DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

SIMD

Organización del Computador II

Segundo Cuatrimestre a Distancia 2020

Integrante	LU	Correo electrónico
María Belén Páez	78/19	bpaez2@gmail.com
Federico Hernán Suaiter	37/19	fsuaiter@dc.uba.ar
Tomás Sujovolsky	113/19	tsujovolsky@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 exploraremos los usos y las ventajas del procesamiento vectorial de la SIMD de Intel. Para ello realizamos tres algoritmos que se encargan de aplicarles un filtro a una imágen en particular. Estos filtros son: Imágen Fantasma, Color Bordes y Reforzar Brillo.

Índice

1. Introducción

El objetivo de este Trabajo Práctico es estudiar las ventajas y/o desventajas del procesamiento vectorial de la arquitectura x86 mediante la implementación de algoritmos a bajo nivel en lenguaje assembly y comparándolo con C. La ventaja visible del procesamiento vectorial es la capacidad de trabajar simultáneamente con mas de un dato, acelerando así el rendimiento de los algoritmos. Sin embargo, a lo largo del trabajo estudiaremos qué costos tiene utilizar esta metodología y cuáles son las maneras más apropiadas de usarla.

Abordaremos esta temática mediante la implementación de filtros de imágenes. Las imágenes están compuestas por píxeles de 32 bits. Cada uno posee cuatro componentes de 8 bits cada uno, encargados de darle valor en entero sin signo a la cantidad de rojo (R), azul (B), verde (G) y al nivel de transparencia (A) que tiene la imagen. Este último debe mantenerse como el máximo posible dentro del rango de lo enumerable con 8 bits, es decir 0xFF, si lo notamos en hexadecimal. A continuación, procedemos a describir nuestras implementaciones en assembler, para luego explicar nuestros experimentos y poder sacar conclusiones.

2. Desarrollo

2.1. Algoritmos

2.1.1. Imagen Fantasma

Cómo organizamos nuestras idas a memoria:

Este filtro, tiene la peculiaridad de que, dado el pseudocódigo, podemos inferir que para duplicar el tamaño de la imagen, vamos a requerir solo un cuarto de la imagen original. Dado que los indices de la matriz, i y j, son enteros, la división por dos que se da para calcular ii y jj (los indices del desplazamiento) es entera. De esta forma, para cada cuatro píxeles, hay un píxel que les corresponde, sobre el cual se hace el cálculo del brillo. Al conjunto de píxeles que se les busca el valor del brillo, los llamaremos imagen desplazada. Antes de iniciar el ciclo, calcularemos dónde se posicionará (según los offsets pasados por parámetro) y usaremos $\frac{ancho}{2}$ como el offset para saltar de fila en la imagen desplazada.

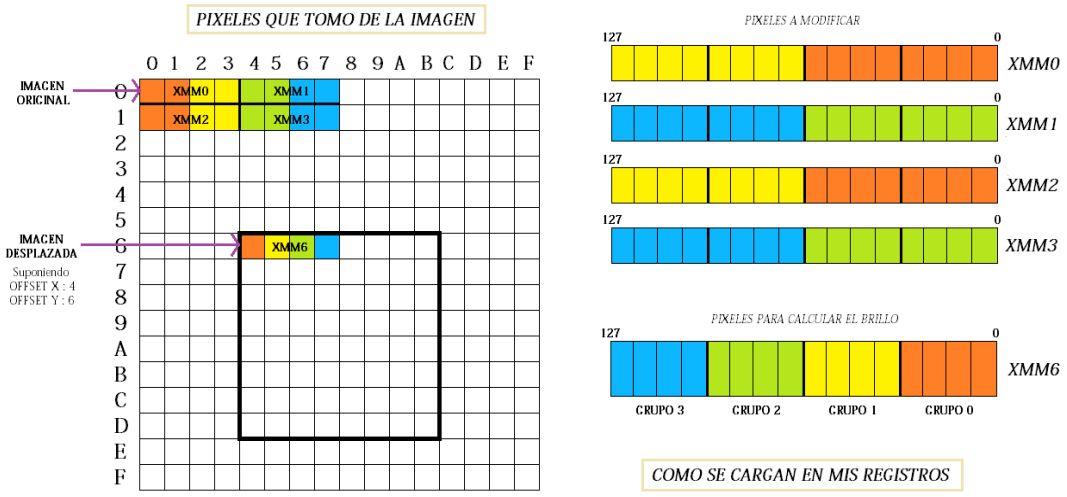


Figura 1: Cargado de memoria a los registros

Sin dar más vueltas, en la figura 1 mostramos cómo hicimos para "dividir" la imagen en grupos de cuatro pixeles y cómo se dan las distintas correspondencias, usando colores para referirnos a los grupos

y a los pixeles desplazados con los que se relacionan. Tal como se puede ver, ya que desde la imagen original traemos cuatro pixeles consecutivos, en cada registro XMM quedarían dos pixeles correspondientes a un grupo y dos a otro. Mientras que, al traerlos desde la imagen desplazada, cada pixel corresponde a un grupo distinto.

Cómo funciona nuestro ciclo:

Calculo del brillo

Dentro de nuestro ciclo, lo primero que hacemos es calcular el b, usando el vector de los cuatro píxeles de desplazamiento. Para ello, dado que usaremos sumas horizontales, seteamos temporalmente los A (los niveles de transparencia) en 0, ya que lo que nos interesa sumar son los R, G y B solamente. Tras extender cada componente a tamaño word, usando una mascara y PAND, me guardo los valores de G de los píxeles del desplazamiento y se los sumo a si mismos de manera saturada. Luego, realizo un par de sumas horizontales para tener tanto en la parte baja como en la parte alta, las sumas de las componentes (con A = 0) de cada píxel, siendo el píxel 0 el que se encuentra en la parte menos significativa del registro. Para llevar a cabo las divisiones de estos valores, lo extendemos a tamaño dword y lo convertimos a floats. Así, cada suma es flotante y las podemos dividir por ocho (que es dividir por 4, en el calculo del brillo, y luego por dos, al sumarlo para pasar el píxel a memoria), sin perder precisión.

