Departamento de Computación,
Facultad de Ciencias Exactas y Naturales,
Universidad de Buenos Aires

Trabajo Práctico 2

Organización del Computador II

Segundo Cuatrimestre de 2013

Grupo: **Crema Americana/Persico**

Apellido y Nombre	LU	E-mail
Silvio Vileriño	106/12	svilerino@gmail.com
Esteban Rey	657/10	estebanlucianorey@gmail.com
Matias Chapresto	201/12	matiaschapresto@gmail.com

PC utilizada para las mediciones:

CPU	Sandy Bridge Core i5-2500k @3.30Ghz
GPU	AMD Radeon 7800 HD
Mem.	8Gb Ddr3

Índice

1. Introducción

El objetivo de este trabajo practico explorar y realizar experimentos sobre la tecnologia Streaming SIMD extensions provista por Intel que promete optimizar procesamientos que sean iguales para una cantidad multiple de datos.
Con este objetivo en mente, realizaremos varias implementaciones de 3 filtros, 2 sobre videos y uno sobre imagenes, considerando una implementacion en lenguaje C y otra en lenguaje Assembler utilizando las instrucciones brindadas por el set de instrucciones SIMD. Esperamos obtener mejoras respecto a rendimiento. Realizaremos una serie de experimentos y analisis sobre cada filtro y a continuacion de estos escribiremos conclusiones sobre los datos obtenidos.

2. Desarrollo

2.1. Filtro Color

El filtro color consiste en realizar una iteracion sobre los pixeles de una imagen, estos se dividen en 3 componentes, cada una de ellas identifica un color de la codificacion RGB de la imagen. La iteracion del filtro deja sin modificaciones los pixeles cuya distancia euclideana de sus componentes se encuentren dentro de un rango especificado por el parámetro threshold,los pixeles que no cumplan dicha condicion, se convertiran a blanco y negro por medio de una conversion basada en dejar las 3 componentes del pixel con el promedio de dichas componentes originales.

Como en esta ocasión estamos trabajando con archivos de video, los cuales se pueden definir como una secuencia de imágenes o frames. El filtro aplicará frame por frame la operatoria explicada arriba como si fueran imágenes.

Dado que el objetivo de este trabajo práctico es explorar el modelo de procesamiento SIMD que provee Intel, por este motivo se produjeron 2 implementaciones, una en lenguaje C, y la otra en ASM de 64 bits utilizando instrucciones SIMD.

2.1.1. Descripción del algoritmo

Veamos la implementacion en C y utilizemosla como pseudocodigo para explicar el filtro:
Para cada frame del video de entrada, se realiza el siguiente algoritmo:

```
1  #define OFFSET_BLUE 0
2  #define OFFSET_GREEN 1
3  #define OFFSET_RED 2
4  #define TAMANIO_MAT_BYTES 3*width*height
5
6  void color_filter_c(unsigned char *src,
7                      unsigned char *dst,
8                      unsigned char rc,
9                      unsigned char gc,
10                     unsigned char bc,
11                     int threshold,
12                     int width,
13                     int height)
14 {
15     unsigned char r;
16     unsigned char g;
17     unsigned char b;
18     int diffR;
19     int diffG;
20     int diffB;
21     int distanciaCuad;
22     unsigned short promedio;
23     unsigned char promedioRes;
24     threshold*=threshold;
25
26     for(int index=0;index<TAMANIO_MAT_BYTES;index+=3){
27         r = src[index + OFFSET_RED];
28         g = src[index + OFFSET_GREEN];
29         b = src[index + OFFSET_BLUE];
30
31         diffR=r-rc;
32         diffG=g-gc;
33         diffB=b-bc;
```

```
34
35          diffR *= diffR;
36          diffG *= diffG;
37          diffB *= diffB;
38
39          distanciaCuad = (diffR+diffG+diffB);
40          if(distanciaCuad>threshold){
41              promedio = r+g+b;
42              promedio /= 3;
43              promedioRes = (unsigned char)promedio;
44              dst[index + OFFSET_RED] = promedioRes;
45              dst[index + OFFSET_GREEN] = promedioRes;
46              dst[index + OFFSET_BLUE] = promedioRes;
47          }else{
48              dst[index + OFFSET_RED] = r;
49              dst[index + OFFSET_GREEN] = g;
50              dst[index + OFFSET_BLUE] = b;
51          }
52      }
53  }
```

Explicacion del algoritmo:
Dividamos en secciones el algoritmo en secciones y analicemos en detalle una por una.

1. **Declaracion de variables : Lineas 15-25**
   Declaramos afuera del ciclo las variables que vamos a utilizar para el procesamiento.

2. **Lectura de un pixel como 3 bytes desde la imagen origen : Lineas 26-30**
   Obtenemos de memoria los 3 bytes que corresponden a los colores azul, verde y rojo de un pixel. En la Figura se indica la lectura de este pixel.

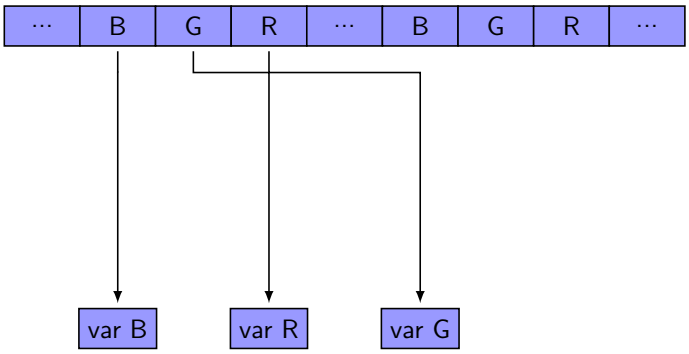


Figura: Lectura de un pixel en 3 variables componentes.

3. **Procesamiento para obtener la distancia euclidiana : Lineas 31-39**
   Realizamos las operaciones aritmeticas necesarias sobre las 3 componentes del pixel obteniendo la distancia euclidiana. Para esta seccion se realizaron algunas conversiones de datos, a continuacion de justificaran las elecciones de variables.

   - **Almacenamiento en double words:** Como la diferencia de dos bytes no signados puede estar en el rango [-255,255] necesitamos un tamaño de datos mas grande para contener este resultado, por ejemplo signed short(word). Se eligio utilizar variables signadas double word, pues luego, cuando se calcula el cuadrado, dicho numero puede tomar como maximo el valor 65535, que no podria representarse en un word signado. La distancia se almacena en double word pues como maximo puede tomar el valor 3*65535.

4. **Comparacion y procesamiento correspondiente: Lineas 40 y 47**
   Se realiza la comparacion entre la distancia calculada y el parametro umbral dado como parámetro, se decide por cual de las siguientes 2 etapas continuar el procesamiento.

5. **Procesamiento Caso 1: Lineas 41-46**
   Esta seccion es ejecutada en caso de que la distancia euclidiana este fuera del rango de filtrado. Puede caracterizarse por la transformacion a blanco y negro de los pixeles que sean procesados

por esta seccion. La conversion se realiza por medio de el calculo del promedio de las componentes del color del pixel, y la posterior escritura de dicho pixel dejando el promedio en cada una de las 3 componentes. Para esta seccion se realizaron algunas conversiones de datos, a continuacion de justificaran las elecciones de variables.

- **Variable auxiliar promedio y reconversion a byte sin signo:** Se expreso la variable promedio como un word sin signo, ya que alcanza para almacenar la suma maxima de 3 bytes no signados.
  Utilizando el siguiente lema, podemos asegurar que el promedio de 3 bytes no signados puede almacenarse como byte no signado, lo que nos permite una conversion segura de word sin signo a byte sin signo.
  Lema: Sea $A = \{a_1, a_2, a_3, ...a_n\}$ un conjunto de bytes sin signo. Luego vale que $promedio(A) \leq max(A)$.

6. **Procesamiento Caso 2: Lineas 48-50**
   Esta seccion es ejecutada en caso de que la distancia euclidiana se encuentre dentro del rango establecido por el umbral. Consiste en escribir en destino el pixel de la misma forma que fue leido, sin modificaciones.

Luego de realizar el algoritmo en C, se le aplicaron una serie de optimizaciones, a saber:

1. **Eliminacion de llamadas innecesarias dentro del ciclo:** Una temprana implementacion del filtro contenia la logica de la distancia tal cual especificada en el enunciado, y se encontraba plasmada en una funcion auxiliar que recibia los parametros y devolvia la distancia. Con el fin de eliminar saltos innecesarios dentro del ciclo, se decidio introducir esta funcion dentro del ciclo.

2. **Eliminacion del calculo de la raiz cuadrada en cada ciclo:** Una temprana implementacion calculaba la distancia y la comparaba con el parametro threshold (o umbral) pasado por parametro. Se decidio elevar al cuadrado ambos miembros de la desigualdad, de forma que con calcular $threshold^2$ fuera del ciclo, y comparar con la distancia al cuadrado, se reducia la cantidad de calculos aritmeticos realizados.

3. **Declaracion de variables auxiliares(de pila) fuera del ciclo** Una temprana temprana implementacion declaraba las variables auxiliares dentro del ciclo, se decidio declararlas fuera, de forma que con una sola declaracion con un scope mas general de las mismas se puedan reutilizar.

2.1.2. Implementacion en assembler utilizando set de instrucciones SIMD

Para realizar esta implementacion se tomo la idea general de la implementacion en C, pero dada la naturaleza de la implementacion de SIMD se eliminaron los saltos condicionales, y se utilizaron mascaras para procesar los casos de aplicacion del filtro sobre los pixeles.
Se procesaran 4 pixeles por ciclo.

Dividamos en secciones el codigo y expliquemos cada una de ellas.

- **Declaracion de mascaras y precomputo de calculos auxiliares**

255	255	255	9	255	255	255	6	255	255	255	3	255	255	255	0

Figura 1: Mascara de reordenamiento de canal azul

255	255	255	10	255	255	255	7	255	255	255	4	255	255	255	1

Figura 2: Mascara de reordenamiento de canal verde

255	255	255	11	255	255	255	8	255	255	255	5	255	255	255	2

Figura 3: Mascara de reordenamiento de canal rojo

255	255	255	255	255	255	12	255	255	8	255	255	4	255	255	0

Figura 4: Mascara de reordenamiento inversa de canal azul

255	255	255	255	255	12	255	255	8	255	255	4	255	255	0	255

Figura 5: Mascara de reordenamiento inversa de canal verde

255	255	255	255	12	255	255	8	255	255	4	255	255	0	255	255

Figura 6: Mascara de reordenamiento inversa de canal rojo

Estas mascaras de definen para los reordenamientos de los pixeles al comienzo y al final del ciclo.

Otros valores precalculados:

- **THREE:** DB 3 Esto se define de esta manera, y luego con instrucciones de mezclado y conversion se replica en un registro XMM como 4 floats empaquetados para realizar el calculo de la division por 3 en el promedio del blanco y negro.

3	3	3	3

Figura 7: Mascara de constante 3 como 4 puntos flotantes de precision simple

- **Generacion de registros XMM con los parametros rc, bc, gc** Se definen 3 registros con 4 enteros de 32 bits empaquetados conteniendo replicados estos parametros, para realizar los calculos aritmeticos.

RC	RC	RC	RC

Figura 8: Mascara de constantes RC como 4 enteros de 4 bytes

BC	BC	BC	BC

Figura 9: Mascara de constantes BC como 4 enteros de 4 bytes

GC	GC	GC	GC

Figura 10: Mascara de constantes GC como 4 enteros de 4 bytes

$threshold^2$	$threshold^2$	$threshold^2$	$threshold^2$

Figura 11: 4 floats empaquetados con el valor de $threshold^2$.

- **Mascara con** $threshold^2$ Se define esta mascara replicando como 4 floats empaquetados el valor de $threshold^2$.

- **Ciclo principal: Lectura de origen y reordenamiento de pixeles**

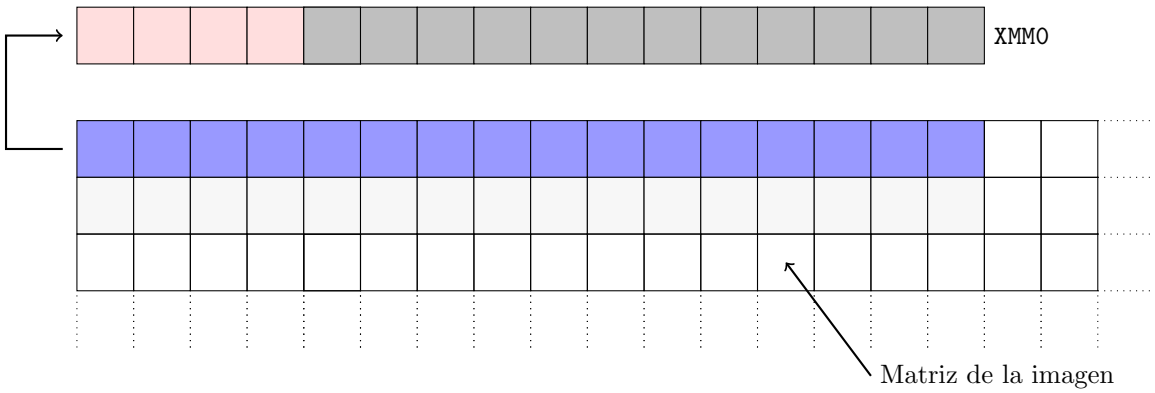


Figura 12: Obtención de los 16 bytes (12 a procesar) en la iteración actual.

Indice de colores de la figura anterior:

- **Azul** Indica la porcion de memoria que se lee.
- **Gris** Indican las componentes que seran procesadas en este ciclo.
- **Rojo** Indican las componentes que seran descartadas en este ciclo y procesadas en el proximo.

