DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

Procesamiento de imagenes mediante instrucciones SIMD

Organización del computador 2
Primer Cuatrimestre de 2017

Grupo Estrellitas

Integrante	LU	Correo electrónico
Hofmann, Federico	745/14	federico2102@gmail.com
Lasso, Andrés	714/14	lassoandres2@gmail.com
Berríos Verboven, Nicolas	46/12	nbverboven@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Índice

Parte I
Introducción

En el siguiente trabajo se buscó conseguir un primer acercamiento a las instrucciones SIMD, las cuales se utilizan en el procesamiento de datos de manera simultanea. Para ello implementamos distintos filtros para imágenes tanto en el lenguaje de programación C como en Assembler. Una vez implementados procedimos a comparar ambas versiones de cada filtro buscando así poder conocer mejor las ventajas y desventajas de cada una.
Ademas, realizamos una serie de experimentos con cada filtro con el objetivo de realizar un análisis lo mas completo posible para de esta manera lograr justificar por que una implementación funciona mejor que la otra.
Por ultimo, intentaremos exponer todos los datos de la manera mas detallada posible, describiendo paso a paso los algoritmos implementados, explicando de forma completa la experimentación realizada y mencionando siempre todos los detalles que consideremos relevantes para el desarrollo de este trabajo y la comprensión del mismo por parte del lector.

Parte II
Implementaciones

Las imágenes utilizadas en este trabajo están en formato BMP y cada píxel de las mismas esta compuesto por 4 componentes de 1 byte cada una, ocupando así 4 bytes cada píxel. Dichas componentes son A, B, G y R, representando la componente de transparencia, el color azul, el verde y el rojo respectivamente. Cada componente esta representada por un entero sin signo, con lo cual sus posibles valores están entre 0 y 255. El ancho de las imágenes es siempre superior a 16 píxeles y múltiplo de 4 píxeles. En ningún caso se alterará el valor de la componente A. Las imágenes son consideradas como matrices.

1. Convertir RGB a YUV y YUV a RGB

Este filtro realiza una transformación lineal de las componentes RGB, llevándolas así a un nuevo espacio donde las componentes obtenidas serán llamadas YUV.

De forma análoga, al convertir YUV a RGB se realiza el procedimiento inverso para volver al subespacio anterior.

En ambos casos la expresión se debe realizar algún tipo de saturación para que el resultado de la operación se encuentre entre 0 y 255. La manera en la que decidimos lograr esto fue quedándonos con el mínimo entre el resultado de la transformación lineal y 255, y luego buscando el máximo entre dicho resultado y 0.

Lo que se hace entonces es ir recorriendo toda la imagen componente por componente, aplicándoles su transformación lineal correspondiente y escribiendo dichas componentes en la imagen destino; imagen que obtenemos como resultado.

1.1. Pseudo código del algoritmo en C

RGB2YUV:

```
1: j = 0
2: while j < cantPixeles do
3:     R = pixel[j].R
4:     G = pixel[j].G
5:     B = pixel[j].B
6:     pixel[j].Y = Sature(((66 * R + 129 * G + 25 * B + 128) >> 8) + 16)
7:     pixel[j].U = Sature(((−38 * R − 74 * G + 112 * B + 128) >> 8) + 128)
8:     pixel[j].V = Sature(((112 * R − 94 * G − 18 * B + 128) >> 8) + 128)
9:     j + +
10: end while
```

YUV2RGB:

```
1: j = 0
2: while j < cantPixeles do
3:     Y = pixel[j].Y
4:     U = pixel[j].U
5:     V = pixel[j].V
6:     pixel[j].R = Sature((298 * (Y − 16) + 409 * (V − 128) + 128) >> 8)
7:     pixel[j].G = Sature((298 * (Y − 16) − 100 * (U − 128) − 208 * (V − 128) + 128) >> 8)
8:     pixel[j].B = Sature((298 * (Y − 16) + 516 * (U − 128) + 128) >> 8)
9:     j + +
10: end while
```

