Licenciatura en Sistema

TP SIMD

Organización del Computador II

Abregu, Pablo – Yagusz, Salvador – Santillán, Javier
4-6-2020

Introducción

Directorio:

/src: Localización de los archivos del programa

/images: Contiene las imágenes que el programa utiliza

Limpieza de Directorio

El archivo *clear.sh* se encarga de limpiar el directorio, eliminando los archivos generados por el programa (*.o, *.bmp, *.csv)

Ejecución

El archivo *run.sh* se encarga de la ejecución del programa.

Prueba de Ejecución:

$ sh run.sh

```
pablo@pablo-VirtualBox:~/Escritorio/SIMD/src$ sh run.sh
Organización del Computador 2.
Trabajo Práctico Nro. 2
Programa para procesamiento de imágenes BMP.

######## Filtros ########

1- Blanco y negro
2- Aclarar
3- Median Filter
4- Blend
5- BlendSIMD
6- Negativo
7- Escala de Grises
0- Salir

>>
```

Al final de cada operación de filtro, se abrirá la imagen generada.

1. Filtro Aclarar

Se aplica el filtro para aclarar una imagen.
Para este filtro le suma una determinada cantidad a cada canal del pixel, siendo RGB los valores R(Red), G(Green), B(Blue):

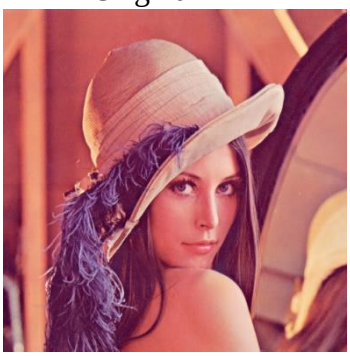
RGB = (R + n, G + n, B + n)

La función recibe como parámetro una imagen a aclarar, y un valor n, siendo este el nivel de aclaración.
Código:

```
63  void aclarar(BMPDATA *bmpData, int n)
64  {
65
66      for (int i = 0; i < cantPixels(bmpData); i++)
67      {
68          unsigned char r = bmpData->red[i];
69          unsigned char g = bmpData->green[i];
70          unsigned char b = bmpData->blue[i];
71          bmpData->red[i] = min(r + n, 255);
72          bmpData->green[i] = min(g + n, 255);
73          bmpData->blue[i] = min(b + n, 255);
74      }
75  }
```

Prueba:

Original



Modificada

2. Median Filter

Se aplica el filtro de mediana.
Este consiste en aplacar el ruido de una imagen, reparando cada pixel.
En este caso se agarra una ventana de 3x3, que va recorriendo toda la imagen. Esos 9 pixeles, obtenidos en la ventana, se ordenan por cada canal del pixel RGB, dividiéndolo en 3 listas uno para cada canal del pixel.

```
typedef struct
{
    unsigned char r;
```

```
    unsigned char g;
    unsigned char b;
} Pixel;

typedef Pixel t_Window[9];
```

Luego se procede a ordenar cada lista de canales. y se obtiene la media del ordenamiento de la ventana. Por último, al pixel inicial de la ventana seleccionada sin ordenar se le asignan los valores de mediana de cada uno de los tres canales.

```
bmpData->red[(m - 1) * N + (n - 1)]   = window[4].r;

bmpData->green[(m - 1) * N + (n - 1)] = window[4].g;

bmpData->blue[(m - 1) * N + (n - 1)]  = window[4].b;
```