Se leen los bytes indicados en la figura y se replican en `XMM0`, `XMM1`, `XMM2`

Luego se procede a reordenar los pixeles como se indica mas abajo.

x	x	x	x	R_3	G_3	B_3	R_2	G_2	B_2	R_1	G_1	B_1	R_0	G_0	B_0

Figura 13: Contenido de los registros `XMM 0,1,2` antes del reordenamiento

`XMM0`	B_3	B_2	B_1	B_0

Figura 14: Contenido del registro **AZUL** luego del reordenamiento con **PSHUFB** con la mascara correspondiente

`XMM1`	G_3	G_2	G_1	G_0

Figura 15: Contenido del registro **VERDE** luego del reordenamiento con **PSHUFB** con la mascara correspondiente

`XMM2`	R_3	R_2	R_1	R_0

Figura 16: Contenido del registro **ROJO** luego del reordenamiento con **PSHUFB** con la mascara correspondiente

Este reordenamiento pretende tener en los registros `XMM0`, `XMM1`, `XMM2`, las componentes de los 4 pixeles a procesar en simultaneo como 4 enteros de 32 bits empaquetados.
Luego de esta etapa, podremos realizar operaciones sobre 4 componentes de forma paralela con una sola instruccion gracias al set de instrucciones SIMD. Resguardo en `XMM3`, `XMM4`, `XMM5` los registros `XMM0`, `XMM1`, `XMM2` luego del reordenamiento, dado que los voy a modificar y necesito tener los originales en etapas mas.

- **Ciclo principal: Calculo de distancia euclidiana**
  Nota inicial: en representacion entera de 32 bits puedo realizar las restas, las elevaciones al cuadrado y las sumas sin problemas.
  En esta seccion, ya tenemos los pixeles organizados de forma tal que podemos realizar los siguientes calculos de forma empaquetada y en este orden:

Operandos de la resta

R_3	R_2	R_1	R_0
RC	RC	RC	RC

Resultado de la resta

(R_3-RC)	(R_2-RC)	(R_1-RC)	(R_0-RC)

Figura 17: Resta empaquetada ilustrativa entre 1 registro de componentes rojas y su parametro rc. Analogamente son las restas empaquetadas entre las componentes verdes y azules y sus parametros.

Operandos del producto

$(R_3\text{-}RC)$	$(R_2\text{-}RC)$	$(R_1\text{-}RC)$	$(R_0\text{-}RC)$
$(R_3\text{-}RC)$	$(R_2\text{-}RC)$	$(R_1\text{-}RC)$	$(R_0\text{-}RC)$

Resultado del producto

$(R_3\text{-}RC)^2$	$(R_2\text{-}RC)^2$	$(R_1\text{-}RC)^2$	$(R_0\text{-}RC)^2$

Figura 18: Producto empaquetado de los registros por si mismos, de esta forma obtenemos las diferencias al cuadrado. Analogamente se realiza para las componentes verdes y azules.

Operandos de la suma

$dr_3 = (R_3\text{-}RC)^2$	$dr_2 = (R_2\text{-}RC)^2$	$dr_1 = (R_1\text{-}RC)^2$	$dr_0 = (R_0\text{-}RC)^2$
$db_3 = (B_3\text{-}BC)^2$	$db_2 = (B_2\text{-}BC)^2$	$db_1 = (B_1\text{-}BC)^2$	$db_0 = (B_0\text{-}BC)^2$
$dg_3 = (G_3\text{-}GC)^2$	$dg_2 = (G_2\text{-}GC)^2$	$dg_1 = (G_1\text{-}GC)^2$	$dg_0 = (G_0\text{-}GC)^2$

Resultado de la suma

$(dr_3 + db_3 + dg_3)$	$(dr_2 + db_2 + dg_2)$	$(dr_1 + db_1 + dg_1)$	$(dr_0 + db_0 + dg_0)$

Figura 19: Suma de las distancias de las 3 componentes de los pixeles. De esta forma obtenemos la distancia euclidiana al cuadrado. Notar que la suma es ilustrativa, pues la suma se realiza en 2 etapas, ver codigo.

- **Ciclo principal: Creacion de mascaras a partir de comparacion con umbral** En esta sección trabajaremos con punto flotante para realizar las comparaciones, ya que las instrucciones de punto flotante me permiten obtener de forma mas directa las mascaras que me indicaran que pixeles y cuales no deberan filtrarse. Realizo una comparacion por menor o igual entre las sumas y un registro conteniendo 4 veces threshold(ya precomputado fuera del ciclo).Asimismo realizo la comparacion inversa(not lower than) para obtener la mascara negada.

  En un etapa anterior habiamos replicado en `XMM3`, `XMM4`, `XMM5` los registros `XMM0`, `XMM1`, `XMM2`, dado que durante el calculo de la distancia y la creacion de mascaras, destruimos los valores en `XMM0`, `XMM1`, `XMM2`, en esta parte restituimos los valores destruidos y notando que las 2 mascaras son excluyentes al ser una la negacion de la otra las aplicamos sobre `XMM0`, `XMM1`, `XMM2` y `XMM3`, `XMM4`, `XMM5` respectivamente. De esta forma tenemos en cada tripla de registros (`XMM0`,`XMM1`,`XMM2`) y (`XMM3`,`XMM4`,`XMM5`) los 4 pixeles separados por componente y en en la primera tripla estan los que serán procesados a blanco y negro, mientras que en la segunda, se encuentran los que quedarán sin modificaciones.

Comparación lower than threshold

$(dr_3 + db_3 + dg_3)$	$(dr_2 + db_2 + dg_2)$	$(dr_1 + db_1 + dg_1)$	$(dr_0 + db_0 + dg_0)$
$threshold^2$	$threshold^2$	$threshold^2$	$threshold^2$

Máscara resultado

0xFFFFFFFF/00000000	0xFFFFFFFF/00000000	0xFFFFFFFF/00000000	0xFFFFFFFF/00000000

Figura 20: Ejemplo de comparación lower than. Obtenemos una máscara con 0xFFFFFFFF ó 0x00000000 depediendo de si en esa posición, la suma de los cuadrados de las diferencias era menor o mayor o igual a threshold.

Aplicación de la máscara lower than en el registro ROJO

R_3	R_2	R_1	R_0

(Máscara ejemplo)

0xFFFFFFFF	0x00000000	0xFFFFFFFF	0x00000000

Resultado:

R_3	0x00000000	R_1	0x00000000

Figura 21: Nos quedan los valores rojos de los pixeles que no tienen que cambiar, y 0s donde habia un valor rojo de un pixel que tiene que ser convertido a blanco y negro

- **Ciclo principal: Calculo de blanco y negro** En esta seccion debemos realizar el calculo de promedio empaquetado entre los 3 registros de la tripla (`XMM0`,`XMM1`,`XMM2`). Para ello realizamos sumas empaquetadas de enteros de 32 bits entre `XMM0` y `XMM1` y luego entre `XMM0` y `XMM2`, obteniendo las sumas en `XMM0`. La division por 3 es realizada en punto flotante contra un registro precomputado que contiene la constante 3 replicada como 4 floats.
  Nota: Se realiza una conversion de entero de 32 bits a punto flotante simple antes de la division y una conversion inversa luego de esta.

Promedio de los componentes de los 4 pixeles

R_3	R_2	R_1	R_0
B_3	B_2	B_1	B_0
G_3	G_2	G_1	G_0

Resultado de la suma

$R_3 + B_3 + G_3$	$R_2 + B_2 + G_2$	$R_1 + B_1 + G_1$	$R_0 + B_0 + G_0$

Conversión a float, división, y re-conversión a int

$\frac{R_3+B_3+G_3}{3}$	$\frac{R_2+B_2+G_2}{3}$	$\frac{R_1+B_1+G_1}{3}$	$\frac{R_0+B_0+G_0}{3}$

Figura 22: Obtenemos el promedio de las 3 componentes de cada pixel. Notar que este ejemplo es un caso en el cual todos los pixeles deben ser modificados

- **Ciclo principal: combinacion de casos, reordenamiento de pixeles y escritura a destino**
  Esta seccion final, se encarga de combinar los 2 casos que habian quedado separados luego de aplicar las mascaras, y finalmente reordena las componentes de forma tal que queden en formato apropiado para ser escritas a destino. Notemos algunas cosas:
  - **Los dos casos que se separan en la aplicacion de mascaras son disjuntos**
    Por lo tanto con operaciones logicas de disyuncion, se pueden unificar.
  - **Cuando se pasa un pixel a blanco y negro, las 3 componentes contienen el promedio**
    Por lo tanto la unificacion entre los pixeles originales y el blanco y negro puede realizarse utilizando una sola componente del blanco y negro varias veces.

Ejemplo de una unificación por medio de una operacion de disyuncion

Registro de promedios

$\frac{R_3+B_3+G_3}{3}$	0x00000000	$\frac{R_1+B_1+G_1}{3}$	0x00000000

Registro ROJO

0x00000000	R_2	0x00000000	R_0

Registro AZUL

0x00000000	B_2	0x00000000	B_0

Registro VERDE

0x00000000	G_2	0x00000000	G_0

Luego del Packed OR:

Registro ROJO

$\frac{R_3+B_3+G_3}{3}$	R_2	$\frac{R_3+B_3+G_3}{3}$	R_0

Registro AZUL

$\frac{R_3+B_3+G_3}{3}$	B_2	$\frac{R_3+B_3+G_3}{3}$	B_0

Registro VERDE

$\frac{R_3+B_3+G_3}{3}$	G_2	$\frac{R_3+B_3+G_3}{3}$	G_0

Figura 23: Ya tenemos agrupadas y procesadas las componentes de los 4 pixeles. Lo único que queda ahora es hacer un reordenamiento de pixeles como el del principio, pero de manera inversa y finalmente escribir a destino el resultado

2.1.3. Optimizaciones y tecnicas especificas aplicadas a la implementacion con SIMD

Se aplicaron las siguientes optimizaciones al codigo con el objetivo de mejorar la performance del filtro:

- **Tecnica de desenrrollado de ciclo:** Esta tecnica se realizo mediante la aplicacion de una macro sobre el cuerpo del ciclo, y una adaptacion del ciclo original a uno que incluye 16 llamados a la macro, indicando el correspondiente desplazamiento relativo de la porcion de memoria a procesar, y la modificacion del incremento a los indices del ciclo correspondientes. De esta forma, se espera obtener una mejora en la performance dado que esta tecnica en teoria reduce 16 veces la cantidad de comparaciones que se realizan en la guarda del ciclo.
- **Optimizaciones heredadas de la implementacion C:** De la implementacion original en C,

tomamos la idea de eliminar el calculo de la raiz cuadrada de la distancia euclidiana.

- **Minimizacion de conversiones y operaciones con punto flotante:** Una temprana implementacion del filtro en assembler realizaba el 100 % de los calculos aritmeticos sobre 4 numeros empaquetados de punto flotante simple, luego nos dimos cuenta que esto ralentizaba mucho los calculos y la mayoria de ellos podian realizarse sobre enteros, dejando unicamente el calculo de mascaras por conveniencia de las instrucciones disponibles para el problema en el que estabamos trabajando, asi tambien la division empaquetada del promedio como un calculo necesario sobre punto flotante, se intento mejorar esto investigando una aproximacion de la division basada en multiplicar por una fraccion de la forma $\frac{j}{2^k}$ donde j,k son numeros enteros, de esta forma, podria realizarse una division con operaciones de producto por j y un shift a derecha de k lugares(division por 2^k), pero tanto los errores de precision de la aproximacion como las complicaciones de modificacion del codigo y de operatoria paralela empaquetada que esto implicaba, hicieron que finalmente fuese descartada esta modificacion.
- **Reordenamientos y accesos a memoria:** Utilizamos reordenamientos de registros por medio de las instrucciones de **shuffle** para procesar los canales de los pixeles de la mejor forma que se nos ocurrio. Para la aplicacion de estas mascaras, utilizamos la mayor de cantidad de registros disponibles para evitar los accesos a memoria para datos estaticos, es decir, que no se modifican a medida que pasan los ciclos. Asi tambien, notando que 2 de las 3 las mascaras de reordenamiento inverso se pueden obtener realizando operaciones de corrimiento de bits de otra, se barajó la posibilidad de crear en cada ciclo dichas mascaras, evitando 2 de 3 accesos a memoria, sorprendentemente, los resultados numericos fueron peores realizando esto que manteniendo los 3 accesos, creemos que se debe a la memoria cache, por lo tanto descartamos esta ultima posibilidad.

2.1.4. Analisis y resultados

Luego de realizar la implementacion en assembler utilizando SIMD, comenzamos a realizar un analisis sobre las diferencias estructurales del codigo en ASM contra la implementacion en C, aprovechamos las opciones de compilacion con diferentes niveles de optimizacion que nos brinda el compilador **GCC** y para cada una de esas optimizaciones automaticas, realizamos una revision del codigo assembler generado con la herramienta **objdump**, lo cual nos permitio observar cuales son las optimizaciones que realiza cada nivel, cuales cosas mejoran, cuales se podrian continuar optimizando y de ser posible, como.

Ademas, se realizo un experimento para poder ver el impacto de los saltos condicionales dentro de un ciclo, dicho analisis se realizo sobre la version del filtro en C, compilado con el primer nivel de optimizacion.