Separación en casos

Si bien a todos los registros XMM donde guardamos los pixeles de la imagen original les aplicamos las mismas operaciones, debemos separarlos en casos (donde cada registro es un caso). Por ejemplo, inicialmente teníamos en el registro XMM0 a los primeros cuatro pixeles. Para extender cada componente a dword y convertirlo en float, debemos guardar un pixel por registro. De esta manera, guardamos el pixel 0 en XMM6, el 1 en XMM5, el 2 en XMM7 y el 3 en XMM0. (Sí, podría haber estado en orden para ser mas facil de recordar). Una vez que tengo cada pixel por registro, multiplico cada componente por 0.9. Dado que en cada registro inicialmente dos de los pixeles correspondían a un grupo y dos a otro, tomo a XMM6 y XMM5 por un lado, y a XMM7 y XMM0 (siguiendo al ejemplo) por otro. A cada par les sumo el b correspondiente, usando shuffle para guardarlo desde el vector de b's y posicionarlo en el lugar de todas las componentes, menos en A. Para entenderlo mejor:

```
MOVUPS XMM11, XMM14                 ; en xmm14 tengo un vector de 0s de tipo float
SHUFPS XMM11, XMM4, 00000000B       ; XMM11 : | B.0 | B.0 |  0  |  0  |
MOVUPS XMM13, XMM11                 ; XMM13 : | B.0 | B.0 |  0  |  0  |
MOVHLPS XMM11, XMM11                ; XMM11 : | B.0 | B.0 | B.0 | B.0 |
SHUFPS XMM11, XMM13, 00110000B      ; XMM11 : |  0  | B.0 | B.0 | B.0 |
ADDPS XMM6, XMM11
ADDPS XMM5, XMM11
```

Luego de sumar el brillo, convertimos las componentes a enteros de 32 bits y usando PACK, volvemos a unir los píxeles hasta que cada componente vuelva a ser de un byte. Entonces, seteamos las A en 0xFF y lo muevo a la posición de memoria del destino, la cual varía según el caso en el que estoy, pero que cuyos offset coinciden con los utilizados al traer los datos de memoria.

Terminación del ciclo

Al final del ciclo, avanzamos ocho pixeles en la imagen original (la fuente y la destino); y cuatro en la imagen desplazada. Para poder ver si llegamos al final de la fila, contamos los pixeles recorridos en la fila actual y lo comparamos con el ancho pasado por parámetro. Si no lo superamos, volvemos al ciclo; si no, reiniciamos el contador, le sumamos dos al contador de filas (ya que en el ciclo operamos con dos filas a la vez) y lo comparamos con el alto, pasado como entrada. Si es igual, salimos del ciclo; si no lo es, le sumamos el offset al puntero de la dirección de la imagen desplazada y avanzamos los de la imagen fuente y destino a la fila siguiente.

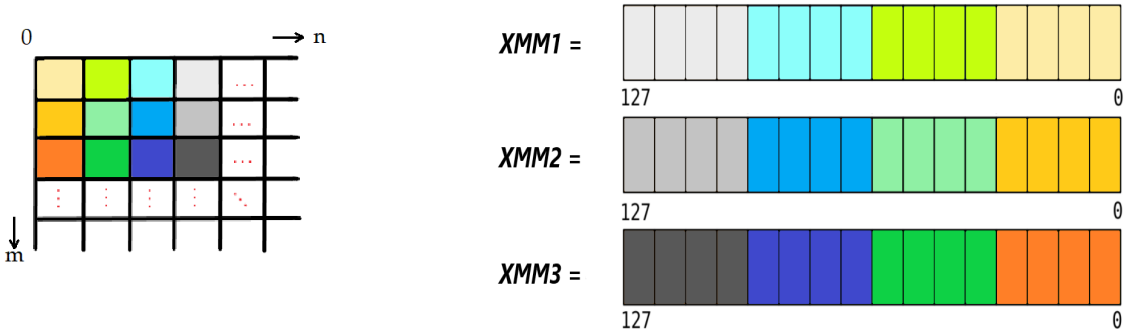
2.1.2. Color Bordes

Una iteración del ciclo principal de nuestro algoritmo:

En una operación del ciclo principal (al que en el código llamamos “cicloAltura”) comenzamos aumentando el contador del mismo, y luego comparando el contador principal con la altura-1.

> Si efectivamente son iguales, esto significa que llegamos al final de la matriz, y por lo tanto debemos saltar a la última parte del código.
> De no ser así, continuamos...

Continuando, restauramos el valor del ciclo interior (al que en el código llamamos “cicloAncho”) y procedemos al mismo. Una vez dentro de nuestro ciclo interior, reiniciamos los valores de XMM0, registro que usaremos para luego volcar los datos a memoria, y traemos de memoria los datos con los que voy a operar.



(a) Píxeles en la matriz

(b) Píxeles cargados en los registros

Con estos 12 pixeles que acabamos de traer de memoria, operaremos sobre 2 de ellos. Para esto, haremos cuentas con los pixeles que se encuentran alrededor de nuestro pixel objetivo (C8 adyacentes). Estos pixeles objetivo son los dos que se encuentran rodeados por el resto de los colores (en la imagen, estos son el verde agua y el azul claro).

Expandiremos cada pixel que se encuentra representado por 4B (32bits) a 8B (64bits). Ahora, en lugar de que cada parte del pixel este representado por un byte, esta representado por una word. De esta manera puedo operar sin perder precisión al realizar restas (Por ejemplo, al restar, no podemos representar -255 con 8 bits).

En un momento debemos hacer:
abs(src_matrix[i][j-1].r - src_matrix[i][j+1].r)
abs(src_matrix[i][j-1].g - src_matrix[i][j+1].g)
abs(src_matrix[i][j-1].b - src_matrix[i][j+1].b)

Esto implica que debemos restarle las componentes r,g y b del pixel ubicado en la columna anterior, el pixel de la columna posterior. Después debemos calcularle el absoluto. Para lograr esto, efectuamos el cálculo en simultaneo en los registros XMM, para poder operar con ambos pixeles objetivo al mismo tiempo. De esta manera, por ejemplo, cuando calculamos el ABS sobre el registro XMM, realizamos una sola operación que se aplica a ambos pixeles.

