DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

subtitulo del trabajo

Organización del Computador II

Primer Cuatrimestre de 2014

Integrante	LU	Correo electrónico
Tomas Shaurli	671/10	tshaurli@gmail.com
Sebastian Aversa	379/11	sebastianaversa@gmail.com
Fernando Gabriel Otero	424/11	fergabot@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Resumen

En el presente trabajo se describe el análisis de siertos filtros para imágenes, comparando las versiones realizadas en C y en assembler con procesamiento SIMD.

El análisis principal será el tiempo que tardan dichos algoritmos y cómo se ha encarado el algoritmo en cada lenguaje.

Índice

1. Objetivos generales

El objetivo de este Trabajo Práctico fue el aprendizaje de las instrucciones SIMD(Single Instruction, Multiple Data) mediante su uso en filtros de imágenes, y las comparaciones entre los algoritmos en assembler realizados con éstas instrucciones y los mismos algoritmos en C. Tanto en los lenguajes assembler como C se compiló y ensambló para arquitecturas Intel x64.

2. Contexto

Los algoritmos descriptos y analizados fueron probados en las computadoras de los laboratorios de la facultad. Todas las comparaciones fueron realizadas en las mismas computadoras para un correcto análisis. Las implementaciones en assembler fueron hechas específicamente para el uso de las instrucciones SIMD de Intel x64 con los siguientes parámetros:

- -felf64:
- -g:
- -F:
- dwarf:

Los sigientes parámetros fueron usados en el compilador de C:

- ggdb:
- -Wall:
- -std=c99:
- -pedantic:
- -m64:
- pkg-config:
- –cflags:
- –libs opencv:

3. Filtro de tiles

3.1. Enunciado

Programar el filtro *tiles* en lenguaje C y luego en ASM haciendo uso de las instrucciones vectoriales (**SSE**).

Experimento 1 - análisis el código generado

Utilizar la herramienta `objdump` para verificar como el compilador de C deja ensamblado el código C. Como es el código generado, ¿cómo se manipulan las variables locales?¿le parece que ese código generado podría optimizarse?

Experimento 2 - optimizaciones del compilador

Compile el código de C con optimizaciones del compilador, por ejemplo, pasando el flag `-O1`[1]. ¿Qué optimizaciones realizó el compilador? ¿Qué otros flags de optimización brinda el compilador? ¿Para qué sirven?

Experimento 3 - secuencial vs. vectorial

Realice una medición de las diferencias de performance entre las versiones de C y ASM (el primero con -O1, -O2 y -O3).
¿Como realizó la medición?¿Cómo sabe que su medición es una buena medida?¿Cómo afecta a la medición la existencia de *outliers*[2]?¿De qué manera puede minimizar su impacto?¿Qué resultados obtiene si mientras corre los tests ejecuta otras aplicaciones que utilicen al máximo la CPU? Realizar un análisis **riguroso** de los resultados y acompañar con un gráfico que presente estas diferencias.

Experimento 4 - cpu vs. bus de memoria

Se desea conocer cual es el mayor limitante a la performance de este filtro en su versión ASM.

¿Cuál es el factor que limita la performance en este caso? En caso de que el limitante fuera la intensidad de cómputo, entonces podrían agregarse instrucciones que realicen accesos a memoria y la performance casi no debería sufrir. La inversa puede aplicarse si el limitante es la cantidad de accesos a memoria.

Realizar un experimento, agregando múltiples instrucciones de un mismo tipo y realizar un análisis del resultado. Acompañar con un gráfico.

Experimento 5 (*opcional*) - secuencial vs. vectorial (parte II)

Si vemos a los pixeles como una tira muy larga de bytes, este filtro en realidad no requiere ningún procesamiento de datos en paralelo. Esto podría significar que la velocidad del filtro de C puede aumentarse hasta casi alcanzar la del de ASM. ¿ocurre esto?

Modificar el filtro para que en vez de acceder a los bytes de a uno a la vez se accedan como tiras de 64 bits y analizar la performance.

[1] agregando este flag a `CCFLAGS64` en el makefile
[2] en español, valor atípico: `http://es.wikipedia.org/wiki/Valor_atpico`

3.2. Análisis previo

El filtro de tiles define un rectángulo en una imagen en color. La función debe replicar dicho sector de la imágen en el resto de la imagen original. Para esto, se nos ocurrieron tres formas de encararlo: con la fórmula dada en el enunciado (versión C), guardando el primer elemento del rectángulo a copiar y guardando un puntero a ese primer elemento.