2.1.5. Tiempos de ejecucion de diferentes experimentos

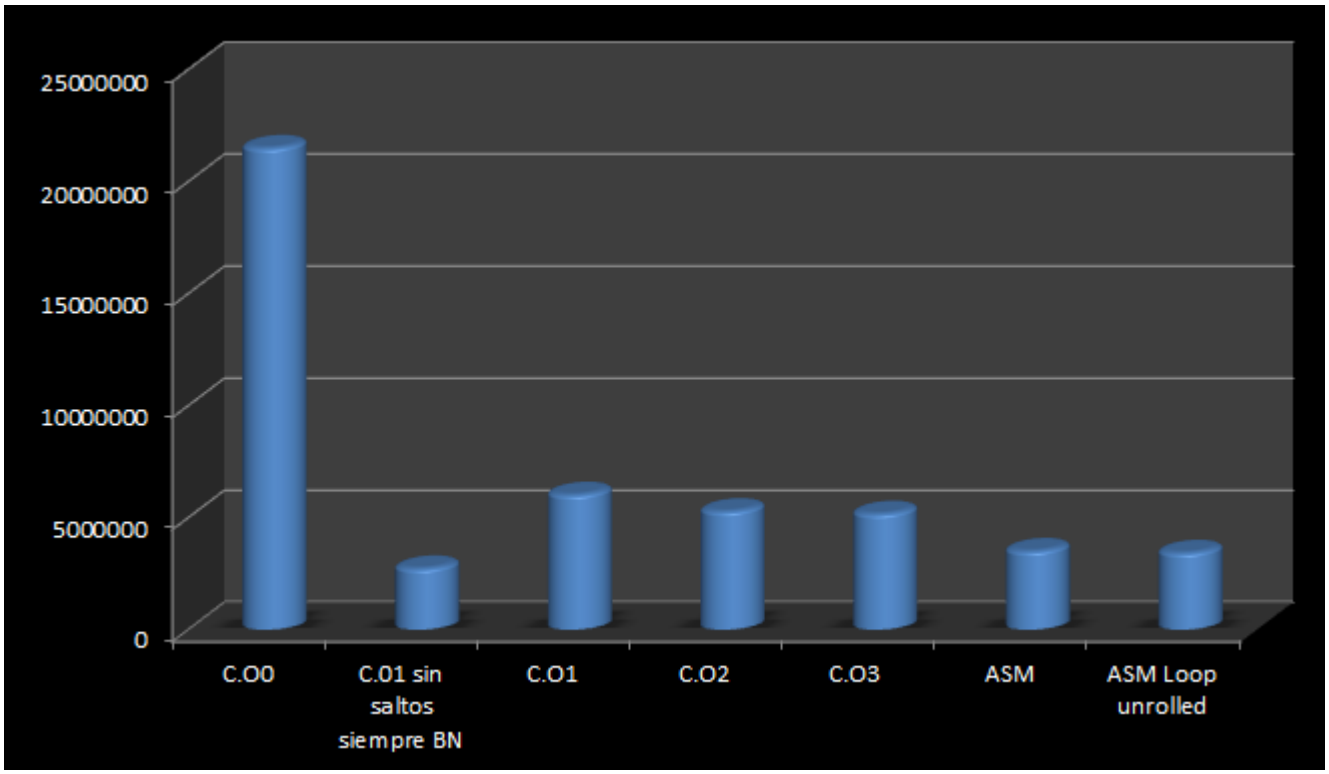
Notas iniciales: Las mediciones se realizaron tomando dos promedios, el primero, es un promedio de los tiempos que toma cada llamada a la funcion particular del filtro a los k frames del video de entrada, esto nos dice cuantos ciclos consume en promedio la aplicacion del filtro a un frame. Luego ademas, se hace un promedio de la cantidad de veces o iteraciones que se aplica el filtro al mismo video. Creemos que es una buena medicion, ya que se diluyen los valores atipicos producto que produce el scheduler del sistema operativo al cambiar de contexto. Asi tambien pensamos que realizando la medicion de esta manera, sin tomar en cuenta el procesamiento de la libreria *OpenCV* para el procesamiento del archivo de video, es mas limpia y se centra mas en el analisis del codigo implementado.

Caracterizacion del experimento:

- **Iteraciones:** 10
- **Comando:** ./tp2 -t 10 -i < *implementacion* > fcolor ink.avi 100 100 100 100
- **Nota:** Los experimentos de optimizaciones del compilador no suponen ninguna alteracion sobre el comando de testeo ya que se realizan sobre la compilacion de los fuentes C del filtro.
- **Nota 2:** El experimento sobre C.01 eliminando los saltos condicionales se realizo, dejando siempre por defecto la conversion a blanco y negro de los pixeles.

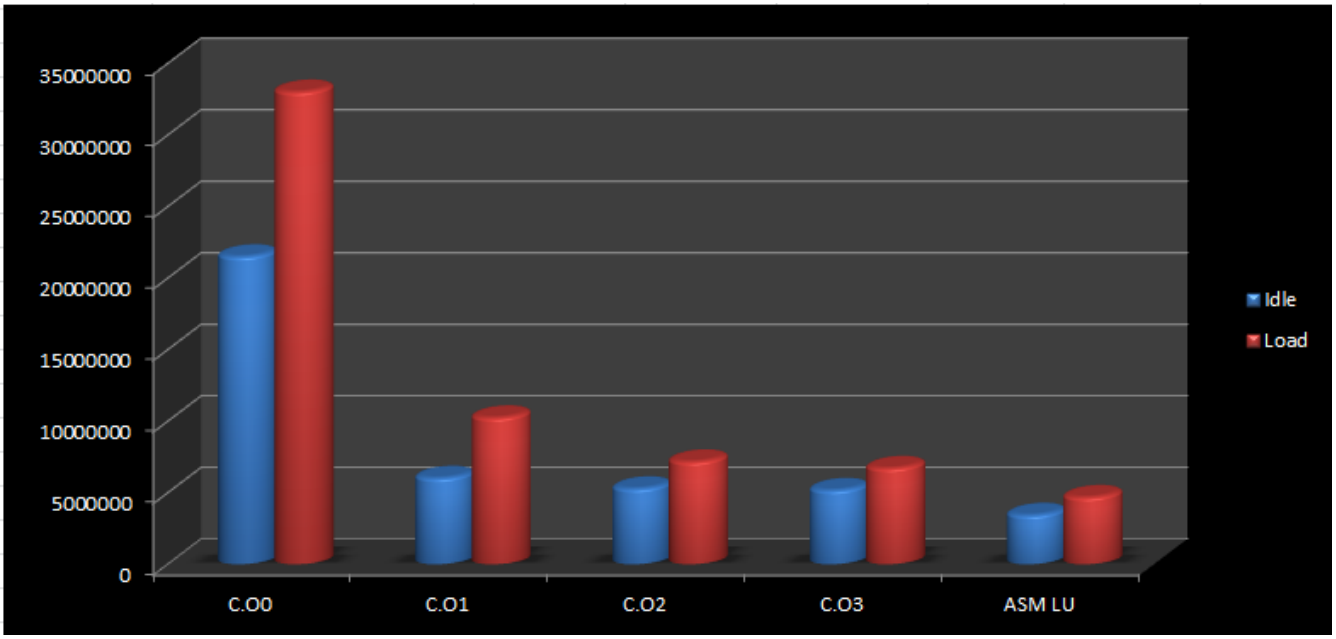
Resultados:

Medición	C.O0	C.O1 ss	C.O1	C.O2	C.03	ASM	ASM LU
Ciclos por llamada	21499106	2729604	6018971	5282602	5169947	3507714	3426934



Una prueba adicional se hizo mientras se corria paralelamente un programa que utilizaba el 100 % del cpu arrojo resultados proporcionales. A continuacion de presentan en un grafico las relaciones entre las mediciones en estado *idle* y en estado 100 % load.

Medición	C.O0	C.O1	C.O2	C.O3	ASM LU
Load	33009164	10239740	7165908	6709664	4677277
Idle	21499106	6018971	5282602	5169947	3426934
Relacion de incremento	53 %	70 %	35 %	30 %	30 %



2.1.6. Diferencias estructurales de las diferentes versiones analizadas

En esta seccion nos centraremos en analizar las diferencias entre la version SIMD y las versiones C con sus diferentes niveles de optimizacion, de esta forma esperamos poder explicar los resultados numericos de la seccion anterior.

Consideraciones iniciales:

Notoriamente entre la version del algoritmo en C y la version del algoritmo utilizando SIMD lo primero que hay que ver es que en la version de C se lee y procesa un pixel de origen por ciclo, mientas que utilizando SIMD, se procesan paralelamente 4 pixeles en instrucciones atomicas. Asi tambien cambia la forma de pensar el algoritmo como una division en casos con saltos condicionales entre ramificaciones de la ejecucion, hacia un pensamiento mas enfocado en procesamiento paralelo de valores, en los cuales mediante mascaras decidimos cuales componentes se procesaran con cada caso del filtro y luego, al ser casos disjuntos, pueden ser mezclados y escritos a destino, luego de realizar las conversiones y/o mezclados necesarios.
Aclaracion sobre multiplicidad: No tuvimos en cuenta los casos en donde la entrada tuviera un tamaño en pixeles que no fuera un multiplo de 4. Asumimos esta precondicion sobre la entrada para simplificar el algoritmo, podria solucionarse de forma sencilla sobre el codigo realizado, realizando un calculo de resto modulo 12 y una resta al puntero indice fuera del ciclo. Para informacion mas detallada de como solucionar este problema, dirigirse a la seccion del filtro **decode** donde se explica con mas detalle.

Desensamblado de la implementacion original en C

Tomemos el codigo desensamblado con la herramienta *objdump* y busquemos posibles optimizaciones. Podemos observar en el desensamblado del C sin optimizaciones, que las variables locales se utilizan como espacio de pila, produciendo un acceso a memoria cuando son requeridas para leer o escribir.Tambien afecta el traslado adicional entre memoria y registros para realizar operaciones.

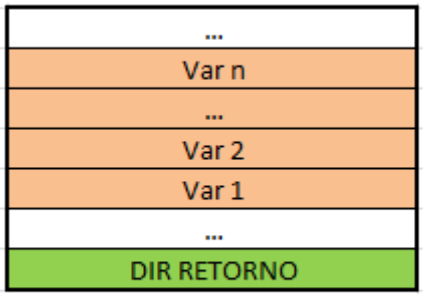


Figura: Stack con variables auxiliares

Se encontro tambien la optimizacion en la division por 3 con el metodo de fraccion con denominador potencia de 2 mencionado anteriormente.
El resto del codigo parece acorde a la implementacion en C, de hecho es asi para permitir una correcta depuracion segun la documentacion de *GCC*.

Obviamente, el hecho de acceder a memoria para las variables locales, mas alla de lo que mejore la memoria cache, puede ser optimizado utilizando los registros internos del CPU y evitar estos accesos.

Incrementando la intensidad de la optimizacion: Primer nivel de optimizacion O1

En este modo se puede observar claramente el aprovechamiento de los registros internos del cpu para almacenar valores y variables temporales y evitar los accesos a memoria innecesarios. Asi tambien se observa que cambio el modo de direccionar la matriz ahora de una forma mas directa y con menor cantidad de instrucciones.
Es notorio el cambio en el tiempo de ejecucion este cambio, basta ver los graficos para ver que la reduccion considerable de accesos a memoria produjo una mejora importante.

Analizando las optimizaciones mas agresivas: O2 y O3

GCC nos ofrece una amplia variedad de flags para optimizar, se puede especificar en particular cada una de las posibles optimizaciones a realizar al codigo para obtener un ajuste fino. Por motivos de tiempo, solo analizaremos los flags que permiten optimizar de una forma automatica con cierto grado de agresividad, cada uno de estos flags engloba un grupo de optimizaciones particulares al codigo.

Un breve listado de las posibles optimizaciones estandares se muestra debajo:

Intensidad	Descripcion
O0	Casi ninguna transformacion al codigo, solo pasaje a assembler.
O1	Optimizaciones que no consuman mucho tiempo de compilacion.
O2	Optimizaciones que modifican el orden de ejecucion mejorando la velocidad del codigo resultante. Pueden verse alteradas variables del usuario y el cuerpo de algunas funciones.
O3	Optimizaciones que modifican aun mas el orden de ejecucion y pueden o no mejorar la velocidad del codigo resultante. Puede verse alterada la semantica de las operaciones (particularmente las de punto flotante).

Existen otros parametros, entre ellos a saber $-Ofast$ que provee una mayor optimizacion que $-O3$ pero con el costo de no respetar los estandares. Por ejemplo, se implementa la optimizacion $ffast-math$ que acorde a la documentacion de *GCC* puede producir errores en donde se espere una implementacion exacta del estandar IEEE o ISO de las funciones matematicas, es decir, puede verse alterada la precision de ciertos calculos.

Otro modo de optimizacion es $-Os$ que optimiza el tamaño del codigo manteniendo las optimizaciones de $-O2$ que no interfieran con el objetivo de este modo.

Nota: Para mas informacion dirigirse a:

`http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`
`http://www.redhat.com/magazine/011sep05/features/gcc/`

2.1.7. Conclusiones acerca de las mediciones

Luego de realizar estos analisis, podemos concluir acerca de las optimizaciones brindadas por el compilador *GCC* que mejoran sorprendentemente la performance del mismo algoritmo, el mayor salto en rendimiento se ve con el primer nivel de optimizacion $O1$ que brinda el compilador. Asi mismo, la implementacion en assembler con SIMD obtiene mejores resultados que el C optimizado en su maximo nivel. El desenrrollado de ciclos brindo alguna mejora adicional, pero no demasiado significativa, y finalmente con respecto a los saltos condicionales, es muy notorio, como el procesador con su prediccion de saltos puede acelerar muchisimo la ejecucion, menos de la mitad de ciclos, ejecutando siempre la rama de ejecucion con mas costo, en este caso pasando siempre a blanco y negro realizando calculos aritmeticos de punto flotante, creemos que esto se debe a que la cpu no puede predecir cosas sobre los datos, condicion por la cual se evalua la guarda del salto condicional eliminado para el experimento.

Con respecto a los experimentos realizados con otra aplicacion utilizando el 100 % del cpu, se debieron correr 2 aplicaciones paralelamente que consuman el total del cpu, pues con una sola, creemos que al ser multicore, trasladaba el procesamiento a un solo core, dejando equilibrado el sistema operativo sobre los cores restantes. Los resultados fueron proporcionales, con excepcion de la version sin optimizaciones de C, atribuimos este valor atipico, a la cantidad de accesos a memoria que realiza esta version con respecto a las demas testeadas.
Como objetivo de este tp, logramos comprobar la efectividad de las instrucciones SIMD a la hora de realizar calculos identicos a varios datos que pueden paralelizarse.

