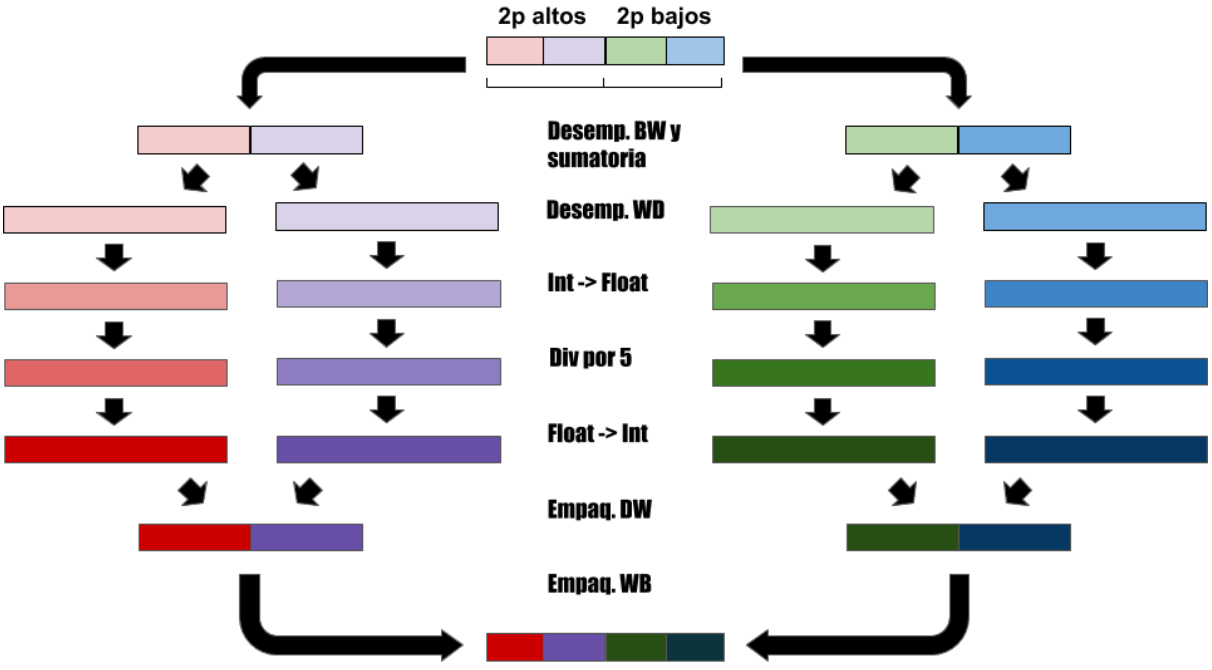

Figura 4: Una iteración del ciclo módulo 0 o 2

3. Comparación

En esta sección se realiza una comparación de cada una de de las implementaciones de los filtros, tanto en su versión en el lenguaje C, como en su versión en Assembler. La comparación se realiza sobre la cantidad de ciclos de clock que le lleva a cada implementación, para ello utilizamos la instrucción del procesador rdtsc permite obtener el valor del Time Stamp Counter (TSC) del procesador. Este registro se incrementa en uno con cada ciclo del procesador. Obteniendo la diferencia entre los contadores antes y después de la llamada a la función, podemos obtener la cantidad de ciclos de esa ejecución.
Para realizar la comparación se varia el parámetro de entrada en 3 medidas: Tomando una imágenes de 256x128(32.768 píxeles), 512x256(131.072 píxeles) y otra de 800x450(360.000 píxeles)

Imagen de 256x128: Para este caso se tomaron 200 muestras de cada implementación y se realizaron boxplots de una de ellas que se presentan en la Figuras 5, 6 y 7 , en las cuales se puede apreciar la distribución de las mismas y los datos atípicos, al igual que en el caso anterior donde el parámetro corresponde a una imagen mas grande, se puede apreciar la gran diferencia entre las implementaciones C vs ASM donde ASM tiene una media por debajo de los 250.000 clocks de reloj para todos los filtros testeados mientras que C en su mejor optimización tiene un piso de 350.000 clocks de reloj en adelante para todos los filtros muestreados.