Codigo:

```
77  void medianFilter(BMPDATA *bmpData)
78  {
79      long N = bmpData->infoHeader.biWidth;
80      long M = bmpData->infoHeader.biHeight;
81
82      //   Recorremos los pixeles
83      for (int m = 1; m < M - 1; ++m)
84      { // Alto
85          for (int n = 1; n < N - 1; ++n)
86          { // Ancho
87              int k = 0;
88              t_Window window; //Inicializamos ventana de pixeles 3x3 ;
89              for (int j = m - 1; j < m + 2; ++j)
90              {
91                  for (int i = n - 1; i < n + 2; ++i)
92                  {
93                      window[k].b = bmpData->blue[j * N + i];  //
94                      window[k].g = bmpData->green[j * N + i]; //  Carga los pixeles en la ventana 3x3
95                      window[k].r = bmpData->red[j * N + i];   //
96                      k++;
97                      //   Ordenamos los elementos  azules
98                      for (int j = 0; j < 5; ++j)
99                      {
100                         int min = j;
101                         for (int l = j + 1; l < 9; ++l)
102                         {
103                             int aux, aux2;
104                             aux = (int)window[l].b;
105                             aux2 = (int)window[min].b;
106                             if (aux < aux2)
107                             {
108                                 min = l;
109                             }
110                         }
111                         const unsigned char temp = window[j].b;
112                         window[j].b = window[min].b;
113                         window[min].b = temp;
114                     }
```

```
115                     //   Modificamos la componente azul del pixel de la imagen
116                     bmpData->blue[(m - 1) * N + (n - 1)] = window[4].b;
117                     //   Ordenamos los elementos verdes
118                     for (int j = 0; j < 5; ++j)
119                     {
120                         int min = j;
121                         for (int l = j + 1; l < 9; ++l)
122                         {
123                             int aux, aux2;
124                             aux = (int)window[l].g;
125                             aux2 = (int)window[min].g;
126                             if (aux < aux2)
127                             {
128                                 min = l;
129                             }
130                         }
131                         const unsigned char temp = window[j].g;
132                         window[j].g = window[min].g;
133                         window[min].g = temp;
134                     }
135                     //   Modificamos la componente verde del pixel de la imagen
136                     bmpData->green[(m - 1) * N + (n - 1)] = window[4].g;
```

```
137                     //   Ordenamos los elementos rojos
138                     for (int j = 0; j < 5; ++j)
139                     {
140                         int min = j;
141                         for (int l = j + 1; l < 9; ++l)
142                         {
143                             int aux, aux2;
144                             aux = (int)window[l].r;
145                             aux2 = (int)window[min].r;
146                             if (aux < aux2)
147                             {
148                                 min = l;
149                             }
150                         }
151                         const unsigned char temp = window[j].r;
152                         window[j].r = window[min].r;
153                         window[min].r = temp;
154                     }
155                     //   Modificamos la componente roja del pixel de la imagen
156                     bmpData->red[(m - 1) * N + (n - 1)] = window[4].r;
157                 }
158             }
159         }
160     }
161 }
162
```

Prueba:

Imagen Corrompida

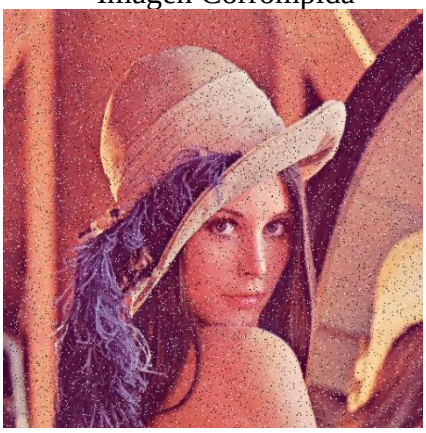
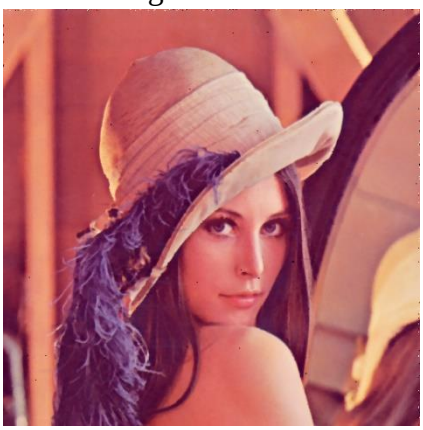


Imagen Sin Ruido



3. Blend

Se aplica el filtro Blend.
Este filtro consiste en superponer dos imágenes, multiplicando los canales de cada pixel de una imagen con la otra, generando una nueva.