1.2. Descripción de implementación en assembler

Para la implementación del filtro convert RGB to YUV en este lenguaje decidimos procesar de a 2 píxeles a la vez mediante la utilización de las instrucciones del modelo de procesamiento SIMD. Los registros en este modelo son de 128 bits, con lo cual dos píxeles ocupan solo la mitad. La decisión de trabajar solo de a dos píxeles a la vez partió del hecho de que al realizar las operaciones de transformación lineal a cada componente de cada píxel se precisa de al menos 16 bits por componente para evitar perdida de información, dado que cada componente puede tomar como máximo el valor 255 (máximo posible en 8 bits), y al sumarlo y multiplicarlo con otros valores 16 bits serán suficientes para almacenar estos resultados.

Para realizar las operaciones de la forma mencionada debimos proceder a desempaquetar los dos píxeles de bytes a words, utilizando así los 128 bits del registro XMM ocupando 16 bits cada componente (los 8 bits de la parte alta de cada componente se extienden con ceros).

(ACA VA UN DIBUJO DEL MOVQ DE DOS BYTES A LA PARTE BAJA DEL XMM Y OTRO DIBUJO QUE MUESTRE EL DESEMPAQUETADO, O UN SOLO DIBUJO QUE LOS MUESTRE YA DESEMPAQUETADOS)

Una vez hecho esto, procedimos a copiar los datos a dos registros distintos del original para luego shiftearlos con la instrucción `PSRLDQ` dejando así a las componentes R, G y B en la misma parte de cada registro y luego asilamos en cada uno a las tres componentes mediante la utilización de mascaras. De este modo un registro contendrá solo dos componentes R en ciertas posiciones del mismo, otro contendrá solo a G en las mismas posiciones que el anterior y lo mismo con el tercer registro que solo contiene a B.

(ACA VA UNA IMAGEN QUE MUESTRE COMO SE SHIFTEARON LOS TRES REGISTROS Y LUEGO COMO SE AISLAN R, G Y B CON LA APLICACION DE MASCARAS)

El siguiente paso consistió en realizar la transformación lineal a cada componente realizando multiplicaciones entre los registros y mascaras con valores constantes y sumando registros entre si.

(PONER ACA DIBUJOS QUE MUESTREN COMO SE MULTIPLICO A LOS REGISTROS POR MASCARAS Y SE LOS SUMARON ENTRE SI PARA CONSEGUIR LA TL BUSCADA. NO HACE FALTA PONER COMO CAMBIA CADA COMPONENTE, SE PUEDE MOSTRAR COMO CAMBIA UNA SOLA Y LOS OTROS DOS CASOS SON ANALOGOS)

Por ultimo, luego de realizar todas las operaciones anteriores ya tenemos tres registros distintos donde cada uno tiene a las componentes R, G y B con la transformación lineal ya aplicada, son lo cual a partir de este momento las llamamos Y, U y V respectivamente. El ultimo paso consiste entonces en empaquetar a los tres registros de word a byte con la instrucción `PACKUSWB`, la cual realiza saturación sin signo, es decir que si los valores superaban 255 o eran menores a 0, entonces pasaban a ser 255 o 0. Después debimos volver a shiftear los registros pero esta vez hacia el lado opuesto al que se los shifteo antes, con la instrucción `PSLLDQ` para de esta forma acomodar a las componentes Y, U y V en las partes que les corresponde en cada registro. Después se juntó a las tres componentes en un solo registro con la instrucción `XORPS` y finalmente se copio la parte baja del registro resultante a su posición correspondiente en la imagen destino.

(ACA VAN DUBUJOS QUE MUESTRAN COMO SE EMPAQUETAN LOS REGISTROS NUEVAMENTE, DESPUES SE SHIFTEAN, SE HACE XORPS PARA JUNTARLOS EN UNO SOLO Y SE COPIAN A LA IMAGEN DESTINO)

Luego de realizar todos estos pasos se vuelve al principio y se realizan los mismos pasaos con los dos píxeles siguientes y así hasta haber recorrido toda la imagen.