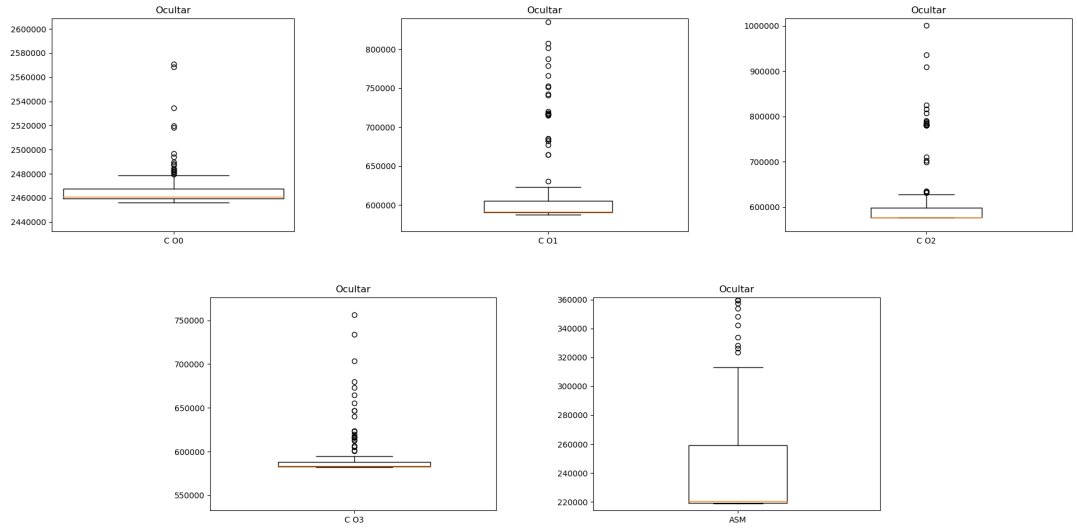

Figura 5: Boxplots de las muestras obtenidas para cada implementación de Ocultar

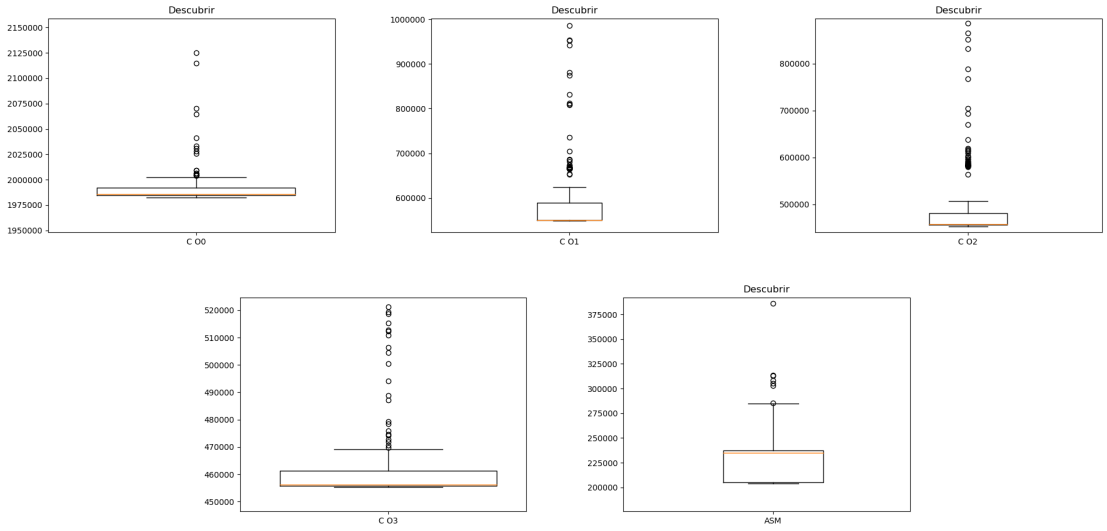

Figura 6: Boxplots de las muestras obtenidas para cada implementación de Descubrir