Formula:

i y j representan fila y columna respectivamente.

dst(i,j) = src [(i mod tamy)+offsety][(j mod tamx)+offsetx]

3.3. Implementación en C

La implementación en C del ejercicio tiles es tomar un rectángulo de la imagen original y repartirla por la imagen destino. Se resuelve con dos ciclos for anidados usando la forma dada en el enuciado.

3.4. Implementación en assembler

Para la resolución en assembler creamos los dos ciclos anidados con los registros de propósito general. AL comienzo del segundo ciclo se hace la copia.

```
MOVDQU XMM0, [RDI]
MOVDQU [RSI], XMM0
```

Luego, para ciclar y recorrer toda la matriz se modifican los valores de RSI y RDI cuando es necesario.

3.5. Resultado de los experimentos

Experimento 1 - análisis el código generado El compilador genera más variables locales que las usadas en nuestro código assembler. En ese sentido, estamos realizando una mejora al código de memoria y velocidad

Experimento 2 - optimizaciones del compilador El compilador ofrece los flags -O1 a -03 para distintos niveles de optimización, asi como los flags para indicar si es assembler de intel, amd, etc. Esto ultimo hace más específico el codigo assembler y correría mejor en una maquina con el microprocesador seleccionado,aunque no correría en uno de otra marca o de una familia de microprocesadores anterior a la elegida.

Experimento 3 - secuencial vs. vectorial

Experimento 4 - cpu vs. bus de memoria

Experimento 5 (*opcional*) - secuencial vs. vectorial (parte II)

4. Filtro Popart

4.1. Enunciado

Filtro *Popart*

Programar el filtro *Popart* en lenguaje C y en en ASM haciendo uso de las instrucciones vectoriales (**SSE**).

Experimento 1 - saltos condicionales

Se desea conocer que tanto impactan los saltos condicionales en el código del ejercicio anterior con `-O1`.

Para poder medir esto, una posibilidad es quitar las comparaciones al procesar cada pixel. Por más que la imagen resultante no sea correcta, será posible tomar una medida del impacto de los saltos condicionales. Analizar como varía la performance.

Si se le ocurren, mencionar otras posibles formas de medir el impacto de los saltos condicionales.

Experimento 2 - cpu vs. bus de memoria

¿Cuál es el factor que limita la performance en este caso?

Realizar un experimento, agregando múltiples instrucciones de un mismo tipo y realizar un análisis del resultado. Acompañar con un gráfico.

Experimento 3 - prefetch

La técnica de *prefetch* es otra forma de optimización que puede realizarse. Su sustento teórico es el siguiente:

Suponga un algoritmo que en cada iteración tarda n ciclos en obtener un dato y una cantidad similar en procesarlo. Si el algoritmo lee el dato i y luego lo procesa, desperdiciará siempre n ciclos esperando entre que el dato llega y que se comienza a procesar efectivamente. Un algoritmo más inteligente podría pedir el dato $i+1$ al comienzo del ciclo de proceso del dato i (siempre suponiendo que el dato i pidió en la iteración $i-1$. De esta manera, a la vez que el procesador computa todas las instrucciones de la iteración i, se estáran trayendo los datos de la siguiente iteración, y cuando esta última comience, los datos ya habrán llegado.

Estudiar esta técnica y proponer una aplicación al código del filtro en la versión ASM. Programarla y analizar el resultado. ¿Vale la pena hacer prefetching?

Experimento 3 - secuencial vs. vectorial

Analizar cuales son las diferencias de performace entre las versiones de C y ASM. Realizar gráficos que representen estas diferencias.

4.2. Análisis previo

El algoritmo de popart reemplaza cada pixel por uno de cinco pixeles preseleccionados. La elección resulta de la suma de los elementos del pixel.

En un primer análisis, el filtro popart tiene 3 partes: el ciclo, la suma de los elementos rgb, y la elección del pixel de reemplazo.