1.3. Comparación con implementación de C

2. Four Combine

El filtro Four Combine consiste en reorganizar los píxeles de la imagen original obtenida como parámetro buscando conseguir como resultado 4 imágenes idénticas a la original pero mas pequeñas, de forma que sumando los tamaños de las 4 (las 4 del mismo tamaño) se obtenga el tamaño de la original.

Los píxeles se reorganizan de la siguiente manera, donde i recorre las filas, j las columnas y ambas comienzan en la posicion 0:

1. Buscamos los píxeles en las posiciones (i,j), (i, j+1), (i+1,j), (i+1.j+1).

2. Los copiamos en la imagen destino en las posiciones (i,j), (i, j+(Ancho/2)), (i+(Largo/2),j), (i+(Largo/2), j+(Ancho/2)).

3. i pasa a valer i+2, j pasa a valer j+2 y se repiten los pasos 1 y 2 hasta que se termine de recorrer toda la imagen.

Los píxeles no son modificados en ningún momento.

2.1. Pseudo código del algoritmo en C

FourCombine:

```
1: i = 0
2: while i < cantFilas do
3:     j = 0
4:     while j < cantColumnas do
5:         pixel[i][j].dst = pixel[i][j].src
6:         pixel[i][j + cantColumnas/2].dst = pixel[i][j + 1].src
7:         pixel[i + cantFilas/2][j].dst = pixel[i + 1][j].src
8:         pixel[i + cantFilas/2][j + cantColumnas/2].dst = pixel[i + 1][j + 1].src
9:         j = j + 2
10:    end while
11:    i = i + 2
12: end while
```

2.2. Descripción de implementación en assembler

2.3. Comparación con implementación de C

3. Máximo Cercano

Máximo Cercano busca el máximo de cada componente en una matriz de 7x7 píxeles donde el píxel P se encuentra justo al centro y luego realiza una serie de operaciones entre el nuevo píxel obtenido (donde las componentes R, G y B son las mayores en la matriz de 7x7 mencionada previamente), y sobre el píxel P. Dichas operaciones consiguen generar un nuevo píxel que contenga un porcentaje del píxel original (P) y otro porcentaje del máximo cercano. El porcentaje (α) sera representado con un numero de precisión simple (float) entre 0 y 1 y sera un parámetro de entrada de la función implementada. Debemos evitar perder precisión en los cálculos. La cuenta realizada es:
dst[i,i] = src[i,i]*(1-α) + max*(α).
Donde dst es la imagen destino y src es la imagen fuente. Esta operación se realiza con cada una de las componentes de cada píxel.

El margen de 3 píxeles de la imagen es pintado de blanco, es decir que las componentes de dichos píxeles tendrán el valor 255. De este modo no hay que preocuparse por los bordes al momento de armar la matriz de 7x7 al rededor de cada píxel a modificar.

3.1. Pseudo código del algoritmo en C

MaxCloser:

```
1:  i = 0
2:  while i < cantFilas do
3:      j = 0
4:      while j < cantColumnas do
5:          if distanciaAlBorde < 3 then
6:              pixel[i][j].dst = 255
7:          else
8:              k = −3
9:              while k < 3 do
10:                 l = −3
11:                 while l < 3 do
12:                     maxR = max(maxR, pixel[i + k][j + l].R)
13:                     maxG = max(maxG, pixel[i + k][j + l].G)
14:                     maxB = max(maxB, pixel[i + k][j + l].B)
15:                     l + +
16:                 end while
17:                 k + +
18:             end while
19:             pixel[i][j].dst = pixel[i][j].src ∗ (1 − α) + maxRGB ∗ α
20:         end if
21:         j + +
22:     end while
23:     i + +
24: end while
```

3.2. Descripción de implementación en assembler

3.3. Comparación con implementación de C

4. Zoom Lineal

4.1. Pseudo código del algoritmo en C

4.2. Descripción de implementación en assembler

4.3. Comparación con implementación de C

Parte III
Discusión

Parte IV
Conclusiones