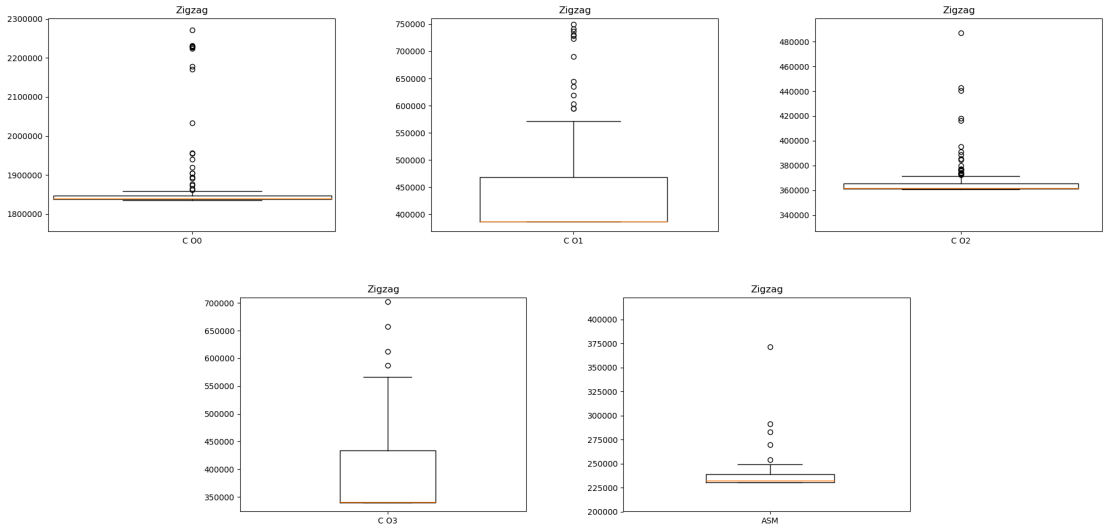

Figura 7: Boxplots de las muestras obtenidas para cada implementación de Zigzag

Imagen de 512x256: Para este caso se tomaron 200 muestras de cada implementación y se realizaron boxplots de una de ellas que se presentan en la Figuras 8, 9 y 10 , al igual que en el tamaño anterior se observa la misma tendencia en cuanto a la comparación C vs ASM, donde el lenguaje C en su mejor implementación supera ampliamente el millón de clocks para su mejor optimización en C en todos los filtros mientras que el lenguaje ensamblador permanece por debajo de dicho valor.