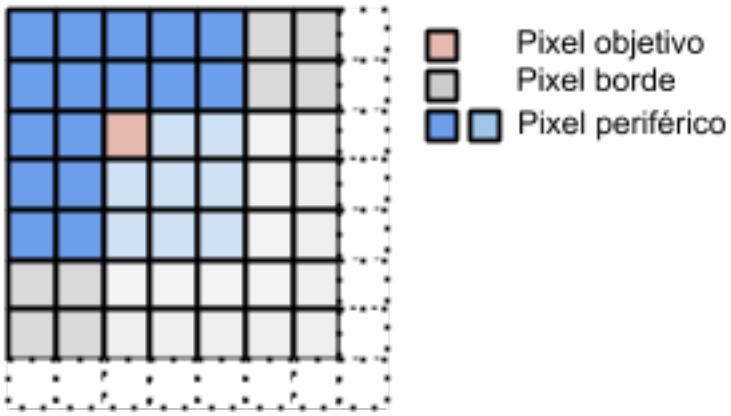
2.1.8. Aclaraciones especiales respecto a los tests del fcolor

Hay un test particular **test-fcolor-city.tar.gz** que tiene desactualizado el threshold en la pagina de la materia. Remitirse al email enviado por Marco Vanotti en la fecha 20-09-2013 en el cual explica como alterar el threshold del comando, cambiando su valor de 5000 a 71. Luego de este cambio, nuestro algoritmo pasa dicho test.

2.2. Miniatura

El filtro miniatura busca crear un efecto de miniaturizacion de una seccion de una imagen. Esto se logra mediante la aplicacion de un filtro de blureado de pixeles a una banda superior e inferior de la imagen, creando de esta forma en la banda central un efecto de miniatura.

El filtro de blureado se basa en la aplicacion de una matriz de coeficientes a los pixeles circundantes al pixel objetivo y el cambio de los valores de color del mismo por el promedio de los resultados de las multiplicaciones, siendo los coeficientes representantes a una campana de gauss, se logra el disfuminado del pixel objetivo. Para maximizar el efecto, se pasa repetidas veces el filtro sobre la imageen, reduciendo la caantidad de filas por banda, creando asi un blureado en degrade haia el centro de la imagen.
Para la implementacion del filtro para el trabajo se utiliza una matriz de dimension 5, con lo cual, por la simetria de la campana, se poseen 25 valores de los cuales solo 6 son distintos entre si. Al aplicarse esta matriz se debe considerar que, al posicionarse el pixel objetivo en el centro de la misma, se debe contar con pixeles en el radio de la matriz, para poder aplicarse los coeficientes. Debido a esto, se toman 2 pixeeles de marco para la imagen en los bordes, los cuales quedaran iguales.



Los coeficientes de la matriz, siendo valores comprendidos entre 0 y 1, requeririan la utilizacion de valores de punto flotante. Al ser las operaciones con los mismos mas caras en tiempo, se pasan los numeros a valores enteros, multiplicando los flotantes por 100. La presicion del resultado se conserva porque los coeficientes son de presicion centesimal.
Para no aumentar el brillo de la imagen se divide por la suma de todos los coeficientes de la matriz (6) .A esto se le agrega la division por 100 para generar los valores correctos.

2.2.1. Descripcion del algoritmo

Para la descripcion del algoritmo contamos con el pseudocodigo acotado de la solucion implementada, el cual servira de guia para la interpretacion de las tareas realizadas. La logica del mismo fue pensada para tamaños de imagenes a partir de 5 x 5 pixeles en el caso de C y 5 x 7 en el caso de assembler. Para explicar el algoritmo se describira en primera instancia la idea y consideraciones generales que comparten ambas versiones; para luego profundizar en cada una, detallando en la seccion de assembler, los cambios realizados para la adaptacion a las instrucciones SIMD.

```
1
2 void miniature_c(
3                 unsigned char *src,
4                 unsigned char *dst,
5                 int width,
6                 int height,
7                 float topPlane,
```

```
8                   float bottomPlane,
9                   int iters)
10 {
11
12      int matriz[5] = {1,5,18, 32, 64};
13
14      for(iteracion = 0 to iters-1){
15
16          for (fila = 0 to height-1){
17
18              for(columna = 0 to width - 1){
19
20                  indice_pixel = calcularPosicionReal(src, fila, columna);
21
22                  bool condicion1 = no_esBorde(src, indice_pixel);
23                  bool condicion2 = esta_dentroFrames();
24
25                  if( condicion1 && condicion2 ){
26
27                      aplicarFiltro(src, dst, matriz, indice_pixel);
28
29                  }else{
30
31                      copiarIgual(src, dst, indice_pixel)
32
33                  }//if
34              }// columnas for
35          }// filas for
36
37          intercambiar(src, dst);
38
39      }// iteraciones for
40
41 }// funcion
```

1. **La matriz de coeficientes**
   La matriz, como fue dicho en la introduccion, al ser simetrica posee solo 6 valores distintos entre si: 0.01, 0.5, 0.18, 0.32, 0.64 y 1. Estos valores son guardados en forma entera multiplicandolos por 100, sin perder presicion.

2. **Recorrido de la imagen**
   En una primera implementacion se recorria la matriz de pixeles en forma lineal (tomando la matriz como una lista). No obstante se vio que la lectura y desarrollo del codigo se complicaba a la hora de calcular limites y casos bordes, al contrario de un recorrido por filas y columnas.

3. **Posicion del pixel en la matriz**
   Al hacerse un recorrido por filas y columnas es necesario un calculo adicional para traducir la posicion: por lo que la posicion se calcula haciendo columna * tamPixel + fila * width * tamPixel, siendo tamPixel = 3 (imagen en RGB).

4. **Limites de bandas e iteraciones**
   Los limites de las bandas por iteracion fueron dados por consigna mediante la aplicacion de una ecuacion que presenta una division. No estaba especificado lo que se debia hacer en caso de que la misma no fuese exacta (con resto). Se pudo determinar en la etapa de testeo, que en el caso de la bada superior, se debia redondear para arriba. Este cambio se realizo al notar que solo el ultimo test: test-miniature-03-08-5-city; aparecian en el primer frame procesado, en la banda superior, una diferencia en las imagenes por 2 pixeles. Al aumentar la cantidad de iteraciones del test y bajar la tolerancia en la comparacion, se pudo determinar que en las bandas de las iteraciones de division no exacta, se perdia una linea de pixeles. El cambio permitio pasar 29 frames del test final. Otros pxeles fueron marcados como diferentes en la esquina inferior del frame 30. Se intento la aplicacion de la misma solucion que para el caso anterior pero sin exito. Al ser los ultimos pixeles filtrados de la imagen (5 pixeles diferentes), y ser objetivo de varias iteraciones, se supuso algun tipo de error numerico entre nuestra version y la version que produjo las imagenes del test. Se descarto la posibilidad de perdida de presicion ya que en las operaciones para adaptar los coeficientes a valores enteros, no se recorto ninguna cifra y en las operaciones se cuido de no producir redondeos de mas.

5. **Bordes de la imagen**
   El algoritmo solo se puede aplicara a los pixeles cuya posicion diste en 2 de los bordes de la imagen. No obstante, en la etapa de testeo se determino que, para pasar exitosamente los tests, se debia considerar los pixeles a partir de la fila 4

6. **Intercambio**
   Para aumentar el efecto del blureado se pasa la imagen blureada a la proxima iteracion como imagen fuente.

Hasta este punto el algoritmo implementado en C y en Assembler son iguales. La diferencia entre ellos radica en la implementacion de **aplicarFiltro** y **copiarIgual**

2.2.2. Implementacion en C

Tanto en la aplicacion del filtro como en la copia de los pixeles se hace un procesamiento por pixel, es decir, se lee un pixel se copia un pixel.Como para cada pixel a filtrar son necesarios 2 filas de pixeles circundantes, la imagen mas chica a evaluar debera tener por lo menos una dimension de 5 x 5 pixeles.

```
void aplicarFiltro(int matriz[],
                   int indice_pixel,
                   unsigned char *src,
                   unsigned char *dst,
                   int width){

    int r=0;
    int g=0;
    int b=0;



    int fila = 0;
    int columna = 0;

    int offset_area_blur = indice_pixel - 2 * PIXEL - 2 * width * PIXEL;


    for (fila = 0; fila < LADO_BLUR; fila++){

        for (columna = 0; columna <LADO_BLUR; columna++){

            int indice_pixel_contorno =  offset_area_blur +  (columna * PIXEL + fila * width * PIXEL);

            int r_aux = (int) src[indice_pixel_contorno+OFFSET_RED];
            int g_aux = (int) src[indice_pixel_contorno+OFFSET_GREEN];
            int b_aux = (int) src[indice_pixel_contorno+OFFSET_BLUE];
            int coef = 0;


            //Aplicacion de coeficientes de la matriz
            if (fila == 0 || fila == 4){

                if(columna == 0 || columna == 4){

                    coef =  matriz[0];//0.01

                }else if (columna == 1 || columna == 3){

                    coef =  matriz[1];//0.05

                }else{

                    coef = matriz[2];//0.18
                }

            }else if (fila == 1 || fila == 3){

                if(columna == 0 || columna == 4){

                    coef =  matriz[1];//0.05

                }else if (columna == 1 || columna == 3){

                    coef =  matriz[3];//0.32

                }else{
```

```
59
60                      coef = matriz[4];//0.64
61                  }
62              }else{
63
64                  if(columna == 0 || columna == 4){
65
66                      coef =  matriz[2];//0.18
67
68                  }else if (columna == 1 || columna == 3){
69
70                      coef =  matriz[4];//0.64
71
72                  }else{
73
74                      coef = 100; //pixel objetivo;
75                  }
76
77              }//fin coeficientes matriz
78
79              r += r_aux * coef;
80              g += g_aux * coef;
81              b += b_aux * coef;
82          }
83      }
84
85      //---------------normalizo valores-----------------//
86
87      r = (r/100)/6;
88      g = (g/100)/6;
89      b = (b/100)/6;
90      //---------------guaardo en destino---------------//
91      dst[indice_pixel+OFFSET_BLUE] = (unsigned char) b;
92      dst[indice_pixel+OFFSET_GREEN] = (unsigned char) g;
93      dst[indice_pixel+OFFSET_RED] = (unsigned char) r;
94
95
96  }
```

El filtro obtiene un cuadrado de pixeles de 5 x 5 tomando como centro el pixel a blurear y a cada pixel, dependiendo su posicion se lo multiplica con un coeficiente de la matriz. El resultado se guarda en una variable temporal por cada color del pixel. Al finalizar de multiplicar a todos los pixeles del cuadrado, normalizamos por 600 (6 del brillo y 100 por la conversion a enteros).

Este procedimiento se realiza 25 veces, que es la cantidad de pixeles dentro del cuadrado. Finalmente se escribe el pixel objetivo en el archivo de destino.

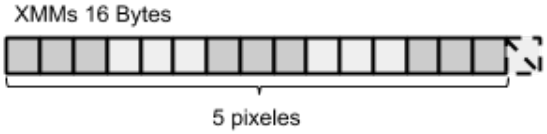
2.2.3. Implementacion en assembler utilizando set de instrucciones SIMD

Al disponer de instrucciones que nos permiten paralelizar calculos, se busco la forma de juntar el procesamiento de las partes del algoritmo en donde se analizaba de a un dato. Este aspecto es fundamental, ya que nos permite disminuir los accesos a memoria, principal cuello de botella de C. La idea implementada con SIMD se dividio en 2 aspectos del algoritmo: la copia de pixeles iguales, como los bordes y el frame central, y el filtrado de aquellos pixeles de las bandas superior e inferior. Incluido en la optimizacion de accesos a memoria, se busco, aparte del copiado de mas de un pixel a la vez, optimizar los calculos con la matriz, dado que representa el grueso de peticiones a memoria de nuestra solucion.

Al estar trabajando con mas de un dato, surgieron casos bordes y detalles que en C, no habia. En los siguientes puntos se muestran los distintos aspectos que fueron adaptados para el uso de las SIMD.

Copia de pixeles:

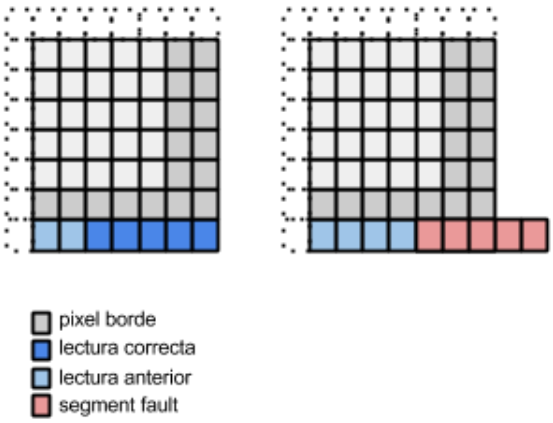
Para la copia de pixeles iguales se aprovecho el hecho de que los pixeles levantados son consecutivos, por lo que se pudo copiar de a 5 pixeles enteros por acceso a memoria. Esto se debe al uso de los registros de 16 bytes, (descartandose el ultimo byte).



Caso borde:

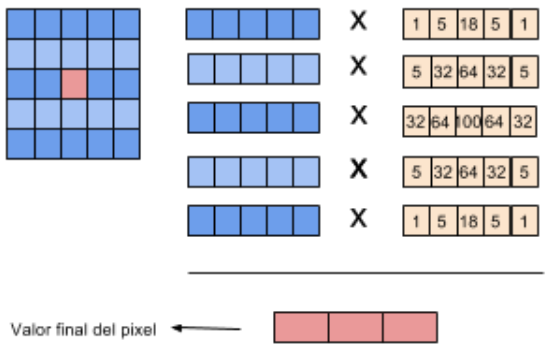
Bajo este modo de copia surge un caso borde al final del archivo de imagen. Como se esta levantando de a 16 bytes, el archivo puede no ser divisible por dicho numero (en la mayoria de los casos), con lo cual

se debe detectar la proximidad al final del archivo y correr el puntero de la memoria a levantar 16 bytes antes del borde.



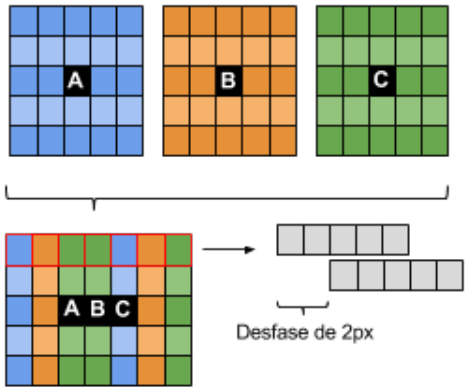
Filtro:
Para la implementacion del filtro en Assembler se realizaron 2 etapas: Paralelismo en cuanto a calculo con la matriz de coeficentes y paralelismo en cuanto a pixeles copiados.