Una vez calculadas las restas, los valores absolutos y las sumas correspondientes con saturación, en el registro XMM0 se va a tener los 2 valores a retornar a la imagen destino. Antes de esto, se aplica una máscara para mantener las transparencias con 0xFF, tal como se pide. Ahora sí, se mueven los 2 pixeles a la imagen destino.

Luego de mover estos valores a memoria, se modifican los punteros y el registro contador del ciclo interior. Y se vuelve a iterar sobre el ciclo actual.

Una vez se recorrió toda la fila, se termina de recorrer el ciclo interior. Aquí se procede a ubicar 2 píxeles blancos en los bordes de la imagen y se modifican los punteros.

Partes que nos interesa destacar:
Al comienzo del código, tenemos un pequeño ciclo, en el cual ubicamos píxeles blancos en la primera fila de la imagen destino. Aprovechamos esto y dejamos el puntero a la imagen destino donde lo necesitamos a la hora de comenzar con el desarrollo de la imagen.

Dentro del ciclo principal de la función, para ser más exactos, dentro del cicloAncho que se encuentra dentro del cicloAltura, no se realizan ciclos. En el pseudocódigo que nos brindan, nos dan a entender que debemos hacerlos, pero preferimos evitarlos ya que estos jumps pueden perjudicar los tiempos de ejecución debido a que ocasionan la discontinuidad de la Pipeline.

Aprovechamos y cuando se termina la fila, en la imagen destino colocamos el ultimo pixel de la fila N y el primer pixel de la fila N+1 en blanco. De esta manera, evitamos tener que hacer un ciclo luego para tener que ubicar los pixeles de la imagen en blanco. (Misma idea de antes, para evitar romper la continuidad de la Pipeline).

2.1.3. Reforzar Brillo

El algoritmo de Reforzar Brillo aumenta el brillo de las partes más luminosas y lo disminuye en las partes más oscuras, en base a los criterios pasados por parámetros. Estos son los siguientes: Umbral Superior, Umbral Inferior, Brillo superior y Brillo inferior. Cuando el brillo de un pixel se excede del umbral superior se aumentan sus componentes R, G y B por el brillo superior. Caso contrario, si el brillo de un pixel es menor al umbral inferior se disminuye sus componentes R, G y B por el brillo inferior. El algoritmo escrito en assembler hará esta metodología para cuatro píxeles a la vez.

Antes de comenzar con el ciclo, pasamos las mascaras de memoria a registros para no acceder a memoria cada vez que se las quieran usar. Luego, XMM12, mediante un XOR consigo mismo, se setea el vector cero que se usara para comparar brillos mas adelante. Por ultimo, en XMM0 hasta XMM4, se setean los parámetros. Esto lo hacemos de dos maneras distintas, los brillos a sumar y restar los pasamos con un pin y un shuffle. Los umbrales se pasan a registros utilizando un espacio reservado en memoria.

Una vez preparados todos registros que no se alterarán en el ciclo, se inicializa el ciclo multiplicando el largo de la imagen por su ancho, dando la totalidad de píxeles, y dividiéndolo por 4 ya que procesamos 4 píxeles a la vez. RCX se utilizara como contador del ciclo.

El ciclo del algoritmo consiste de 5 partes:

La primer parte es cargar los 4 pixeles indicados por la dirección de memoria de la imagen original y duplicarlos en otro registro. Uno de ellos se utilizará para los cálculos de los brillos de los pixeles, mientras que el otro será para retornar los píxeles alterados.

La segunda parte del ciclo es calcular los brillos de los pixeles. La metodología es similar al cálculo del brillo en Imagen Fantasma. Las diferencias son no dividir por 8, si no por 4 y no convertir a float. Al finalizar las cuentas de los brillos, los guardamos en dos registros diferentes.

La tercera parte del ciclo consiste en conseguir mascaras que indiquen qué píxeles modificar mediante los brillos calculados. Esto los hacemos con las siguientes funciones:

```
PSUBD XMM6, XMM2        ; restamos el umbral superior al brillo calculado
PCMPGTD XMM6, XMM12 ; aquellos números que sean mas grandes que cero, indicaran qué
                        pixeles modificar por umbral superior
MOVDQU XMM5, XMM3     ; movemos el brillo inferior a un registro de uso temporal
PSUBD XMM5, XMM7      ; restamos al brillo inferior el brillo calculado
PCMPGTD XMM5, XMM12 ;aquellos números que sean mas grandes que cero, indicaran qué
                        píxeles modificar por umbral inferior
```

Una vez armadas las mascaras que indican qué pixeles modificar, la siguiente parte del algoritmo se ocupara de aplicárselas los vectores que guardaban los brillos que había que sumar y restar. Utilizando, XMM7 como registro temporal, se traslada el registro de brillo a sumar (o restar) a XMM7 y se le aplica la mascara calculada que corresponde. Luego, se suma (o resta) XMM7 (con la mascara ya aplicada) al registro que tenía los 4 píxeles originales inalterados.

Por ultimo, se guardan los 4 píxeles modificados en la dirección indicada por RSI, se aumenta RSI y RDI por 16 bytes ya que cada píxel ocupa 4 bytes, se decrementa RCX en 1 y se lo compara con cero. Si RCX es cero, indica que que ya se procesaron todos los píxeles por lo que debe terminar el algoritmo. Si es distinto a cero, quiere decir que todavía hay píxeles por analizar y debe volver a realizar el ciclo.

3. Resultados

3.1. Análisis preliminar

3.1.1. Método de evaluación

El método utilizado para todas las evaluaciones de los algoritmos y los experimentos fue el siguiente: Utilizando la biblioteca ”time.h.[en] una versión modificada de tp2.c, se registra el clock antes y después de aplicar el filtro, se restan los clocks al empezar a los clocks al finalizar y se los divide por la cantidad de clocks por segundo. Se registra el tiempo en una lista de tipo double (ordenada de menor a mayor). Este proceso se repite 10000 veces. Luego, sobre esa lista se calcula el promedio y el desvío estándar para el 97 % de los datos, sin tomar los mas grandes, eliminando del calculo los outliers. La modificación de tp2.c estará detallada en el apéndice.

Los experimentos se corrieron en 2 maquinas distintas que denominaremos Maquina 1 y Maquina 2.