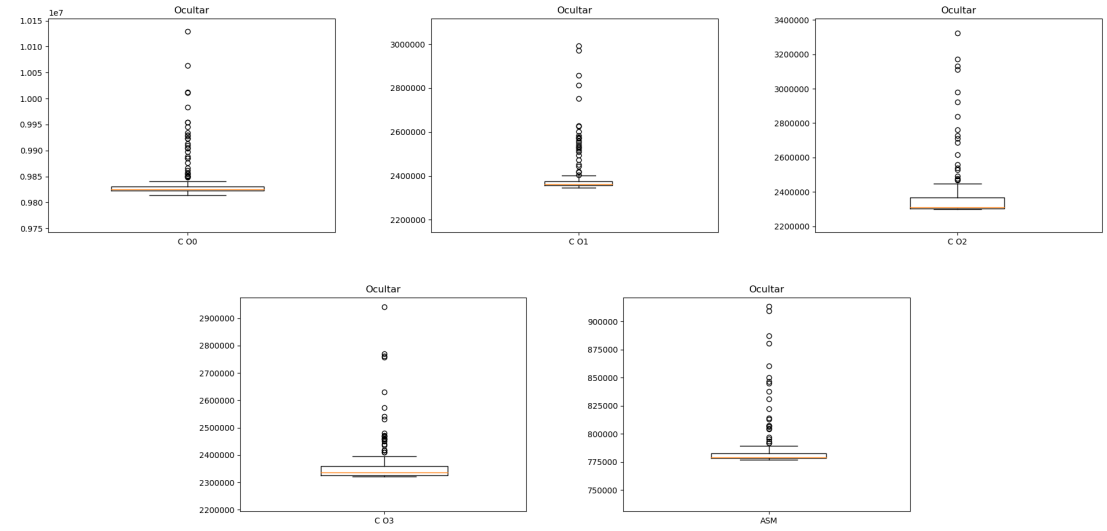

Figura 8: Boxplots de las muestras obtenidas para cada implementación de Ocultar

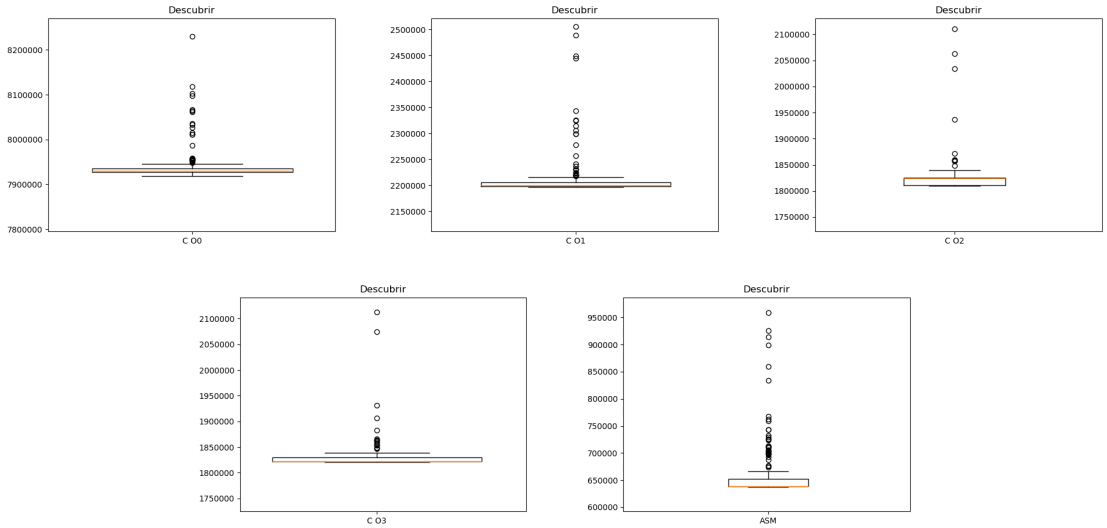

Figura 9: Boxplots de las muestras obtenidas para cada implementación de Descubrir

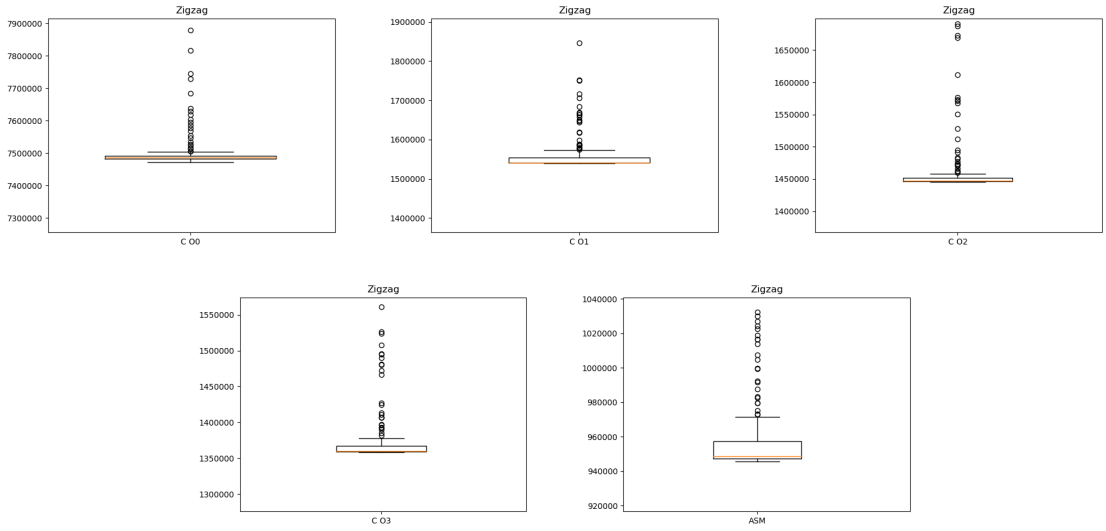

Figura 10: Boxplots de las muestras obtenidas para cada implementación de Zigzag

Imagen de 800x450: Para lograr una medida confiable, afectada en la menor cantidad posible por ruido producido por las tareas del Sistema Operativo, realizaremos veinte iteraciones con cada implementación y de ellas nos quedaremos con la mediana. Consideramos que este valor nos provee de un buen nivel de seguridad para la obtención de resultados fieles a los experimentos y a la exclusión de posibles outliers presentes en las muestras.