La primera etapa busco reducir el numero de accesos de memoria para el calculo del producto de los coeficientes de la matriz con los pixeles circundantes al objetivo. Se aprovecho el hecho de que la longitud de las filas de la matriz y la cantidad de pixeles levantados por XMMs coincidian, con lo cual se procedio a calcular el resultado parcial de los productos y sumas de cada fila. Luego se normalizaba con 600 y se pasaba el valor del pixel a memoria:



De esta forma estariamos levantando de memoria 5 veces por cada pixel evaluado, con lo que reduce los 25 accesos por pixel de la version en C.

La segunda tapa fue la de lograr incorporar mayor cantidad de pixeles al resultado. Por razones de cantidad de registros se logro procesar un maximo de 3 pixeles a la vez. Esto se alcanza mediante el solapamiento de cuadrados de 5 x 5 de los 3 pixeles:

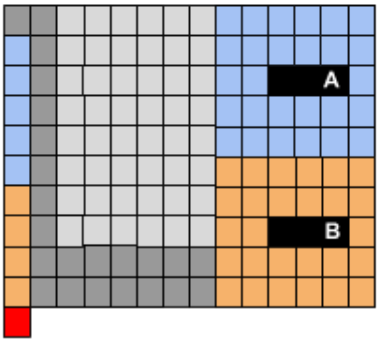


Filtro miniature: Imagen de solapamiento de datos y aprovecahmiento de xmms

De esta forma se estarian efectuando 10 accesos a memoria por cada 3 pixeles evaluados. En cuanto el archivo de entrada debera contar ahora con 2 columnas mas, aumentando a 5 x 7 la menor imagen filtrable por el algoritmo

Caso borde:

Al igual que en la copia de pixeles, el filtrado posee un caso borde en la ultima fila que puede ser filtrada: la antepenultima. Como para el procesamiento del pixel son necesarias las dos filas inferiores al pixel y las dos columnas del borde, si no se detecta a tiempo se produce un error de acceso a segmento. Esto se evita añadiendo logica de borde la cual, de estar analizando alguna de los ultimos 3 pixeles filtrables, se tome el puntero de memoria a levantar de forma que en el calculo de levantado de las filas para los coeficientes, no nos genere un error de segmento:



En la imagen anterior, se pueden ver dos situaciones. El cuadro de pixeles donde se encuentra el pixel A, llego a un borde y lee datos basura del otro borde de la matriz. Sin embargo esto no presenta ningun problema, ya que, los pixeles leidos existen y el dato erroneo del pixel A que se guarda en la nueva imagen, sera pisado cuando el algoritmo detecte en una iteracion siguiente de que pertenece al borde (explicado en seccion siguiente), por lo que copiara nuevamente a la posicion de destino el valor correcto.

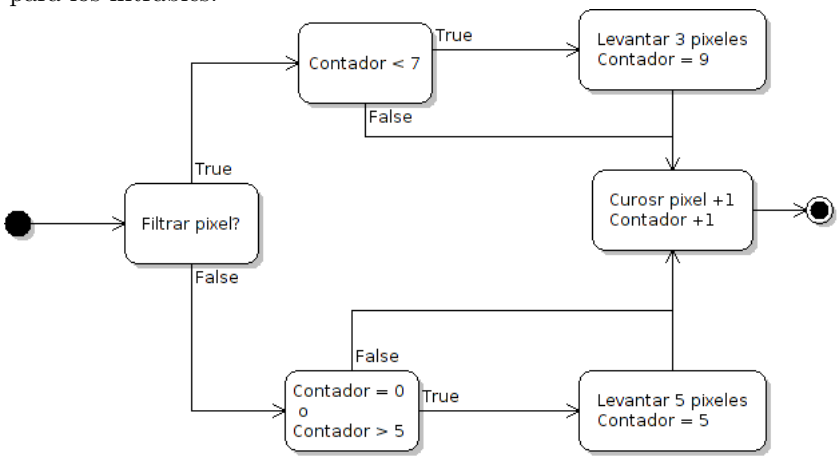
En el caso B se puede ver que se ha llegado al final de la imagen, por lo que, al contrario del caso A, si leo un pixel del borde, no va a ver una fila inferior para leer, con lo que nos iriamos del segmento. Para contrarrestar esto, la logica de borde analiza que de estar en la antepenultima fila filtrando y si estamos en el antepenultimo pixel que se debe filtrar, levante y procese los 3 pixeles, se saltee directamente los 2 proximos pixeles (ultimo y anteultimo, porque de no saltearlos en la proxima iteracion volveriamos al caso borde creanso un loop infinito) y adelante el cursor de columna directamente a los pixeles del borde.

Solapamiento entre filtrado y copiado:

Al estar analizando de mas de un pixel a la vez, se da la situacion en que: por ejemplo, al levantar 5 pixeles para copiar y dejar igual, estemos levantando uno o varios que deben ser filtrados, con lo cual de pasar a analizar los proximos pixeles, estariamos produciendo una imagen con error.

Para evitar esto estamos obligados a analizar indice de pixel a indice, determinando en cada caso cuando debemos filtrar y cuando no. Para seguir con la idea del procesamiento de 3 pixeles (o 5 en el caso de copia sin modificacion) establecemos un contador el cual no indica cuantos pixeles fueron analizados previamente. Si al analizar un pixel el contador no esta en cero, entonces podemos evitar la lectura de

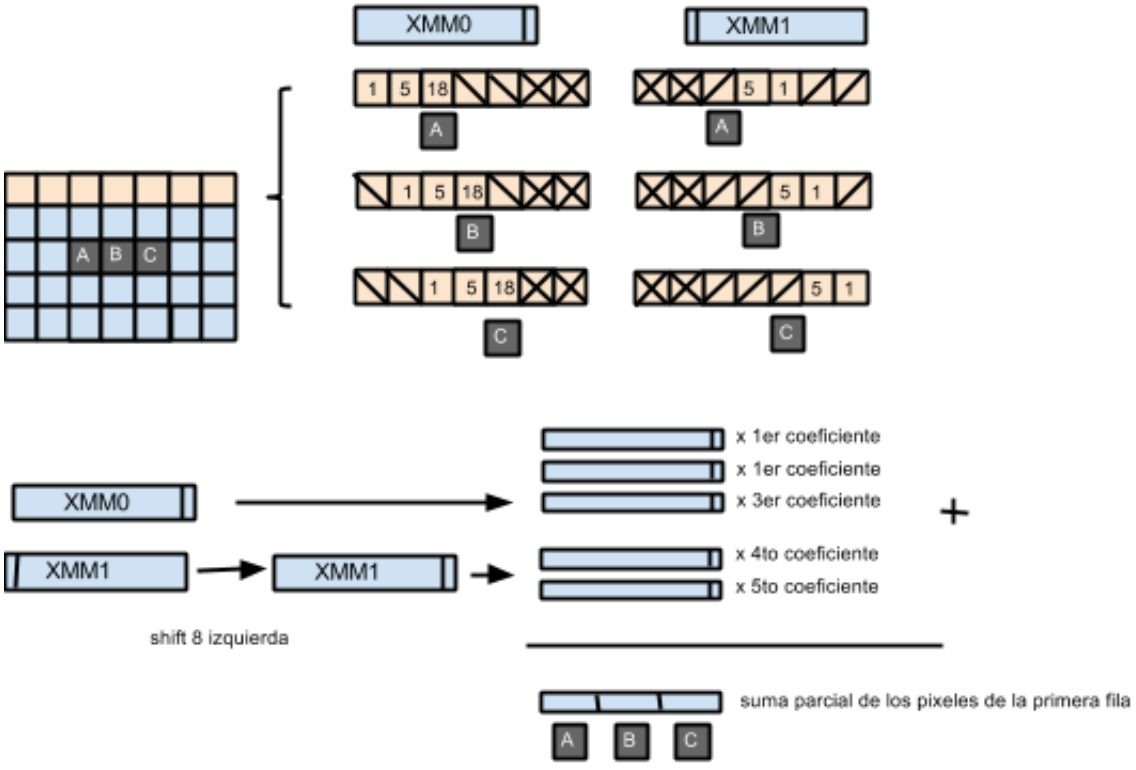
memoria de ese pixel. Al estar escasos de registros de proposito general, se utilizo un solo registro para llevar el contador de ambos: el mismo toma valores de 0 a 5 en caso del levantamiento por copia, y de 6 a 8 para los filtrables:



2.2.4. Detalle de procesamiento de filtrado de a 3 pixeles

Suponiendo a los pixeles A, B, C, pixeles a ser filtrados, vamos a detallar el calculo de los colores. Para el procesamiento se va a necesitar un rectangulo de pixeles que contendra toda la informacion que necesitamos evaluar para el resultado final. Cada fila de este rectangulo se puede obtener mediante 2 accesos de memoria, como puede verse en la imagen anterior de solapamiento de datos.

Los datos obtenidos en estos 2 registros son utiles para el calculo de cada uno de los 3 pixeles, siendo utiles para mas de uno a la vez, con lo cual, se los debe multiplicar por mas de un coeficiente distinto.



Como cada resultado de cada coeficiente le es util de forma ordenada a cada pixel objetivo de forma vertical, se puede aprovechar su uso para un procesamiento en paralelo. Como puede verse en la imagen anterior, en el caso del coeficiente 1 en el XMM0, multiplica el primer, segundo y tercer pixel. Usando esto, podemos multiplicar directamente a todos los pixeles de la fila (osea a cada color) por el coeficiente y sumarselo a las sumas parciales de cada uno de los 3 pixeles objetivo (ya que estamos usando solo 3

pixeles en cada levantada del XMM0 y XMM1).

En el caso del primer coeficiente, los pixeles quedan justo alineados para sumar al resultado parcial. Esto no ocurre para los demas, estando para el segundo coeficiente desfasados en un pixel y para el tercero en 2. Para el 4to y quinto los pixeles se recuperan en un segundo XMM, el cual tiene un corrimiento de 1 pixel y 2/3 para levantar de memoria. Se hace de esta forma, con 1/3 (8 bits) de pixel anterior leido al principio para que todo lo levantado por los XMM sean datos validos y a usar en los calculos del filtro. Este corrimiento de 1/3 se compensa corriendo los datos del XMM 8 bits, ya que es un dato que ya se uso para los coeficientes anteriores, y no es necesario para los calculos del 4to y 5to, puede ser ignorado.

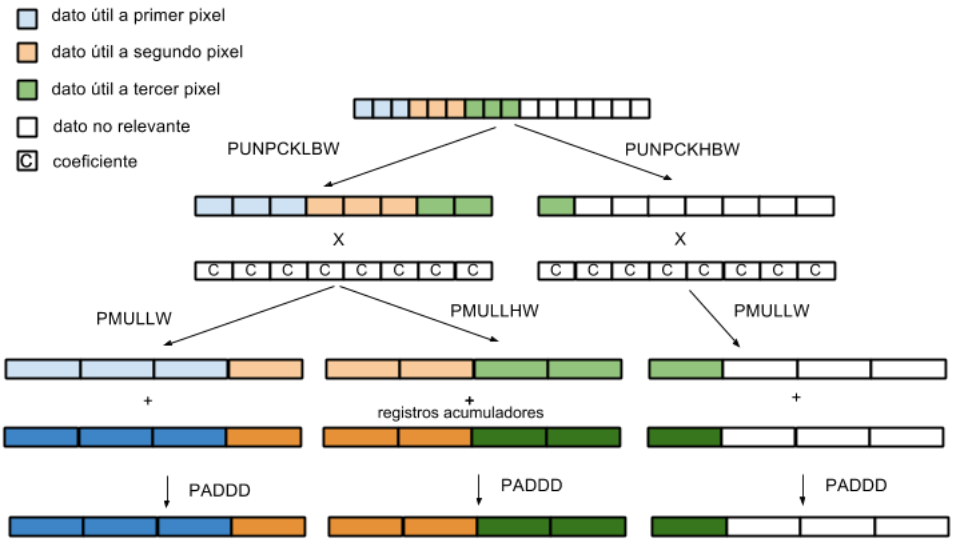
Al finalizar la multiplicacion por cada coeficiente (cuyo resultado esta en las primeras 3 posiciones del XMM) solo falta la suma con las sumas parciales guardadas anteriormente y luego pasar a la segunda fila. Para cada fila la logica del procedimiento es equivalente. En lo unico que varia es el los coeficientes de cada posicion y en el corrimiento por fila para levantar los datos de la matriz.

Al finalizar el procesamiento de los pixeles debemos dividir por el coeficiente normalizador (6) y el ajuste por considerar integer a los coeficientes de la matriz (100) y finalizar ordenando los pixeles obtenidos para escribirlos en memoria.

Para hacer estas multiplicaciones, sumas y divisiones en paralelo se tuvo que tener en cuenta la perdida de presicion de datos. Dado que cada color esta representado en 8 bits, al multiplicarlo por un valor entero se corre el riesgo de un overflow. El maximo numero posible con los coeficientes de la matriz estaria representado por el color 255 y el coeficiente 1 (osea 100) con lo cual tendrriamos al 25500 como supremo y maximo del conjunto, que entra en un tamaño de word.

Para la suma de cada colr esta sumando el valor de 25 words con un valor maximo de 25500 con lo cual se tendria como nueva cota a 637500, el cual entra en un tamaño de doubleword. Por esta razon, la suma total de cada pixel estaria representada por 3 doublewords, con lo cual el resultado de los 3 pixeles entra en 3 XMM, teniendo en el ultimo de ellos solo la primera doubleword con datos de color.