Maquina Nº1
La VM tiene:

- 8192MB de RAM
- 4CPUs con un límite de ejecución del 100 %
- 16MB de memoria de video

La computadora principal tiene:

- Windows 10 Pro - 64 bits
- 16384MB de RAM
- Procesador Intel I5-10210u
- (8CPUs), $\approx$ 2.1GHz
- Placa de vídeo integrada

Maquina Nº2
La VM tiene:

- 9864MB de RAM
- 2CPUs con un límite de ejecución del 100 %
- 16MB de memoria de video

La computadora principal tiene:

- Windows 10 Pro - 64 bits
- 16384MB de RAM
- Procesador AMD A10-7850K Radeon R7
- 12 Cores 4C+8G (4CPUs), $\approx$ 3.7GHz
- Placa de video: AMD Radeon(TM) R7 Graphics (Integrada | 2 GB de VRAM asignados)

3.1.2. Comparaciones

Antes de realizar las mediciones, esperábamos que los algoritmos en ASM fueran los mas rápidos, seguidos de los de C, con las distintas optimizaciones, en el siguiente orden: -O3, -O2, -O1, -O0. Al correr los algoritmos sobre la imágen IndianaJones.bmp, dada por la cátedra, con el método de evaluación en la Máquina Nº1 estos fueron los resultados (los números exactos y los desvíos estándar se encuentran en el apéndice):