Además, para el caso de la implementación en el lenguaje C se utilizan los distintos flags de optimizaciones del código O0, O1, O2,O3. En la figura 11 se exhiben los resultados obtenidos:

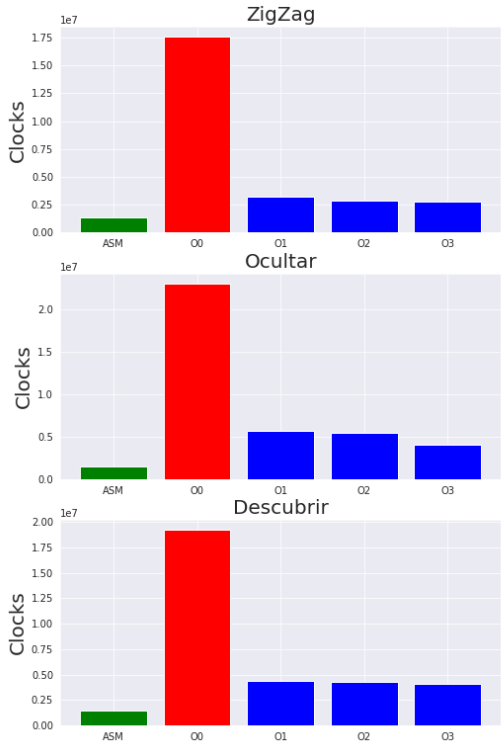

Figura 11: Comparaciones de optimizaciones

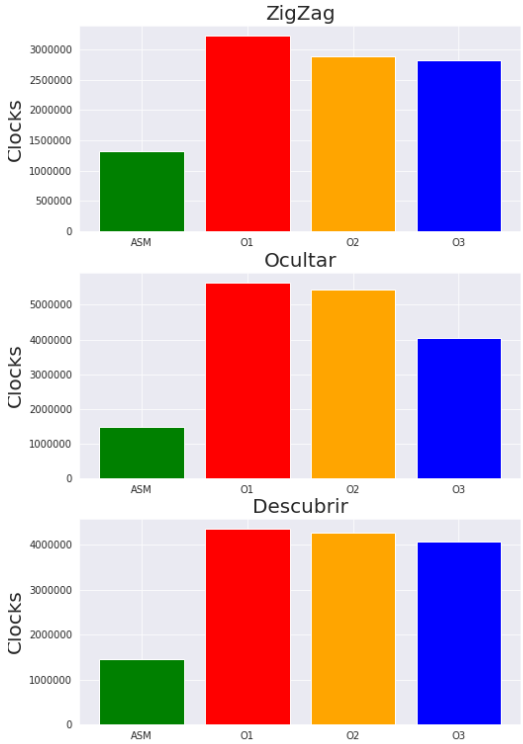

Figura 12: Comparaciones de optimizaciones sin O0

Como podemos apreciar en la figura, independientemente del filtro aplicado, observamos el mismo patrón con cada una de las opciones de optimización. La el flag O0 demuestra ser la peor de las optimizaciones (o, mejor dicho, no optimización). Las implementaciones de Assembler logran aproximadamente un 8 % de su valor, mientras que los demás flags alrededor de un 18 %. Para lograr un mejor análisis de las demás opciones, decidimos incluir el gráfico 12 en el que no aparece el flag O0. Aquí podemos ver las diferencias de forma más clara. Entre el flag O1 y O2 no se presentan grandes diferencias. La mayor diferencia entre estos flags se puede ver en el filtro ZigZag donde O2 llega a ser cerca del 83 % de O1. En cuanto a O3, como es de esperarse, es el flag que presenta la mayor mejora de performance. De entre los tres filtros, Ocultar es donde se nota la mayor diferencia del flag O3 en contraste con las otras dos opciones, logrando una reducción de cerca de un 25 % de clocks comparado con O1.