4.3. Implementación en C

Para la implementación en C usamos un vector con los posibles reemplazos para cada pixel, y una función que devuelve cual va a ser ese reemplazo mediante comparaciones.

```
rgb_t colores[] = { {255,   0,   0},
                    {127,   0, 127},
                    {255,   0, 255},
                    {  0,   0, 255},
                    {  0, 255, 255} };


int pop_art(unsigned int r, unsigned int g, unsigned int b)
{
    unsigned int suma = r + g + b;
    int res;

    if   (suma < 153) {res = 0;}
    else if (suma < 306) {res = 1;}
    else if (suma < 459) {res = 2;}
    else if (suma < 612) {res = 3;}
    else  {res = 4;}

    return res;
}
```

4.4. Implementación en assembler

Armamos máscaras para las comparaciónes con suma(i,j).

```
seisDoce:          DD 0x264, 0x264, 0x264, 0x264    ; 612, 612, 612, 612
cuatroCincoNueve:  DD 0x1CB, 0x1CB, 0x1CB, 0x1CB    ; 459, 459, 459, 459
tresOSeis:         DD 0x132, 0x132, 0x132, 0x132    ; 306, 306, 306, 306
unoCincoTres:      DD 0x99, 0x99, 0x99, 0x99        ; 153, 153, 153, 153
```

También para reemplazar los pixeles (solo cito el primer caso).

```
colores0:          DB  0xFF, 0x0,  0x0, 0x0,        ; 255,   0,   0,   0
                   DB  0xFF, 0x0,  0x0, 0x0,        ; 255,   0,   0,   0
                   DB  0xFF, 0x0,  0x0, 0x0,        ; 255,   0,   0,   0
                   DB  0xFF, 0x0,  0x0, 0x0         ; 255,   0,   0,   0
```

El resto de las máscaras, usadas con `PSHUFB`.
son:

- pixel0BGR : Para tener un pixel (R,G y B) en cada *double word*. El cuarto Byte se rellena con ceros
- soloPrimero : Lo usamos para trabajar sobre el primer Byte de cada *double word*
- pixelFinal : Sirve para sacar el ceros ubicado en el cuarto Byte de cada *double word*. El ultimo DB se rellena con ceros

Éstas máscaras son usadas en *pop_art* para discriminar los Bytes de R, G y B y guardar la suma de cada uno en un *double word*

El uso de *double word* viene del hecho de que un pixel se compone de tres elementos de un Byte cada uno, y necesitamos tener cada pixel en una cantidad par de Bytes para poder desempaquetar correctamente. Esto nos lleva a usar 4 Bytes por cada pixel.

Otra parte del codigo importante es el que genera la sumatoria, para lo cual utilizamos los `PSHUFB`:

```
MOVDQU XMM0, [RDI] ; R|G|B|R|G|B|R|G|B|R|G|B|R|G|B|R coloco los primeros 16bytes
                        ; de la imagen en XMM1

    XORPD XMM2, XMM2
    XORPD XMM3, XMM3
    XORPD XMM4, XMM4
    MOVDQA XMM7, [pixel0BGR]
    PSHUFB XMM0, XMM7              ; ordeno en el registro los pixels de forma 0RGB
                                   ; (tengo 4 pixeles)
    MOVDQU XMM2, XMM0              ; XMM2: R|G|B|0|R|G|B|0|R|G|B|0|R|G|B|0
    PSHUFB XMM2, [soloPrimero]     ; XMM2: R|0|0|0|R|0|0|0|R|0|0|0|R|0|0|0
                                   ; solo dejamos azul para 'suma'
    MOVDQU XMM3, XMM0              ; XMM3: R|G|B|0|R|G|B|0|R|G|B|0|R|G|B|0
    PSRLD XMM3, 8                  ; XMM3: G|B|0|0|G|B|0|0|G|B|0|0|G|B|0|0
    PSHUFB XMM3, [soloPrimero]     ; XMM3: G|0|0|0|G|0|0|0|G|0|0|0|G|0|0|0
                                   ; solo dejamos verde para 'suma'
    MOVDQU XMM4, XMM0              ; XMM4: R|G|B|0|R|G|B|0|R|G|B|0|R|G|B|0
    PSRLD XMM4, 16                 ; XMM4: B|0|0|0|B|0|0|0|B|0|0|0|B|0|0|0
                                   ; solo dejamos rojo para 'suma'
    ;Ahora podemos hacer suma
    PADDW XMM2, XMM3
    PADDW XMM2, XMM4               ; XMM2: dword[R+G+B] dword[R+G+B] dword[R+G+B] dword[R+G+B]
```

4.5. Resultado de los experimentos

Experimento 1 - saltos condicionales

Experimento 2 - cpu vs. bus de memoria

Experimento 3 - prefetch

Experimento 3 - secuencial vs. vectorial

5. Filtro Temperature

5.1. Enunciado

Filtro *Temperature*

Programar el filtro *Temperature* en lenguaje C y en en ASM haciendo uso de las instrucciones vectoriales (**SSE**).