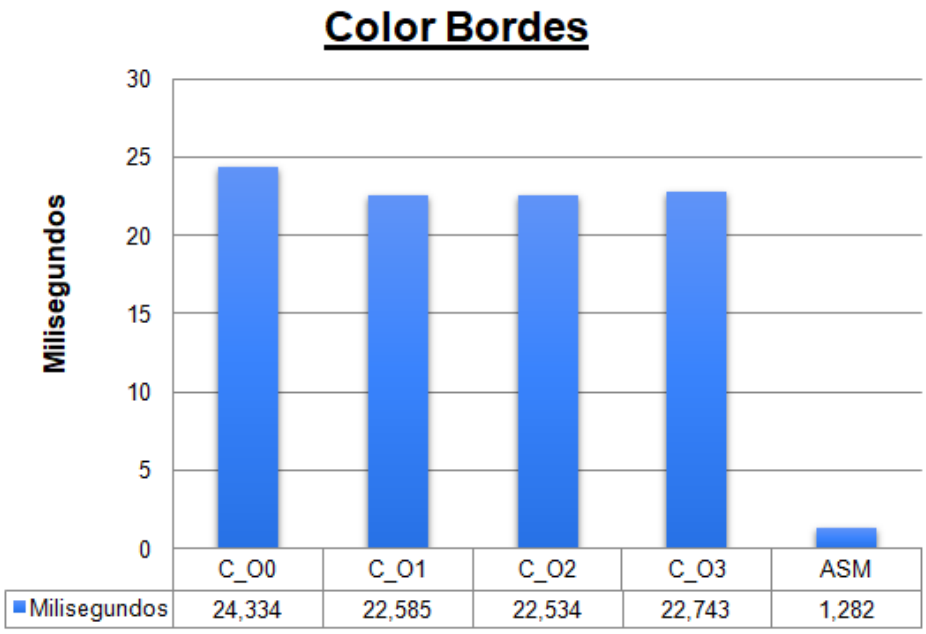


Figura 3: Promedio en milisegundos de Color Bordes IndianaJones.bmp

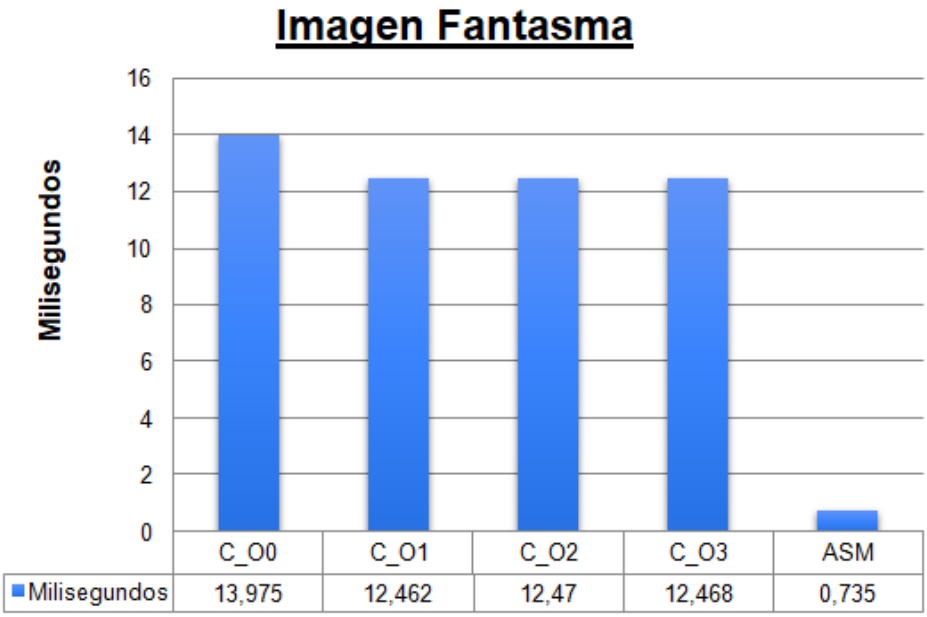


Figura 4: Promedio en milisegundos de Imagen Fantasma IndianaJones.bmp 30 30

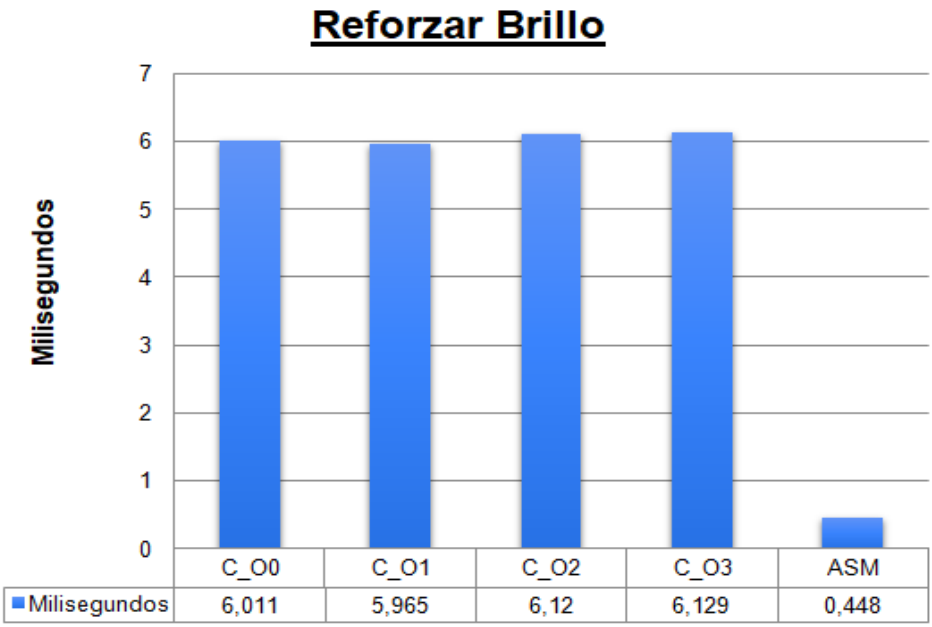


Figura 5: Promedio en milisegundos de Reforzar Brillo IndianaJones.bmp 100 50 30 15

Si bien los algoritmos en ASM fueron mas rápidos como esperábamos, los resultados de las distintas optimizaciones en C nos sorprendieron. En el algoritmo de Reforzar Brillo, no solo no hubo cambios significativos, si no que también la optimización -O3 fue la más lenta. En los otros dos algoritmos, la -O0 fue la marcadamente más lenta, como se esperaba, mientras que las otras optimizaciones se comportaron de manera similar entre sí.

Nuestra teoría acerca de la razón del comportamiento inesperado en el caso del algoritmo de Reforzar Brillo, es que las optimizaciones que se proponen implican modificar cosas innecesarias, ya que su cambio no optimiza el rendimiento. Nos parece que debe ser así, ya que el código es relativamente corto y de baja complejidad.

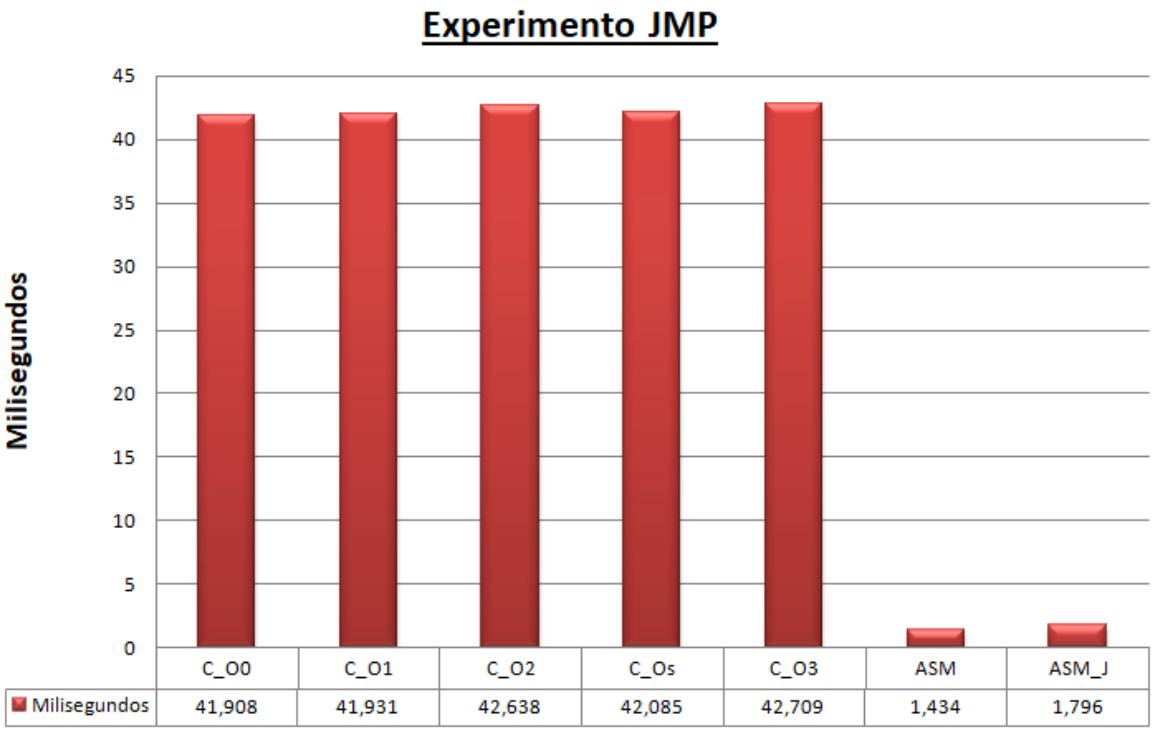
Con respecto a los otros algoritmos, creemos que debe existir una modificación presente en -O1, y por lo tanto en -O2 y -O3, que sí implican una posibilidad de mejorar el rendimiento de los algoritmos con respecto a -O0.