Si bien los distintos flags presentan ciertas mejoras al cambiar a mayor número de optimización, ninguno se le acerca a la implementación en Assembler de los filtros. La misma alcanza cerca del 50 % de los clocks de O3 en el filtro de ZigZag y cerca de un 37,5 % en Ocultar y Descubrir. En conclusión, los flags de optimización son muy eficaces para mejorar los tiempos de ejecución de un código comparados con no optimizar en lo absoluto, pero no lograr a ser tan eficientes como el procesamiento vectorial dado por el modelo SIMD.

4. Experimentos y Resultados

Para esta sección, decidimos dejar el flag O3 a la hora de compilar el código. Como vimos en la sección de comparación, las diferencias entre el código en C y el de Assembler serían demasiado grandes como para obtener buenos resultados a la hora de experimentar.
Llevaremos a cabo dos experimentos: uno que involucra a los filtros Ocultar y Describir y el otro solo a ZigZag.

4.1. Experimento Descubrir y Ocultar

Accesos alineados a memoria: Comparación acceso alineado contra desalineado El experimento de este filtro consiste en realizar dos implementaciones diferente de los filtros Descubrir y Ocultar que poseen varias mascaras utilizadas en cada ciclo principal de su implementación en ASM , una versión con accesos a memoria desalineados en cada iteración y el otro con accesos alineados. Partimos de la siguiente hipótesis: El accesos alineado permite una mayor velocidad de obtención de los datos de memoria para todas las mascaras lo cual indicaría que, a mayor cantidad de accesos alineados nos mejoraría considerablemente el tiempo de ejecución cada implementación en Assembler,utilizando el modelo de procesamiento SIMD. El experimento consistió en tomar 200 muestras de cada implementación, cada muestra consistía en la ejecución de la implementación correspondiente y la toma de tiempo sobre las misma (cantidad de ciclos de clock) y luego al finalizar se grafican boxplots de las muestras obtenidas que se presentan a continuación.

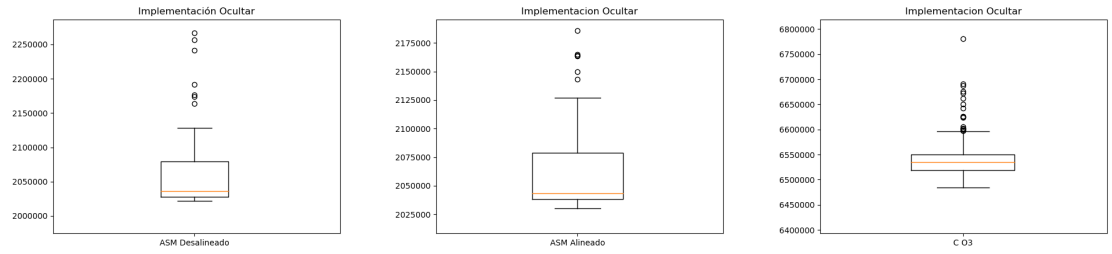

Figura 13: Boxplots de las muestras obtenidas para cada implementación de Ocultar

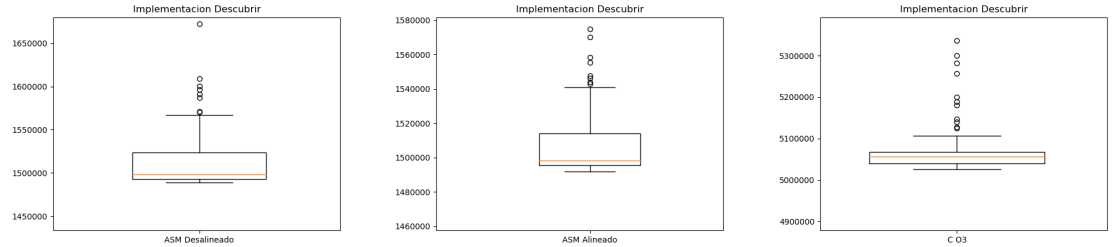

Figura 14: Boxplots de las muestras obtenidas para cada implementación de Descubrir