Experimento 1
Analizar cuales son las diferencias de performace entre las versiones de C y ASM. Realizar gráficos que representen estas diferencias.

5.2. Análisis previo

El filtro temperature nuevamente presenta tres partes: el ciclo, la elección de cómo modificar los pixeles y la modificación propiamente dicha.

En éste caso, la eleccin de los pixeles sale del promedio de los elementos RGB, y la modificación se realiza en base al promedio.

5.3. Implementación en C

En este caso no separamos la comparacion del pixel de su modificación porque cada pixel se modifica usando el valor del promedio de los valores RGB. El resto de la implementación es el ciclo del main que recorre toda la imagen.

```
void temperatura(rgb_t* p_d, unsigned int r, unsigned int g, unsigned int b)
{
    unsigned int prom = (r + g + b) / 3;

    if  (prom < 32) {p_d->r = 0;  p_d->g = 0;  p_d->b = 128 + 4*prom;}
    else if (prom < 96) {p_d->r = 0;  p_d->g = -128 + 4*prom;  p_d->b = 255;}
    else if (prom < 160) {p_d->r = -384 + 4*prom; p_d->g = 255;  p_d->b = 639 - 4*prom;}
    else if (prom < 224) {p_d->r = 255;  p_d->g = 895 - 4*prom;   p_d->b = 0;}
    else  {p_d->r = 1151 - 4*prom; p_d->g = 0;  p_d->b = 0;}
}
```

5.4. Implementación en assembler

En este algoritmo tenemos ciertas partes muy parecidas a *popart_asm*, como la sumatoria de los elementos RGB con `PSHUFB` (para luego dividir por tres y obtener el promedio).

Sin embargo en este caso estas partes fueron encaradas de forma distinta.
Si comparamos los 2 algoritmos que realizan `PSHUFB`, el algoritmo de la función *popart* trabaja de a 4 pixels y la función *temp_aux* de a 2 pixels.

Pero por ciclo, *temperature_asm* procesa 5 pixeles en vez de los 4 que procesa *popart_asm* por ciclo

5.4.1. tempe:

```
tempe:
    PUSH RBP
    MOV RBP, RSP
        ;XMM0: R|G|B|R|G|B|R|G|B|R|G|B|R|G|B|R coloco los primeros 16bytes de la imagen en XMM1
        ;XMM0: 0|1|2|3|4|5|6|7|8|9|10|11|12|13|14|15
        ;llamare al resultado: R0|R1|R2|R3|R4|R5|R6|R7|R8|R9|R10|R11|R12|R13|R14|-
        MOVDQU XMM8, XMM0
        CALL temp_aux
        MOVDQU XMM9, XMM0
        PSHUFB XMM8, [moverPixeles]  ; XMM8: 6 |7 |8 |9 |10|11|12|13|14|15|- |- |- |- |- |-
        PSHUFB XMM9, [moverPixeles2] ; XMM9: - |- |- |- |- |- |- |- |- |R0|R1|R2|R3|R4|R5|-
        MOVDQU XMM0, XMM8
        CALL temp_aux
        MOVDQA XMM13, XMM0
        PSHUFB XMM13, [moverPixeles3]; xmm13: - |- |- |R6|R7|R8|R9|R10|R11|- |- |- |- | -|- |-

        PSHUFB XMM8, [moverPixeles1] ; xmm8:  12|13|14|- |- |- |- |- |- |- |- |- |- |- |- |-
        MOVDQU XMM0, XMM8
        CALL temp_aux                ; xmm0:  R12|R13|R14|basura
        PSHUFB XMM0, [unPixel]       ; xmm0:  R12|R13|R14|- |- |- |- |- |- |- |- |- |- |- |- |-

        POR XMM0, XMM13              ; xmm0: R12|R13|R14|R6|R7|R8|R9|R10|R11|- |- |- |- |- |- |-
        POR XMM0, XMM9               ; xmm0: R12|R13|R14|R6|R7|R8|R9|R10|R11|R0|R1|R2|R3|R4|R5|-
        PSHUFB XMM0, [invertir]      ; xmm0: R0|R1|R2|R3|R4|R5|R6|R7|R8|R9|R10|R11|R12|R13|R14|-
    POP RBP
    RET
```