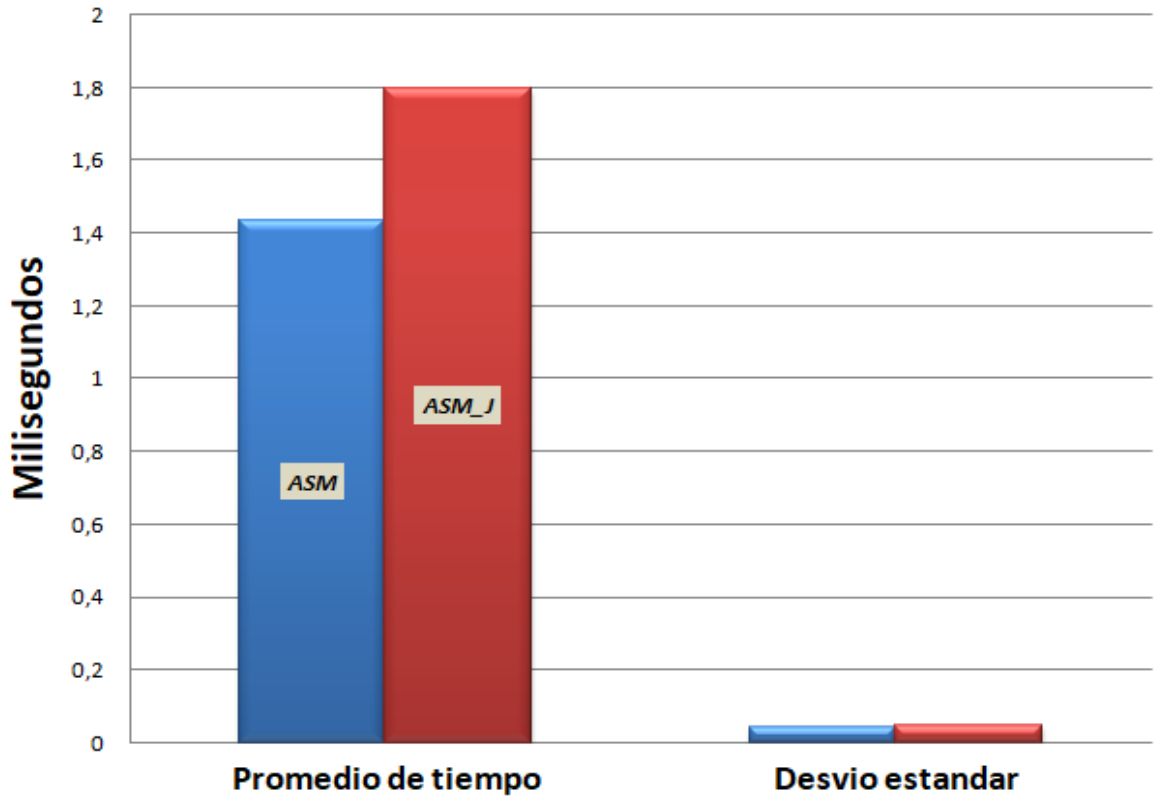
3.2. Experimento Nº1: Los saltos y la pipeline

En este experimento vamos a ver cómo los saltos (JMPs) dentro de nuestro código afectan la continuidad de la pipeline y, como consecuencia de esto, el tiempo que demoran en ser procesados es mayor.

Para lograr esto vamos a comparar el código que implementamos nosotros en ASM con el mismo código modificado para que tenga 6 saltos dentro del ciclo interior. El único objetivo es el de analizar si hay perdida de perfomance con los saltos. Vamos a llamar a este código ASM_J para diferenciarlo del original. En cierto modo, buscamos con estos saltos, que se parezca al pseudocódigo brindando por la cátedra.

Como hipótesis, por lo visto en las clases teóricas, hemos de suponer que nuestro código ASM_J deberá tardar más que nuestro código ASM debido a lo mencionado anteriormente. Veamos que ocurre... (Los gráficos se pueden encontrar en la página siguiente)


Experimento JMP
Milisegundos
45
40
35
30
25
20
15
10
5
0
C_O0
C_O1
C_O2
C_Os
C_O3
ASM
ASM_J
Milisegundos
41,908
41,931
42,638
42,085
42,709
1,434
1,796


Milisegundos
2
1,8
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
ASM
ASM_J
Promedio de tiempo
Desvio estandar

Para este experimento se utilizó la Máquina Nº2

Al probarlo con imágenes cuyo alto es mayor que su ancho, logramos ver que la diferencia entre el promedio se mantiene estable, por lo tanto hacer pruebas extras no aporta mayor información.

Por lo obtenido en la experimentación, podemos concluir que nuestra hipótesis es correcta.
En el primer gráfico podemos ver la comparación entre C (y sus optimizaciones), ASM y ASM_J. En el segundo gráfico podemos ver que, como habíamos predicho, el código ASM_J demora más en ser ejecutado que el código ASM. El desvío estándar de ambos es igual, por lo tanto la diferencia del promedio se puede apreciar con mayor exactitud.

Nosotros suponíamos que la diferencia entre ambos códigos de ASM iba a ser mayor. Luego de reflexionar sobre los resultados, podemos observar que el porcentaje de diferencia entre ASM y ASM_J es del 20 %. Es decir, el código ASM_J es un 20 % más lento que el código ASM. Podemos decir ahora con certeza, y afirmando lo que vimos en las clases teóricas de la materia, que el uso de saltos perjudica la 'performance' de nuestra ejecución.

3.3. Experimento Nº2: Funciones inapropiadas

En SIMD de intel hay funciones especificas para enteros y números de punto flotante. Dentro de estas funciones diferentes, hay incluso algunas que tienen un comportamiento similar. En clase, escuchamos que usar estas funciones intercambiadamente traen consigo penalidades, ya que la CPU tiene que cambiar qué tipo de datos tiene guardado en los registros. Estudiemos entonces las penalidades que implica usar un función inapropiada en ASM.

Para esto, cambiamos la función que utilizamos para mover los píxeles en la parte alta a la parte baja en Reforzar Brillo (llamaremos ASM_I a el código modificado), shift de 8 bytes (PSRLB) por move high to low packed singles (MOVHLPS, que se usa para datos empaquetados de punto flotante) y estudiamos los resultados. Esperamos una muy pequeña diferencia a favor de PSRLB, ya que si bien sabemos que usar las funciones inapropiadas implican penalidades, el cambio es de una sola función que se usa una vez por ciclo. Corramos el método de evaluación en la Maquina Nº1 y analicemos los resultados:

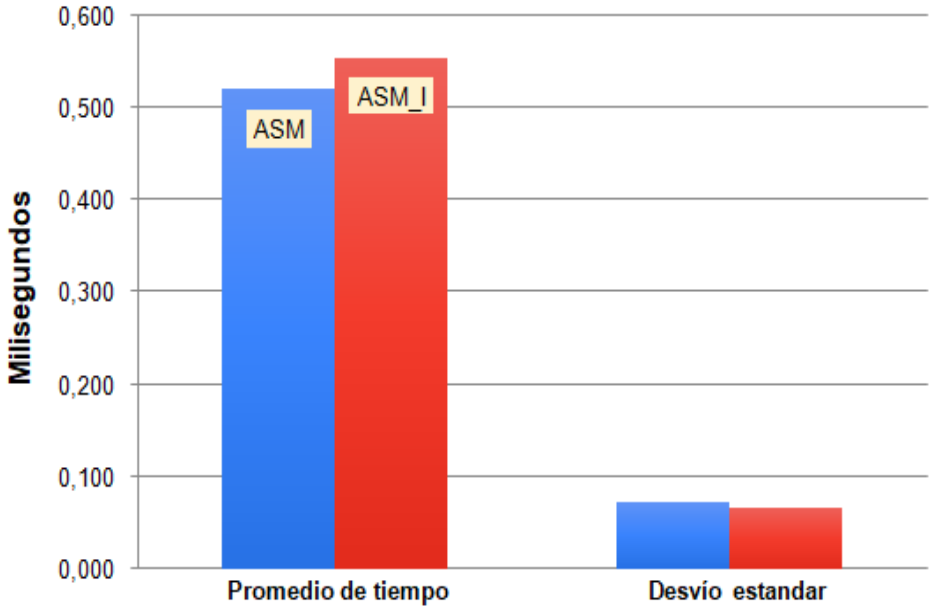


Figura 6: Promedio en milisegundos de Reforzar Brillo IndianaJones.bmp 100 50 30 15

La diferencia que se puede apreciar en los gráficos es de un 5.98 %. Esto quiere decir que aplicar la función inapropiada trajo consigo una baja promedio en el rendimiento del 6 %.

Dentro del grupo hubo distintas opiniones acerca de la relevancia de esta baja del 6 %. Es mucho mas grande de la que esperábamos, ya que esperábamos un perjuicio de hasta el 1 % o 2 %. Esto implica que estas penalidades son verdaderamente significativas y habría que evitarlas en toda situación posible.

Algo importante a destacar es el alto desvío estándar. Este se encuentra presente en todos las corridas del método de evaluación en el algoritmo de reforzar brillo. Creemos que este se debe a que, como el procedimiento (la aplicación del filtro) es tan rápido, cualquier interrupción al procesador durante la corrida implica un desfasaje significativo del tiempo.

4. Conclusión

Problemas encontrados:

Con lo que respecta a los problemas encontrados, nos ocurrió que lo que más tiempo nos llevo fue lograr trabajar con números de punto flotante. En algunos ejercicios se trataba al brillo como un entero, mientras que en otros casos se lo trataba como un float.

Una gran diferencia entre C y ASM, es que en ASM detectar los errores es más difícil. Un simple error como olvidarse unos "[ ]" puede causar que otra parte del código explote y si uno no logra abstraerse lo suficiente puede estar un tiempo relativamente largo hasta poder encontrarlo (¡Si es que lo encuentra!).

La mayoría de los problemas básicos que podían ocurrir fueron pulidos gracias al TP1, en el cual allí los problemas que solían surgir era que un registro quedaba con basura.

Análisis y preguntas planteadas:

Llegada a esta instancia, nos vemos inmersos en la necesidad de hacer hincapié en las diferencias entre el lenguaje ensamblador y el de C. Creemos que es importante analizarlo, ya que, dadas las diferencias vistas en relación con el tiempo de ejecución de los distintos algoritmos, con el entendimiento de sus códigos y con la dificultad a la hora de escribirlos, nos encontramos en un dilema. ¿Qué factor es el que nos conviene priorizar, a la hora de elegir en qué lenguaje programar?

En principio, si hablamos de portabilidad, el lenguaje C se acerca mas a estar en ventaja, dado que los distintos microprocesadores poseen su propio lenguaje ensamblador. Como en este trabajo utilizamos el de Intel (el x86 de 64 bits) aprovechando las instrucciones que nos provee la SIMD, nuestros algoritmos no serían compatibles en todas las máquinas. Sin embargo, en ese caso, no sería una problema de nuestra implementación, sino una limitación tecnológica, y no nos parecería justo que influenciara en la elección.

Por otro lado, dado que Assembly es un lenguaje de bajo nivel, muy cercano al de máquina-humano (el binario), nos permite controlar con precisión las operaciones del procesador. Esto lleva a que el código que escribimos tenga operaciones precisas y a bajo costo de tiempo de ejecución. Sin embargo, C posee distintas bibliotecas que nos permiten acceder a funciones específicas (por ejemplo, las relacionadas a los cálculos matemáticos) y por cuestiones de comodidad, a veces nos es útil usarlas llamándolas desde ASM, ya que nos evitan tener que replicar su comportamiento con una mayor cantidad de instrucciones.

En este trabajo, dado que el objetivo es modificar imágenes y nos manejamos con una gran cantidad de datos en memoria, vamos a centrar nuestro foco en el tiempo de ejecución. Coincidimos en que las implementaciones hechas en C nos permiten entender mas fácilmente el comportamiento del algoritmo y mantenerlo (se puede revisar, modificar y reconocer errores a simple vista, dado que el lenguaje es "mas humano"). Sin embargo, vistas las comparaciones en la sección 3.1.2, nuestras implementaciones

de ASM son remarcadamente más rápidas. Sabemos, y quedó confirmado, que esto es así, dado que el uso de las instrucciones que nos provee SIMD, agilizan las idas a memoria, trayéndonos (o guardando) varios datos al mismo tiempo y permitiendo que se realicen operaciones iguales en distintos operandos a la vez.

Dentro de las ventajas del uso de SIMD, la principal sin duda es la ejecución de distintas operaciones de manera paralela. Sin embargo, el uso de jumps (saltos, ya sean condicionales o no) implicaría una interrupción en la pipeline. Según lo visto en el experimento 1, pudimos apreciar que la teoría se cumple en la práctica.

A la hora de hablar sobre utilizar operaciones para integers o floats, en el caso del filtro de la Imagen Fantasma, se pide que se conviertan las componentes (temporalmente) a valores de tipos flotante, para operar sobre ellas. Sin embargo, si quisiésemos utilizar funciones específicas para valores de punto flotante en datos de tipo entero, ocurriría lo visto en el experimento 2. Por lo tanto, ahora sabemos que usar estas funciones de manera intercambiada es un grave error debido a que se ralentiza el tiempo de ejecución.