Como se puede apreciar en la Figura 13 y la Figura 14 de las muestras obtenidas, se ve una diferencia considerable a la hora de trabajar con memoria alineada, sobre todo en el filtro Descubrir y esto es únicamente a nivel de mascaras utilizadas en las implementaciones en ASM de Descubrir y Ocultar. Con lo cual podemos concluir que si trabajáramos con imágenes que estén alineadas en memoria, la diferencia seria óptima en las implementaciones en el lenguaje Assembler, utilizando el modelo de procesamiento SIMD.

4.2. Experimento Zigzag/Ocultar

Relación costo temporal - cantidad de instrucciones/accesos a memoria El propósito de este experimento consiste en realizar un análisis sobre el peso que tiene una mayor cantidad de instrucciones y accesos a memoria sobre la performance del filtro ZigZag. Para lograr esto, involucramos al filtro Ocultar, con el cual buscamos tener una medida de comparación entre filtros del mismo lenguaje.

Primero realizamos un conteo de las instrucciones utilizadas en los ciclos principales de los filtros de ASM:

- Ocultar cuenta con 68 instrucciones en su ciclo principal (instrucciones que operan sobre 4 pixeles por iteración) de las cuales 18 son accesos a memoria.
- Zigzag cuenta con 54 instrucciones en su ciclo con mayor costo operacional. De estas 54, 7 son accesos a memoria. En sus dos ciclos más simples, solo se requieren 6 instrucciones. Como la cantidad de filas de los bordes es constante, no les damos mayor importancia.

El filtro Ocultar no solo utiliza 14 instrucciones más que ZigZag sino que realiza más del doble de accesos a memoria. En las figuras 15 y 16, podemos apreciar una diferencia significativa entre el tiempo que tarda el filtro Ocultar y ZigZag. Estos datos fueron tomados en base a una imagen de 2048x1200 pixeles y 20 iteraciones por cada filtro.

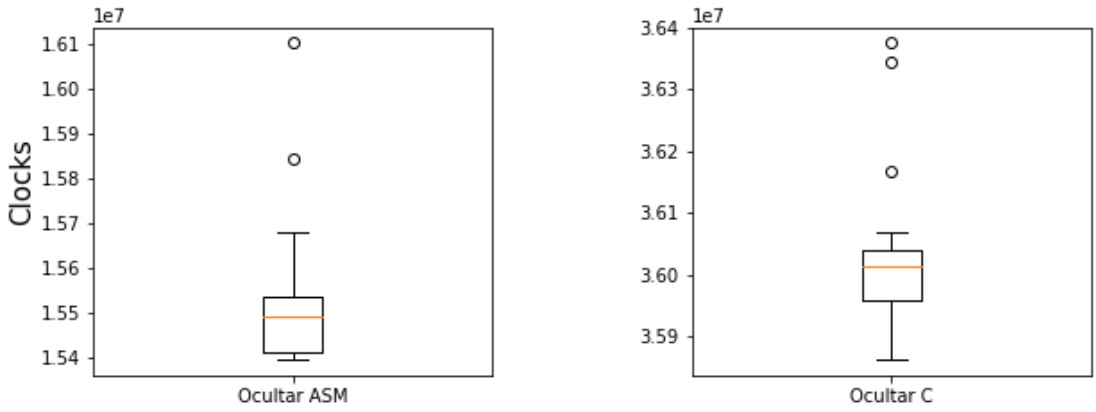

Figura 15: Comparción del costo de Ocultar en ASM y C

Ahora nos proponemos a llevar a cabo un estudio sobre el costo de los accesos a memoria. Para esto, realizamos un cambio en nuestra implementación de ZigZag de tal forma que realice más accesos a memoria de los necesarios. En la figura 16, observamos los resultados a modo de boxplots. Estos, muestran como se distribuyen las cantidades de clocks obtenidas en las 20 iteraciones. Como se puede ver, como consecuencia de una mayor cantidad de accesos a memoria hay un empeoramiento de la performance del filtro. La mediana de la muestra de mayor cantidad de accesos a memoria presenta un incremento de 1,36 %. Si bien este número no parece significativo, la ejecución prolongada de algún programa que utilice el modelo tratado en este trabajo, podría llegar una perder una valiosa cantidad de tiempo por realizar accesos innecesarios.