5.4.2. temp_aux:

En la primer parte de ésta función auxiliar usamos máscaras para separar los colores y sumarlos entre si, dos pixeles al mismo tiempo. Luego armamos para dividir este valor por tres, dando el promedio.

```
temp_aux:
    PUSH RBP
    MOV RBP, RSP
    PUSH R12
    PUSH R13

    XORPD XMM2, XMM2
    XORPD XMM3, XMM3
    XORPD XMM4, XMM4

    MOVDQA XMM7, [pixel0BGR]
    PSHUFB XMM0, XMM7   ; ordeno en el registro los pixels de forma 0RGB (tengo 4 pixeles)
    MOVDQU XMM2, XMM0 ; XMM2: R|0|G|0|B|0|0|0|R|0|G|0|B|0|0|0
    PSHUFB XMM2, [soloPrimero] ; XMM2: R|0|0|0|0|0|0|0|R|0|0|0|0|0|0|0
    MOVDQU XMM3, XMM0 ; XMM3: R|0|G|0|B|0|0|0|R|0|G|0|B|0|0|0
    PSRLQ XMM3, 16 ; XMM3: G|0|B|0|0|0|0|0|G|0|B|0|0|0|0|0
    PSHUFB XMM3, [soloPrimero] ; XMM3: G|0|0|0|0|0|0|0|G|0|0|0|0|0|0|0
    MOVDQU XMM4, XMM0 ; XMM4: R|0|G|0|B|0|0|0|R|0|G|0|B|0|0|0
    PSRLQ XMM4, 32 ; XMM4: B|0|0|0|0|0|0|0|B|0|0|0|0|0|0|0
        ; Ahora podemos hacer suma
    PADDQ XMM2, XMM3
    PADDQ XMM2, XMM4 ; XMM2: qword[R+G+B] qword[R+G+B]
    XORPD XMM5, XMM5
        ; suma / 3
    MOV R12, 3
    MOV R13, 1
    XORPD XMM0, XMM0
    XORPD XMM1, XMM1
    CVTSI2SS XMM0, R12
    CVTSI2SS XMM1, R13
        ; PARA EXTENDER EL 3 A LOS PACKS DEL XMM5
    XORPD XMM5, XMM5
    ADDSS XMM5, XMM1
    PSLLDQ XMM5, 4
    ADDSS XMM5, XMM0
    PSLLDQ XMM5, 4
    ADDSS XMM5, XMM1
    PSLLDQ XMM5, 4
    ADDSS XMM5, XMM0
    CVTDQ2PS XMM2, XMM2
    DIVPS XMM2, XMM5 ; XMM2: prom (float)
    CVTTPS2DQ XMM2, XMM2
    CVTDQ2PS XMM2, XMM2
    MOVDQU XMM12, XMM2 ; XMM12: prom (float)
    CVTTPS2DQ XMM12, XMM12
        ;XMM2: [T|0|0|0|0|0|0|0] [T|0|0|0|0|0|0|0]
```