Frente a la necesidad de optimizar los algoritmos, aparece el uso de instrucciones que requieren que la memoria esté alineada. Estas instrucciones producen interrupciones en la ejecución si la memoria no cumple con la precondición; pero en caso de que puedan usarse, aceleran el funcionamiento del algoritmo. Nos hubiese gustado poder usarlas y realizar un experimento para ver cuál era la diferencia de rendimiento; sin embargo, nada nos garantiza que las imágenes estén alineadas a memoria, por lo que en caso de usarlas, se produciría un segmentation fault.

Entonces, para terminar, creemos que dada la diferencia vista en el rendimiento de la ejecución de los algoritmos de C y los de ASM, las ventajas que aporta el uso de SIMD sobre SISD son irrefutables. Los algoritmos escritos en ASM tienen un rendimiento mucho mayor, y si bien el uso de ciertas instrucciones puede perjudicarlo (como lo vimos en los experimentos), de ninguna manera nos parece que, dadas las circunstancias de estar procesando imágenes, elegiríamos el uso del lenguaje C para implementar los filtros pedidos.

5. Apéndice

5.0.1. tp2.c de experimentación

Listas de double:

```
typedef struct s_listElem {
    double data;
    struct s_listElem* next;
    struct s_listElem* prev;
} listElem_t;

typedef struct s_list {
    uint32_t size;
    listElem_t* first;
    listElem_t* last;
} list_t;

list_t* listNew() {
    list_t* l = malloc(sizeof(list_t));
    l->first = 0;
    l->last = 0;
    l->size = 0;
return l;
}

void listAdd(list_t* l, double data) {
    listElem_t* n = malloc(sizeof(listElem_t));
    l->size = l->size + 1;
    n->data = data;
    listElem_t* prev = 0;
    listElem_t* current = l->first;
    listElem_t** pcurrent = &(l->first);
    while (current != 0 && current->data < data) {
        pcurrent = &((*pcurrent)->next);
        prev = current;
        current = *pcurrent;
    }
    n->next = current;
    n->prev = prev;
    *pcurrent = n;
    if (current == 0)
        l->last = n;
    if (current != 0)
        current->prev = n;
}

void listDelete(list_t* l) {
    listElem_t* current = l->first;
    while (current != 0) {
    listElem_t* tmp = current;
    current = current->next;
    free(tmp);
}
```

```
free(l);
}

void listPrint(list_t* l, FILE* pFile) {
    fprintf(pFile, "[");
    listElem_t* current = l->first;
    if (current == 0) {
        fprintf(pFile, "]");
        return;
    }
    while (current != 0) {
        fprintf(pFile,"%f", current->data);
        current = current->next;
        if (current == 0)
            fprintf(pFile, "]");
        else
            fprintf(pFile, ",");
    }
}
```

Funciones de estadística:

```
double promedio(list_t* l){
    double sum = 0;
    double total = (double)(l->size) * 0.97;
    listElem_t* current = l->first;
    for (int i = 0; i < (int)(total); i++) {
        sum += current->data;
        current = current->next;
    }
    return sum/total;
}

double desvio_estandar (list_t* l){
    double sum = 0;
    double total = (double)(l->size) * 0.97;
    double prom = promedio(l);
    listElem_t* current = l->first;
    for (int i = 0; i < (int)(total); i++) {
        sum += pow((current->data - prom), 2);
        current = current->next;
    }
    return sqrt(sum/total);
}
```

Modificaciones al código preexistente
Cambios en el main:

```
filtro_t *filtro = detectar_filtro(&config);

//lista de datos para experimentos
list_t* datos = listNew();
//

filtro->leer_params(&config, argc, argv);
for (int i = 0; i < 10000; ++i){
```

```
        correr_filtro_imagen2(&config, filtro->aplicador, datos);
        if (i%100 == 0){
            printf("%d\n", i);
        }
    }
    filtro->liberar(&config);

    FILE *pfile = fopen("datos.txt","w");
    listPrint(datos, pfile);
    fprintf(pfile, "\n");
    fprintf(pfile, "Promedio es:%f\n", promedio(datos));
    fprintf(pfile, "Desvio estandar es:%f\n", desvio_estandar(datos));
    fclose(pfile);
    listDelete(datos);

void imprimir_tiempos_ejecucion2(clock_t start, clock_t end, int cant_iteraciones,
list_t* l) {
    double segs = (double)(end-start) / CLOCKS_PER_SEC;
    listAdd(l, segs);
}

void correr_filtro_imagen2(configuracion_t *config, aplicador_fn_t aplicador,
list_t* l) {
    imagenes_abrir(config);

    clock_t start, end;

    imagenes_flipVertical(&config->src, src_img);
    imagenes_flipVertical(&config->dst, dst_img);
    if(config->archivo_entrada_2 != 0) {
        imagenes_flipVertical(&config->src_2, src_img2);
    }
    start = clock();
    aplicador(config);
    end = clock();
    imagenes_flipVertical(&config->dst, dst_img);

    imagenes_guardar(config);
    imagenes_liberar(config);
    imprimir_tiempos_ejecucion2(start, end, config->cant_iteraciones, l);
}
```

5.1. Tablas de Rendimiento Promedio y Desvío Estándar

Experimento JMP		
Nombres	***Promedio:***	***Desvio estandar:***
C_O0	41,908	0,492
C_O1	41,931	0,526
C_O2	42,638	0,544
C_Os	42,085	0,493
C_O3	42,709	0,609
ASM	1,434	0,043
ASM_J	1,796	0,047

Color Bordes		
Nombres	***Promedio:***	***Desvio estandar:***
C_O0	24,334	1,411
C_O1	22,585	0,429
C_O2	22,534	0,394
C_O3	22,743	0,479
ASM	1,282	0,039

Todos los datos brindados estan expresados en ms.

Reforzar Brillo		
Nombres	***Promedio:***	***Desvio estandar:***
C_O0	6,011	0,145
C_O1	5,965	0,124
C_O2	6,12	0,144
C_O3	6,129	0,142
ASM	0,448	0,034

Color Bordes		
Nombres	***Promedio:***	***Desvio estandar:***
C_O0	13,975	0,95
C_O1	12,462	0,203
C_O2	12,47	0,202
C_O3	12,468	0,178
ASM	0,735	0,044