La operacion de desempaquetamiento se hace en otro orden, respetando la misma idea: primero se descomprimenl los 8 bits de cada color a word. Ahora se usa la operacion PMULLW y PMULLHW, las cuales hacen una multiplicacion de a word y devuelven un doubleword.. que seria exactamente lo que se necesita en este caso, ahorrandosse un paso de desempaquetamiento:

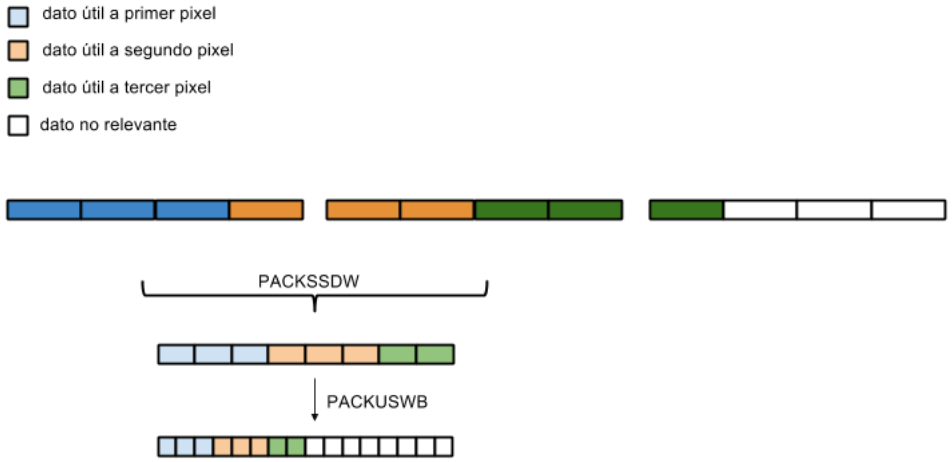


El siguiente paso es convertir estos 3 registros de double words en 1 de 16 bytes en donde los colores esten en orden, al igual que los pixeles para asi poder pasarlos de una a memoria.

Para esto primero vamos a normalizar el resultado por 600. Como se busca hacerlo en paralelo, genero un registro con 600 repetido en cada double word y divido de forma paralela. Para ello necesito pasar antes a punto flotante para poder usar las funciones SIMD, y luego retornar los valores a enteros. Con la operacion DIVPS pasamos a dividir los floats por floats empaquetados, luego con la operacion CVTDQ2PS recuperamos el resultado de las divisiones con redondeo por truncamiento a doubleword

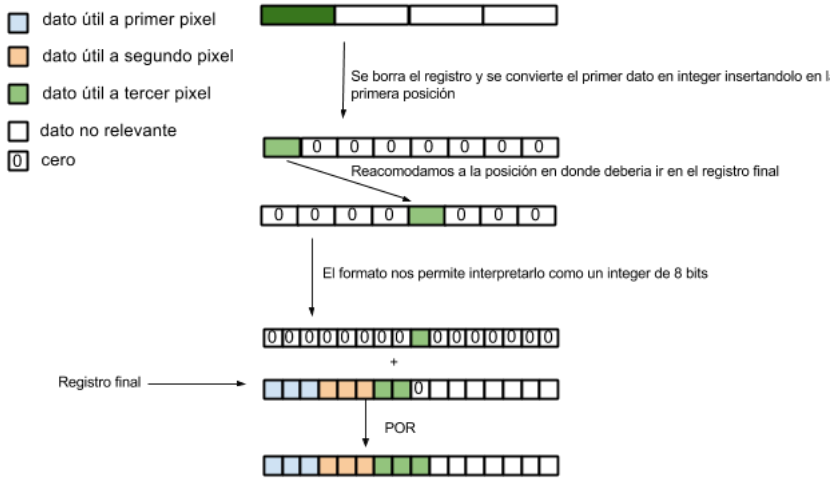
integers.

Teniendo ya los resultados finales, lo que falta es la reconversion a 8 bit de cada color. Para los 2 primeros registros de resultados podemos empaquetar ambos registros a word con una sola instruccion (PACKSSDW) y de word a byte con otra (PACKUSWB):



Como se puede ver en la imagen anterior, aun queda un color del ultimo pixel en el ultimo registro, con lo cual debemos insertarlo en el registro de retorno para poder guardarlo en memoria.

Para lograr esto vamos a hacer uso de una mascara que nos aislara el color en la posicion en la cual tengamos que insertarlo en el registro destino. El valor a aislar, aunque sea double word, vale notar que su valor no supera a 255, ya que posee el valor de un color valido, por lo que el numeo en integer ocupa el tamaño de 8 bits. Gracias a esto se lo puede considerar tambien como un word integer y usar la instruccion PINSRW para insertar en el lugar indicado del registro el valor. Al registro destino se le setearan los bits en 0 en el lugar donde tengamos que insertar el valor resagado y mediante una suma logica vamos a pegar el color en el lugar correspondiente. Al entrar en un byte el valor pegado, los ceros de la parte alta del word no cambian los valores anteriormente empaquetados.



2.2.5. C vs Assembler con SIMD

Las diferencias mas fuertes en la estructura del algoritmo en las implementaciones pueden reducirse a 2, y dejar las demas como consecuencias de estas: La cantidad de datos procesados y accesos a memoria. Como se explico anteriormente, la resolucion en C debe hacer 25 accesos a memoria por pixel filtrado y un acceso por pixel copiado (el que lo pasamos directo). Por otro lado, el algoritmo de assembler logra reducir los accesos a 10 por cada 3 pixeles evaluados, gracias al solapamiento de los cuadros de pixeles y la capacidad de los registros XMM de obtener de a tandas de 5 pixeles. Por esto tambien se pudo reducir los accesos para dejar pixeles iguales, copiando de a 5 pixeles por vez, la velocidad del algoritmo aumenta drasticamente por la manipulacion en paralelo de estos datos que no necesitan procesamiento

alguno. Los resultados pueden ser vistos en los siguientes graficos que comparan la cantidad de ciclos consumidos por el CPU para 2 archivos distintos corridos sobre C y sobre assembler:

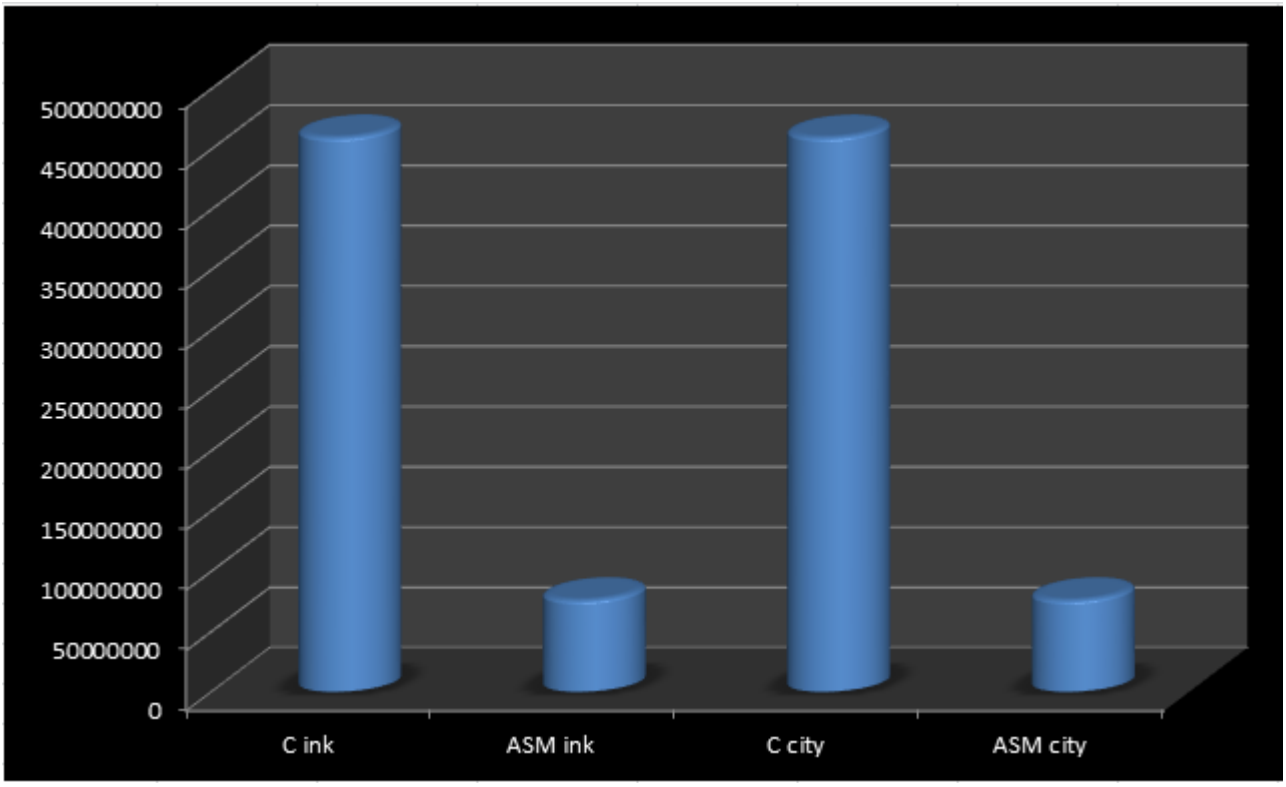
2.2.6. Tiempos de ejecucion de diferentes experimentos

Caracterizacion del experimento:

- **Iteraciones:** 5
- **Comando C City:** ./tp2 -t 5 miniature -i c city.avi 0.3 0.8 3
- **Comando ASM City:** ./tp2 -t 5 miniature -i asm city.avi 0.3 0.8 3
- **Comando C Ink**./tp2 -t 5 miniature -i c ink.avi 0.3 0.8 3
- **Comando Asm Ink**./tp2 -t 5 miniature -i asm ink.avi 0.3 0.8 3

Resultados:

Medición	C ink.avi	ASM ink.avi	C city.avi	ASM city.avi
Ciclos por llamada	459987456	76210288	459888288	76193352



2.2.7. Comentarios sobre los resultados obtenidos del experimento

Las diferencias impactan notablemente sobre los tiempos de ejecucion del algoritmo aumentando notoriamente la velocidad de procesamiento de los videos objetivos en un %600. De esta forma se puede concluir que a la hora del procesamiento de archivos de medios, las instrucciones SIMD logran un mejor aprovechamiento del poder de computo del CPU al permitirnos manipular mayor cantidad de datos y de forma paralela sobre memoria disminuyendo los accesos a ella.

2.3. Decode

El filtro decode consiste en la decodificacion de un texto embebido en los bits menos significativos de los pixeles de una imagen. Cada 4 bytes, se toman los dos bits menos significativos de cada uno y se obtiene un nuevo byte que es la codificacion ascii de un caracter del texto. Tambien se utilizan el tercer y cuarto bit menos significativo de cada byte para saber como interpretar esos bits y aplicarles operaciones.

Se implemento una primera version del algoritmo en C, analicemos el algoritmo y luego veamos que optimizaciones se le aplicaron.

2.3.1. Descripcion del algoritmo

Nota: la idea de este codigo es entender como funciona nuestro algoritmo, por ello se eliminaron varias partes del codigo original.

```
1  //Ciclo
2      for(int index=0;index<CANT_BYTES_MAT_MESSAGE;index+=4){
3          //leo los 4 bytes que forman un char
4          unsigned char fstByte = src[index+OFFSET_FIRST];
5          unsigned char sndByte = src[index+OFFSET_SECOND];
6          unsigned char trdByte = src[index+OFFSET_THIRD];
7          unsigned char frtByte = src[index+OFFSET_FOURTH];
8
9          //obtengo el code
10         unsigned char codeFst = (fstByte & 3);//00000011 = 3
11         unsigned char codeSnd = (sndByte & 3);//00000011 = 3
12         unsigned char codeTrd = (trdByte & 3);//00000011 = 3
13         unsigned char codeFrt = (frtByte & 3);//00000011 = 3
14
15         //obtengo los op
16         unsigned char opFst = (fstByte & 12) >> 2;//00001100 = 12
17         unsigned char opSnd = (sndByte & 12) >> 2;//00001100 = 12
18         unsigned char opTrd = (trdByte & 12) >> 2;//00001100 = 12
19         unsigned char opFrt = (frtByte & 12) >> 2;//00001100 = 12
20
21         //proceso los chars salientes segun los codes.
22         unsigned char salienteFst = modificarSegunOp(opFst, codeFst);
23         unsigned char salienteSnd = modificarSegunOp(opSnd, codeSnd);
24         unsigned char salienteTrd = modificarSegunOp(opTrd, codeTrd);
25         unsigned char salienteFrt = modificarSegunOp(opFrt, codeFrt);
26
27         //armo el char saliente
28         unsigned char final = (salienteFrt << 6) | (salienteTrd << 4) | (salienteSnd << 2) | (salienteFst);
29         code[j] = final;
30         j++;
31     }
32 //modificacion segun op
33     static inline unsigned char modificarSegunOp(unsigned char op, unsigned char code){
34         unsigned char res;
35         switch(op){
36             case 0://00000000
37                 res=code;
38                 break;
39             case 1://00000001
40                 //res=(code == 3) ? 0: (code+1)%4;
41                 res=(code+1)%4;
42                 break;
43             case 2://00000010
44                 res = (code>0) ? ((code-1)%4) : 3;
45                 break;
46             case 3://00000011
47                 res=~(code);
48                 res = res & 3;//00000011 = 3
49                 break;
50         }
51         return res;
52     }
53 }
```

Explicacion del algoritmo:
Analicemos por partes que hace este algoritmo:

- **Lectura de 4 bytes de origen:** Lineas 4-7
  En esta sección se leen de origen 4 bytes que contendrán en los 4 bits menos significativos de cada uno la informacion correspondiente a un char del mensaje codificado.

- **Filtrado con mascaras de valores relevantes:** Lineas 10-19
  Luego, debemos separar de alguna manera los bits que nos interesan del resto, para ello utilizamos mascaras de bits que por medio de operaciones logicas de conjuncion y corrimiento nos permiten quedarnos con los bits que nos interesan, en este caso code(primeros 2 bits) y op(tercer y cuatro bit).