En esta segunda parte se elije mediante máscaras qué pixel será devuelto.

```
    XORPD XMM0, XMM0
        ; ponemos colores[0] en su lugar correspondiente {128 + 4t, 0, 0}

    MOVDQU XMM10, [colores0]  ; XMM10: qword[128(+4T)] qword[128(+4T)]

        ;XMM10: colores[0] nos falta agregar 4T
        ;XMM2: [T|0|0|0|0|0|0|0] [T|0|0|0|0|0|0|0]
    MOVDQU XMM6, XMM2
        ;XMM6: [T|0|0|0|0|0|0|0] [T|0|0|0|0|0|0|0]

        ;PARA MULTIPLICAR POR 4
    MOV R12, 4
    XORPD XMM14, XMM14
    MOVQ XMM14, R12
    XORPD XMM5, XMM5
    PADDQ XMM5, XMM14
    PSLLDQ XMM5, 8
    PADDQ XMM5, XMM14

    CVTDQ2PS XMM5, XMM5
    MULPS XMM6, XMM5 ; XMM6: qword[4T|0|0|0|0|0|0|0] qword[4T|0|0|0|0|0|0|0]

    CVTTPS2DQ XMM6, XMM6

        ; A: 4T + 128
        ; XMM6:  qword[4T|0|0|0|0|0|0|0]  qword[4T|0|0|0|0|0|0|0]
        ; XMM10:  qword[128(+4T)|0|0|0|0|0|0|0]  qword[128(+4T)|0|0|0|0|0|0|0]
    PADDUSW XMM10, XMM6
    MOVDQU XMM6, XMM10
        ;XMM10 == XMM6 == qword[4T+128] qword[4T+128]
    XORPD XMM15, XMM15
    PUNPCKLQDQ XMM10, XMM15
    PUNPCKHQDQ XMM6, XMM15
        ;XMM10 == qword{[A|0|0|0|0|0|0|0] [0|0|0|0|0|0|0|0]}
        ;XMM6 ==  qword{[A|0|0|0|0|0|0|0] [0|0|0|0|0|0|0|0]}
        ;Lo voy a mirar de otra forma para empaquetar (puedo porque con 16bits
                                    ;me alcanza para representar A):
        ;XMM10 == dword{[A|0] [G|0] [B|0] [0|0] [0|0] [0|0] [0|0] [0|0]}
        ;XMM6 ==  dword{[A|0] [G|0] [B|0] [0|0] [0|0] [0|0] [0|0] [0|0]}
    PACKUSWB XMM10, XMM6
        ;XMM10 == dword{[AGB0][0000][AGB0][0000]}
        ;empaquete saturando word a byte para reducir el tamanio de A de 2 bytes a 1 byte
    PSHUFB XMM10, [shuffleCaverna2]
        ;XMM10 == dword{[AGB0][AGB0][0000][0000]}

    MOVDQU XMM1, [tresDos]
    PCMPGTD XMM1, XMM12
    PSHUFB XMM1, [shuffleCaverna2]
    PAND XMM1 , XMM10  ; estos pixeles le ponemos colores[0]
    XORPD XMM0, XMM0
    POR XMM0, XMM1 ; ponemos colores0 en donde va en dst
;////////////////////////////////////////////////////////////////////////////

        ; ponemos colores[4] en su lugar correspondiente {  0, 0, 1151 -4t}

        ;Codigo no copiado. Se repite el proceso que con [colores0]
```

```
;////////////////////////////////////////////////////////////////////////////

        ; ponemos colores[3] en su lugar correspondiente     {0, 895 - 4t, 255}

        ;Codigo no copiado. Se repite el proceso que con [colores0]
;////////////////////////////////////////////////////////////////////////////
        ; ponemos colores[2] en su lugar correspondiente {639 -4t, 255, -384 + 4t},

        ;Codigo no copiado. Se repite el proceso que con [colores0]
;////////////////////////////////////////////////////////////////////////////

        ; ponemos colores[1] en su lugar correspondiente {255, -128 + 4t, 0}

        ;Codigo no copiado. Se repite el proceso que con [colores0]
;////////////////////////////////////////////////////////////////////////////


        ;ahora en XMM2 tenemos dst con los colores finales
        ;sacamos los ceros para que queden pixeles de 3 bytes


    MOVDQU XMM10, [pixelFinal] ; [ 0 , 1 , 2 , 4, 5 , 6 , - , -, - , - , - , -, - , - , - , -]
    PSHUFB XMM0, XMM10

    POP R13
    POP R12
    POP RBP
    RET
```

5.5. Resultado de los experimentos

Experimento 1

6. Filtro LDR

6.1. Enunciado

Filtro *LDR*

Programar el filtro *LDR* en lenguaje C y en ASM haciendo uso de las instrucciones **SSE**.

Experimento 1

Analizar cuales son las diferencias de performace entre las versiones de C y ASM. Realizar gráficos que representen estas diferencias.

6.2. Análisis previo

6.3. Implementación en C

Titulo del parrafo Bla bla bla bla. Esto se muestra en la figura **??**.

6.4. Implementación en assembler

```
struct Pepe {

    ...

};
```

6.5. Resultado de los experimentos

7. Conclusiones y trabajo futuro

asdf4