La multiplicación del los valores del pixel se hace mediante código assembler (multiply.s)

P1(r, g, b) * P2(r, g, b) = P3(r1*r2, g1*g2, b1*b2)

Codigo:

La función recibe dos imágenes para superponer.

```
162  void multiplyBlend(BMPDATA *bmpOne, BMPDATA *bmpTwo)
163  {
164      for (int i = 0; i < cantPixels(bmpOne); i++)
165      {
166          bmpOne->red[i]   = asmMultiply(bmpTwo->red[i], bmpOne->red[i]);
167          bmpOne->green[i] = asmMultiply(bmpTwo->green[i], bmpOne->green[i]);
168          bmpOne->blue[i]  = asmMultiply(bmpTwo->blue[i], bmpOne->blue[i]);
169      }
170  }
```

Multply.s

```
1   section .data
2
3   section .text
4       global asmMultiply  ;multiply(int a, int b)
5
6   asmMultiply:
7       push    ebp         ;enter 0,0
8       mov     ebp, esp    ;enter 0,0
9
10      mov     eax, [EBP + 8]  ;valor a
11      mov     ecx, [EBP + 12] ;valor b
12
13      mul     ecx
14
15      mov     edx, 0      ;(edx:eax)
16      mov     ecx, 255    ;para dividir por 255
17
18      div     ecx
19
20      mov     ebp, esp    ;reinicio el EBP a su posicion de partida
21      pop     ebp         ;restauro el EBP0 viejo
22
23      ret
24
```

Prueba:

Imagen A

Imagen B

Imagen generada

4. Blend MMX

Se aplica el filtro Blend.
Este filtro consiste en superponer dos imágenes, multiplicando los canales de cada pixel de una imagen con la otra, generando una nueva.
A diferencia del anterior filtro, este funciona con SIMD, utilizando instrucciones MMX.
Lo cual, aumentaría la eficacia en el tiempo de ejecución.

5. Filtro Negativo

Se aplica el filtro Negativo.

Este filtro consiste en la resta de 255 para cada uno de los canales del pixel.

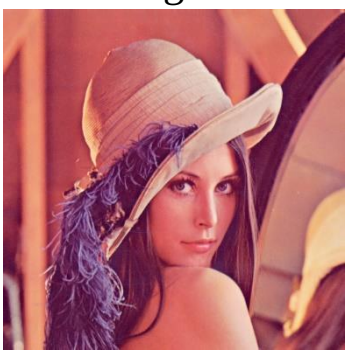
P1(r, g, b) = P (255 – r, 255 – g, 255 – b)

Codigo:

```
221  void negativo(BMPDATA *bmpData)
222  {
223      for (int i = 0; i < cantPixels(bmpData); i++)
224      {
225          bmpData->red[i]   = 255 - bmpData->red[i];
226          bmpData->green[i] = 255 - bmpData->green[i];
227          bmpData->blue[i]  = 255 - bmpData->blue[i];
228      }
229  }
230
```

Prueba:

Original



Negativo

6. Filtro Escala de Grises

Se aplica el filtro de Escala de Grises.

Este filtro consiste en calcular la suma de cada uno de los canales del pixel, y luego dividiéndolo por 3.

P (r, g, b) = P ((r+g+b) / 3, (r+g+b) / 3, (r+g+b) / 3**)**

Codigo:

La función recibe una imagen por parámetro, y en cada del pixel se reemplaza por la suman de cada uno de los canales, dividiéndolo por 3.

```
231  void escalaDeGrises(BMPDATA *bmpData)
232  {
233      for (int i = 0; i < cantPixels(bmpData); i++)
234      {
235          int media = calcularMedia(bmpData->red[i], bmpData->green[i], bmpData->blue[i]);
236          bmpData->red[i] = media;
237          bmpData->green[i] = media;
238          bmpData->blue[i] = media;
239      }
240  }
241
242  int calcularMedia(int r, int g, int b)
243  {
244      int aux = r + g + b;
245      return aux / 3;
246  }
```

Prueba:

Original

Modificada

Repositorio en Git: https://github.com/pabloabregu/SIMD.git