- **Procesado del code segun el op:** Lineas 22-25
  Ya tenemos organizados los datos de las etapas anteriores de forma tal que podemos aplicar una serie de comparaciones al op para saber cual operacion realizarle al code correspondiente, para esto utilizamos una funcion auxiliar que nos devuelve el code listo para ser combinado.

- **Combinacion y escritura a destino:** Lineas 28-30
  Tal como dice el enunciado, realizamos un mezclado de los 4 bytes(notar que solo se utilizan 2 bits de cada uno) por medio de corrimientos y operaciones logicas de disyuncion. De esta forma hemos armado un caracter del mensaje, el cual es escrito al buffer de salida pasado por parametro.

Optimizaciones realizadas:

- **Eliminacion de la llamada a la funcion auxiliar por medio de modificador inline:**
  De esta forma, con este modificador pretendemos eliminar los saltos de la llamada a funcion, requiriendole al compilador introducir el cuerpo de la funcion en cada invocacion a esta.

Detalles de implementacion:

- **Multiplicidad de la cantidad de bytes a procesar:**
  En nuestra implementacion no tuvimos en cuenta el caso donde la cantidad de bytes a procesar no es multiplo de 16, es decir, que el ultimo ciclo, leeria memoria invalida al introducir una entrada que no respete esta precondicion de multiplicidad de tamaño.Para solucionarlo se podria implementar una solucion que consiste en iterar sobre los primeros k-bytes tal que k sea el primer multiplo de 16 hacia menos infinito. y luego, fuera de dicho ciclo, restarle al puntero 16 menos el resto modulo 16 del tamaño de la entrada y hacer un salto no condicional al comienzo del ciclo, luego de ejecutarse una vez mas, automaticamente la guarda se vuelve falsa y termina el ciclo.

2.3.2. Implementacion en assembler utilizando set de instrucciones SIMD

Luego de realizar la anterior implementacion en C, nos pusimos a pensar la mejor manera de utilizar SIMD con el set de instrucciones que teniamos a nuestro alcance. Consideramos realizar la implementacion en assembler con la ventaja de poder utilizar los registros XMM de un tamaño de 16 bytes y maximizar su capacidad para leer mas de 4 bytes en cada ciclo. Dicho esto, la version en assembler obtiene 16 bytes por ciclo, y con las instrucciones de operatorias empaquetadas se espera procesar 4 veces mas datos por ciclo.

Nota: En esta implementacion nos dimos cuenta que en muchos lugares utilizamos unicamente 2 bits de los 16 bytes empaquetados, pero dadas las instrucciones a nuestro alcance, consideramos que es la mejor implementacion en SIMD respetando un balance entre el set de instrucciones a nuestro alcance y el rendimiento.

2.3.3. Analisis

Antes de comenzar el ciclo principal nos vimos con la necesidad de declarar algunas mascaras como variables globales inicializadas para la manipulacion de los bits, y las cargamos en registros XMM para evitar esos accesos a memoria en cada iteracion de ciclo, que reducen significativamente la velocidad del programa. Estas mascaras son:

XMM14	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3

Figura 24: 00000011 replicado, para hacer AND y filtrar los ultimos 2 bits

XMM15	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12

Figura 25: 00001100 replicado, para hacer AND y filtrar el tercer y cuarto bit

XMM10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Figura 26: para comparar los resultados y ver si es el op 0

XMM11	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01	01

Figura 27: para comparar los resultados y ver si es el op 1

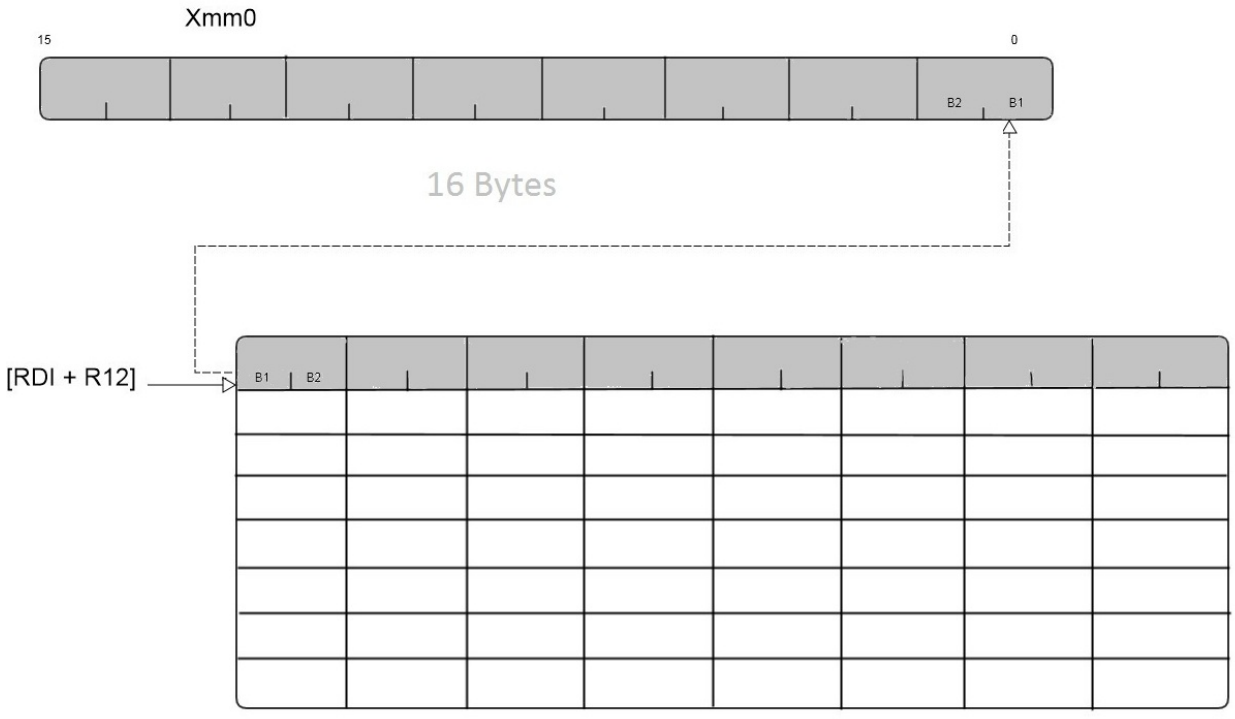
XMM12	02	02	02	02	02	02	02	02	02	02	02	02	02	02	02	02

Figura 28: para comparar los resultados y ver si es el op 2

XMM13	03	03	03	03	03	03	03	03	03	03	03	03	03	03	03	03

Figura 29: para comparar los resultados y ver si es el op 3

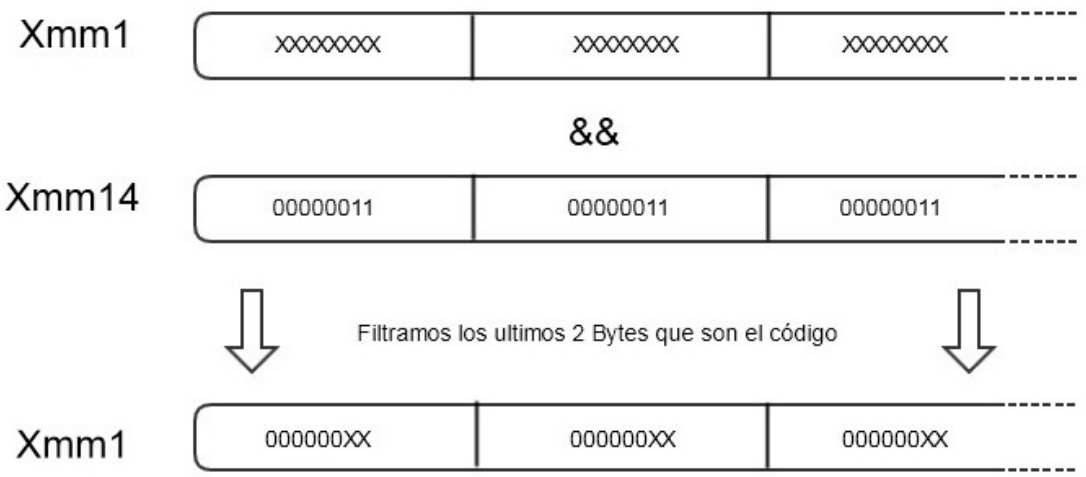
- El primer paso es obtener los 16 bytes de la matriz mediante la operacion MOVDQU XMM0, [RDI + R12] (donde RDI = src y r12 = index) y guardarlos en el registro `XMM0`.



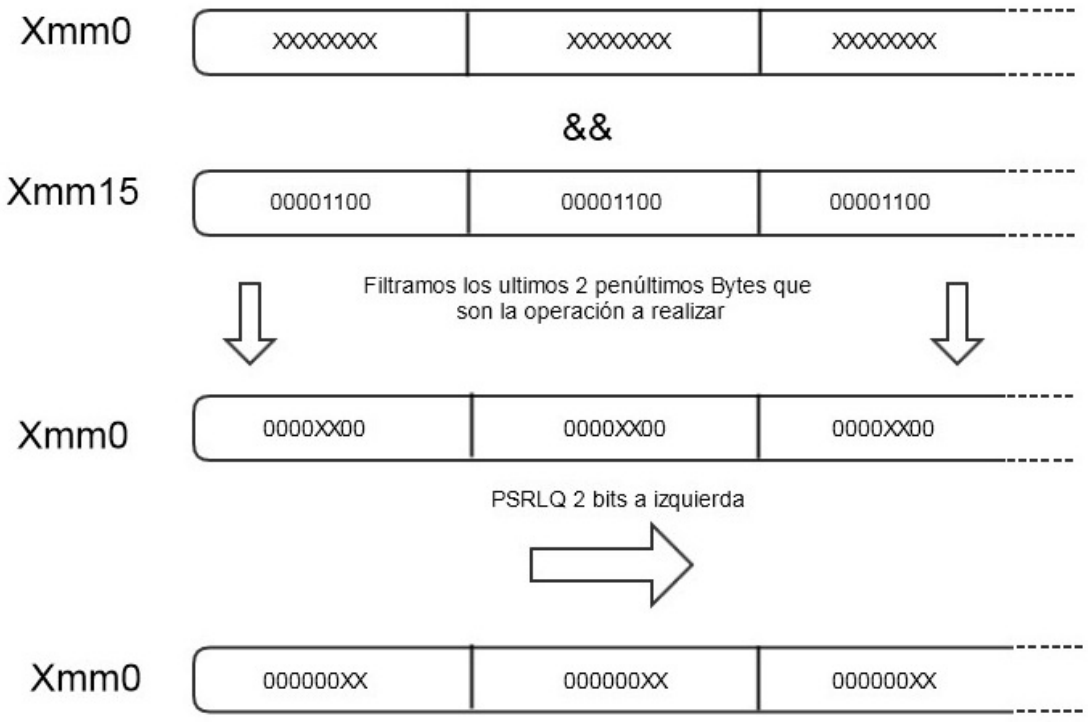
- Replicamos esos datos en `XMM1`. Al hacer una operacion de conjuncion entre `XMM1` con la mascara de `XMM14` obtenemos los bits menos significativos de cada byte que son los codigos. Luego hacemos

una nueva operacion de disyuncion logica entre `XMM0` contra la mascara guardada en `XMM15` y de esta forma nos quedan el tercer y cuarto bit de cada byte filtrado, que son la operacion a realizar sobre los codigos. Pero las operaciones nos quedan en el tercer y cuarto bit por eso utilizamos una intruccion de corrimiento PSRLQ para mover dos bits a la izquierda los datos obtenidos. De esta forma tenemos en `XMM1` los codigos y en `XMM0` los codigos de operaciones que nos dirán que modificacion realizar sobre ellos.

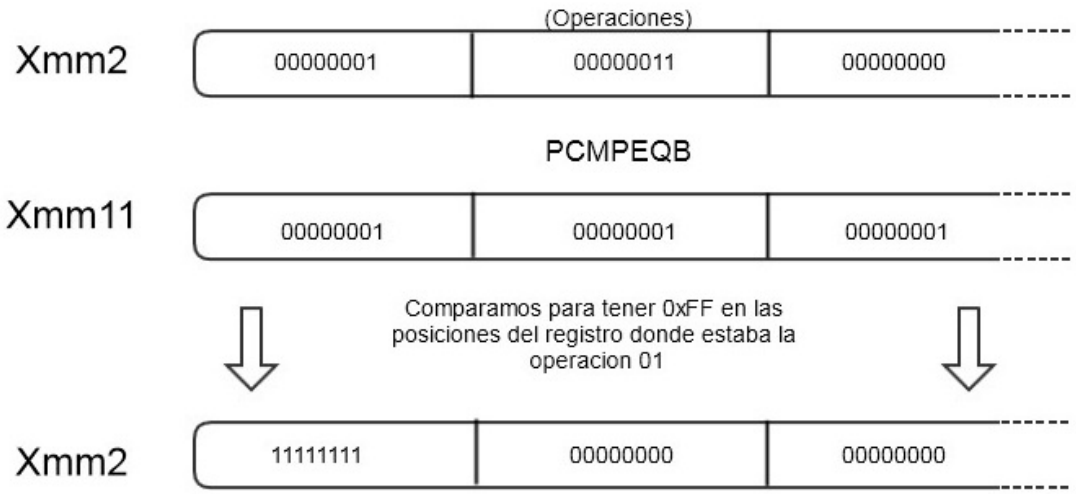
Obtener codigos:



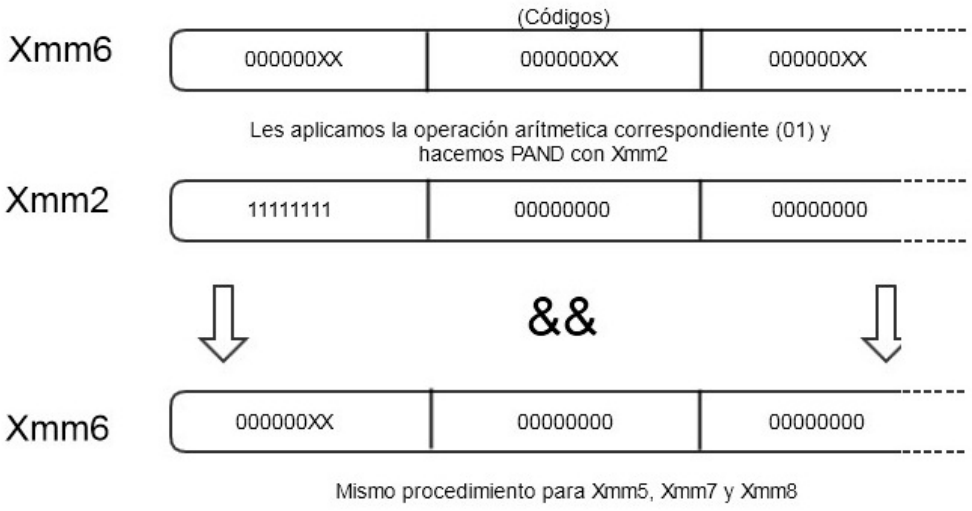
Obtener operaciones:



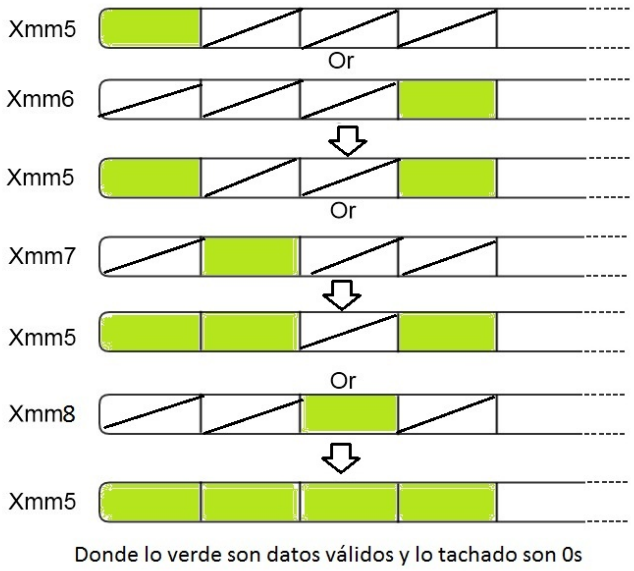
- Replicamos los codigos de operacion en `XMM2`, `XMM3` Y `XMM4`. Paso seguido comparamos mediante la operacion PCMPEQB de comparacion empaquetada por igualdad cada registro `XMM0`, `XMM2`, `XMM3` y `XMM4` con las mascaras guardadas en los registros `XMM10`, `XMM11`, `XMM12`, `XMM13` respectivamente. Obtenemos en los registros el resultado de las comparaciones en forma de mascaras que contienen en cada posicion 0x00 y 0xFF dependiendo de si, para cada byte, en su posicion, estaba la operacion 0, 1, 2 o 3 originalmente en el registro comparado. ie. (`XMM0` tiene 0XFF en las posiciones donde estaba la operacion 0, `XMM2` tiene 0XFF en las posiciones donde estaba la operacion 1, etc.)



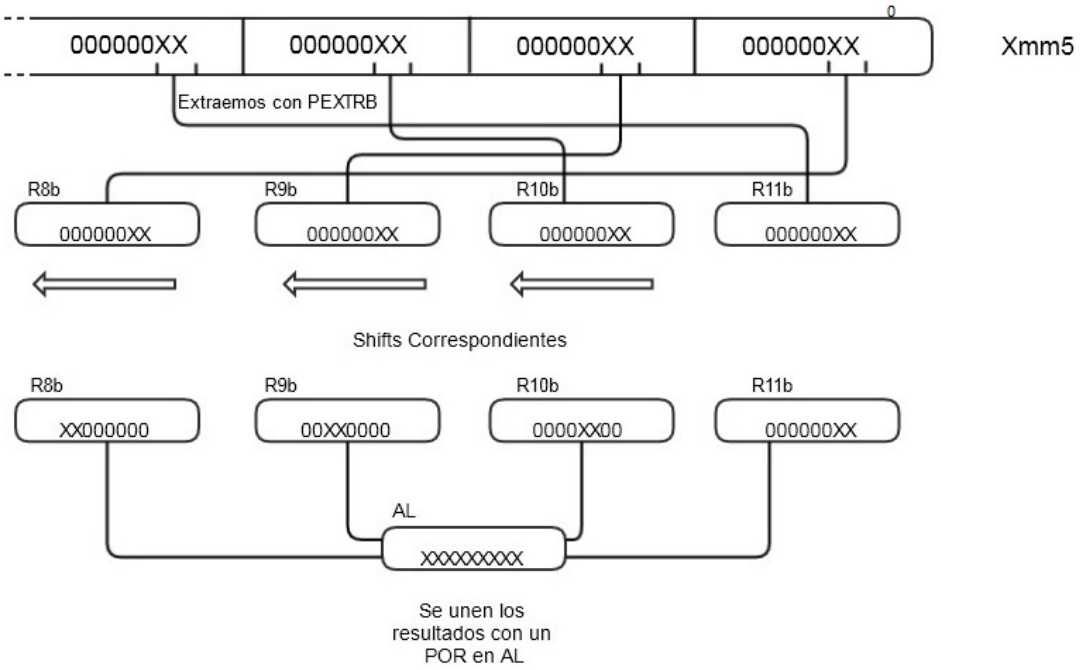
- En los registros `XMM5`, `XMM6`, `XMM7` y `XMM8` replico los codigos (anteriormente en `XMM1`), y aplico a cada uno una de las operaciones a realizar sobre codigos denotadas por los codigos de operacion. Luego, para cada registro, dependiendo de la operacion que le aplique, le aplicamos una operacion logica de disyuncion contra la mascara correspondiente `XMM0`, `XMM2`, `XMM3` y `XMM4`) a los bytes en cuyo indice estaba esa operacion. De esta forma entre `XMM5`, `XMM6`, `XMM7` y `XMM8` me quedan todos los codigos con las operaciones correspondientes aplicadas y notar que los bytes empaquetados , no se superponen entre registros. De esta forma, con una conjuncion logica puedo juntar todos en un solo registro y tengo en `XMM5` los 16 bytes que solo tienen los 2 bits mas bajos ya procesados por las operaciones.



Unimos resultados mediante POR's:



- Lo unico que queda pendiente es la combinacion de los codigos procesados para formar 4 caracteres ascii. Logramos esto mediante el uso repetido de la instruccion PEXTRB para extraer 4 bytes del registro `XMM5`, copiarlos en `R8`, `R9`, `R10` y `R11`,y realizar los corrimientos pertinentes a cada uno con el objetivo de acomodarlos en la posicion correcta y formar un byte entre todos aplicando una operacion logica de disyuncion. Esto se hace una vez por cada uno de los 4 caracteres, y la combinacion se almacena en los registros `AL`, `BL`, `CL`, `DL`



- Finalmente copiamos los 4 bytes que contienen los 4 caracteres al buffer de salida.

2.3.4. Optimizaciones aplicadas al algoritmo

- **Implementacion de software pipelining - Ejecucion fuera de orden** Se implemento esta de la mejor forma a nuestro alcance tecnica para aprovechar la ejecucion fuera de orden que provee el cpu. Esto permite que instrucciones no *dependientes*$_{*1}$ unas de las otras sean procesadas por el cpu favoreciendo el pipeline al utilizar mas unidades de procesamiento internas de la CPU, reduciendo al minimo los stalls. Nuestra implementacion sobre el codigo assembler del filtro decode fue reorganizar las instrucciones de forma tal que el cpu pueda comenzar a ejecutar instrucciones no

dependientes posteriores antes de finalizar la ejecucion de la instruccion actual. Se intento realizar la implementacion desenrrollando el ciclo en 2 procesamientos por ciclo y superponer instrucciones iguales no dependientes para lograr un mayor impacto pero abandonamos esta implementacion porque mientras nos dispusimos a realizarla, nos quedamos sin registros `XMM` para utilizar dentro del ciclo, tal vez podriamos haber encontrado la manera de reemplazar accesos a mascaras estaticas desde registros por accesos a memoria, liberando asi mas registros, pero tomamos la decision de no hacerlo, dado que los accesos a memoria impactarian proporcionalmente al tamaño de la entrada del algoritmo y no nos parecio para nada favorable.

Mas info en: `http://en.wikipedia.org/wiki/Software_pipelining`
∗1 `http://en.wikipedia.org/wiki/Hazard_(computer_architecture)`

2.3.5. Analisis y rendimiento de las diferentes implementaciones

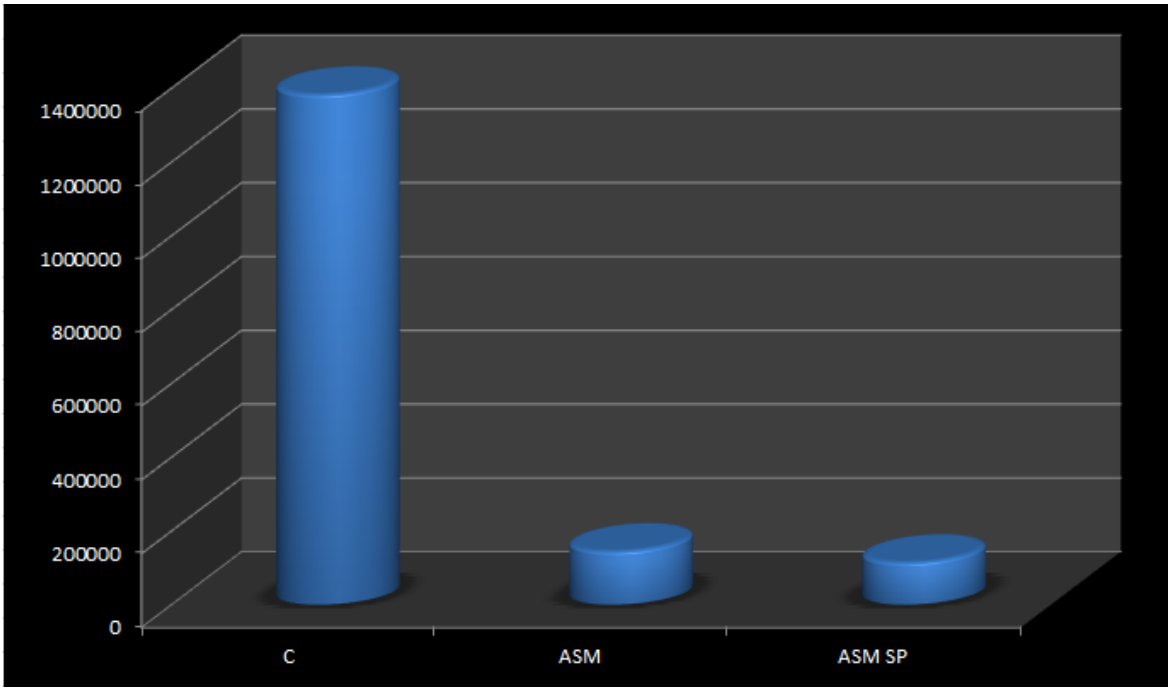
La principal diferencia estructural de las dos implementaciones, es, como puede verse en los respectivos codigos, que la implementacion en C en cada iteracion del ciclo accede a memoria y obtiene 4 bytes de forma secuencial, para luego procesarlos y obtener un caracter del texto, mientras que la implementacion en assembler se obtienen 16 bytes en cada iteracion del ciclo y son procesados paralelamente por medio instrucciones de tipo SIMD para obtener 4 caracteres por iteracion. Intuitivamente la implementacion en assembler deberia realizar 4 veces mas trabajo de procesamiento por ciclo. Todo esto sin considerar ademas temas del compilador, que no utiliza instrucciones SIMD para optimizar, realizando el procesamiento de estos datos en forma secuencial y no paralela.

Nota: La medicion se realiza sobre el metodo decode, sin tener en cuenta procesamientos externos a esta funcion, asi tambien se obtiene un promedio de la cantidad de aplicaciones de la funcion determinadas por el parametro t al invocar el programa. Se probo la medicion con varios archivos provistos por la catedra, al ser de tamaño relativamente parecido, no se observaron diferencias al llevar a cabo distintos experimentos.

Caracterizacion del experimento:

- **Iteraciones:** 1000
- **Comando:** ./tp2 -t 1000 decode -i < *implementacion* > ../data/base/encoded.bmp

Medición	Implementación C	ASM Plano	ASM Soft. Pipeline
Ciclos por llamada	1382837.000	144664.578	114466.883



2.3.6. Conclusiones de los resultados

Se puede observar una mejora notable entre las versiones C y ASM, se ejecuta aproximadamente 10 veces mas rapido la version en assembler, superando las expectativas de mejora de 4 veces estimadas al comienzo de la implementacion. Asi mismo se obtuvo una mejora aproximada del 27 % contra el assembler original aplicando software pipelining. Lo que nos da una mejora total aproximada entre C y ASM con SP de 1200 %

3. Conclusión

Durante todo este trabajo, pudimos comprobar de forma empirica las ventajas que brindan las instrucciones del set SIMD provisto por Intel. A pesar de las optimizaciones brindadas por el compilador *GCC* que mejoran de forma dramatica los resultados entre las implementaciones en C que no estan optimizadas y las versiones que si lo son, la paralelizacion de lectura y calculos para la aplicacion de los filtros, que en definitiva son procesamientos aritmeticos sobre matrices de numeros, redujeron la cantidad de ciclos requeridos para obtener un algoritmo que realice de forma correcta la transformacion impuesta por los diferentes filtros.
Mas alla de estos resultados favorables obtenidos, algunas de las implementaciones de los filtros en assembler con instrucciones SIMD tuvieron que ser planteadas desde otro punto de vista para un mejor aprovechamiento del set de instrucciones a nuestro alcance, lo cual nos ocasiono varias trabas a lo largo del trabajo. Luego de obtener las primeras mediciones, se realizaron sucesivas optimizaciones sobre las implementaciones de los filtros, tanto en C como en Assembler, con el objetivo de ir disminuyendo la cantidad de ciclos consumidos por estos viendose esto reflejado en los analisis publicados en este informe.