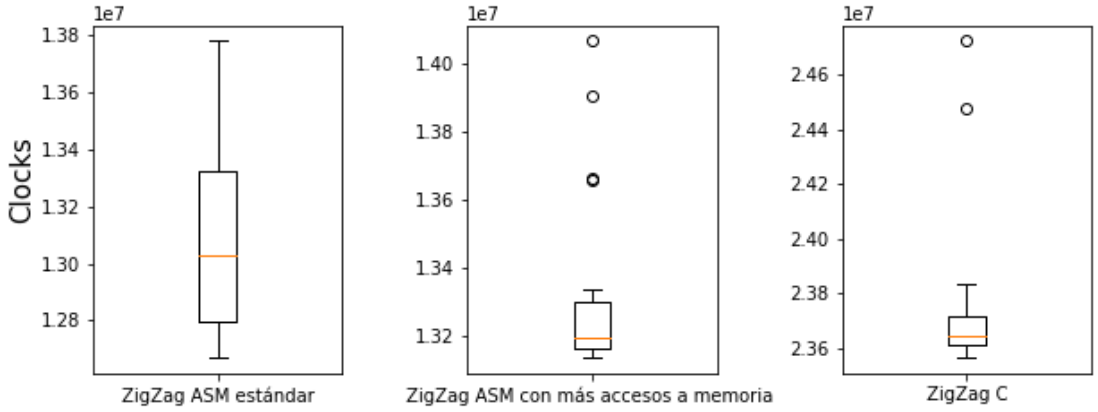

Figura 16: Comparaciones de implementaciones de ASM con diferentes accesos a memoria y su implementación en C

Conclusión Los accesos a memoria conllevan un cierto costo temporal más elevado que las operaciones habituales. Las implementaciones en lenguaje C de los filtros discutidos realizan una gran cantidad de lectura y escritura de y a memoria. A diferencia con el modelo SIMD utilizado en las implementaciones de ASM, las operaciones que realizan las implementaciones de C tiene un comportamiento más atómico en cuanto a que tratan con una pequeña cantidad de datos la vez. Si bien la optimización del compilador permite la autovectorización de los datos, cae en el programador la responsabilidad de hacer un uso eficiente de la vectorización de los datos.

5. Conclusión

A lo largo de este trabajo, se logró comprobar ciertos aspectos de la programación vectorial a bajo nivel y su perfomance en una variedad de casos. En términos generales, llegamos al hecho de que el modelo SIMD es una poderosa herramienta de procesamiento de datos vectorizables (en nuestro caso, imágenes).

Contrastándolo con las implementaciones en lenguaje C, hemos comprobado su gran potencial ya que, en todos los casos analizados, la implementación en ASM resulta la ganadora por un gran margen. Como trabajo futuro, a nuestro grupo le interesaría indagar más en el funcionamiento del flag O y como este cumple su función de optimizado.

Además de las comparaciones C-ASM, también vimos como diferentes implementaciones dentro del mismo lenguaje afectan a la eficiencia de las mismas. Como se esperaba, el acceso a memoria de manera desalineada mediante las instrucciones de ASM tienen un costo de tiempo no insignificante, por lo que el programador astuto debería tener en cuenta este tipo de accesos a memoria en la mayor medida posible.

Para concluir, queremos hacer hincapié en la importancia de tener en mente este modelo de ejecución a la hora de lidiar con datos que pueden tomar la forma de un vector, ya que si no, se reduciría considerablemente la eficacia en cuanto a tiempo de procesamiento de nuestro algoritmo al no aprovechar tan útil herramienta. Con los resultados obtenidos comprobamos que efectivamente el uso de la programación vectorial tiene una gran performance en varios aspectos y su uso para procesamiento de datos que presenten este atributo es altamente recomendado.