Performance Lab

교 과 목 : 고급시스템프로그래밍
교 수 님 : 안준선 교수님
학 과 : 항공전자정보공학부
학 번 : 2016124087
이 름 : 김현용
제 출 일 자 : 2021.05.16

1. Rotate

1) 기존의 코드

```
void naive_rotate(int dim, pixel *src, pixel *dst)
{
    int i, j;

    for (i = 0; i < dim; i++)
        for (j = 0; j < dim; j++)
            dst[RIDX(dim-1-j, i, dim)] = src[RIDX(i, j, dim)];
}
```

위 기존의 코드는 이 중 for문 ij loop으로 단순히 모든 픽셀에 대해서 하나 하나 위치를 옮겨주는 것이다.

2) 수정 코드

```
void rotate(int dim, pixel *src, pixel *dst)
{
    int i, j, k;

    for (i = 0; i < dim; i += 32) { //Loop Unrolling
        for (j = 0; j < dim; j++) {
            for (k = 0; k < 32; k++) {
                //k는 Loop Unrolling시 발생하는 i 간격의 계산 반복 (32일 때 최대 효율)
                //dst i,j를 바꾸고, i를 N-1-i로 변경
                dst[RIDX(dim-1-j, i+k, dim)] = src[RIDX(i+k, j, dim)];
            }
        }
    }
}
```

위에서 RIDX(i,j,N)는 i * N + j를 수행하는 것이기 때문에 곱하기 연산 시 3 cycle이 필요하고, 그 동안 CPU가 쉬게 된다. 그렇기 때문에 Loop Unrolling을 적용시켜 i를 32씩 증가시키고, k for문을 추가해 32개의 연산을 한 번에 하는 것을 구현했다.

3) 실행 결과

```
Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             512     1024    2048    Mean
Your CPEs       9.3     10.8    21.8
Baseline CPEs   9.3     10.8    21.8
Speedup         1.0     1.0     1.0     1.0

Rotate: Version = rotate: Current working version:
Dim             512     1024    2048    Mean
Your CPEs       2.0     3.0     4.3
Baseline CPEs   9.3     10.8    21.8
Speedup         4.6     3.6     5.1     4.4
```

1. Smooth

1) 기존의 코드

```
void naive_smooth(int dim, pixel *src, pixel *dst)
{
    int i, j;

    for (i = 0; i < dim; i++)
        for (j = 0; j < dim; j++)
            dst[RIDX(i, j, dim)] = avg(dim, i, j, src);
}
```

기존의 코드는 단순히 모든 픽셀에 대해 avg 함수를 적용한 것인데, 여기서 avg 함수는 다음과 같다.

```
static void accumulate_sum(pixel_sum *sum, pixel p)
{
    sum->red += (int) p.red;
    sum->green += (int) p.green;
    sum->blue += (int) p.blue;
    sum->num++;
    return;
}

/*
 * assign_sum_to_pixel - Computes averaged pixel value in current_pixel
 */
static void assign_sum_to_pixel(pixel *current_pixel, pixel_sum sum)
{
    current_pixel->red = (unsigned short) (sum.red/sum.num);
    current_pixel->green = (unsigned short) (sum.green/sum.num);
    current_pixel->blue = (unsigned short) (sum.blue/sum.num);
    return;
}

/*
 * avg - Returns averaged pixel value at (i,j)
 */
static pixel avg(int dim, int i, int j, pixel *src)
{
    int ii, jj;
    pixel_sum sum;
    pixel current_pixel;

    initialize_pixel_sum(&sum);
    for(ii = max(i-1, 0); ii <= min(i+1, dim-1); ii++)
        for(jj = max(j-1, 0); jj <= min(j+1, dim-1); jj++)
            accumulate_sum(&sum, src[RIDX(ii, jj, dim)]);

    assign_sum_to_pixel(&current_pixel, sum);
    return current_pixel;
}
```

avg 함수는 이중 for문으로 accumulate_sum을 수행하는 것이고, 이 때, for문은 모서리, 사이드 라인 처리를 위해 max 함수를 쓴 것을 알 수 있다. accumulate_sum은 구조체 sum의 r,g,b 값에 해당 픽셀 값을 쌓는 것을 알 수 있고, assign_sum_to_pixel 함수에서 평균 값을 구해서 할당하는 것을 알 수 있다.

2) 수정 코드

```
void smooth(int dim, pixel *src, pixel *dst)
{
    int i, j, tmp, tmpi;

    /*코너 처리 */
    //왼쪽 위(0,0)
    dst[0].red = (src[0].red + src[1].red + src[dim].red + src[dim + 1].red) >> 2;
    dst[0].blue = (src[0].blue + src[1].blue + src[dim].blue + src[dim + 1].blue) >> 2;
    dst[0].green = (src[0].green + src[1].green + src[dim].green + src[dim + 1].green) >> 2;

    //오른쪽 위(0,N)
    i = dim*2-1;
    dst[dim - 1].red = (src[dim - 1].red + src[dim - 2].red + src[i-1].red + src[i].red) >> 2;
    dst[dim - 1].blue = (src[dim - 1].blue + src[dim - 2].blue + src[i-1].blue + src[i].blue) >> 2;
    dst[dim - 1].green = (src[dim - 1].green + src[dim - 2].green + src[i-1].green + src[i].green) >> 2;

    //왼쪽 아래(N,0)
    j = dim * (dim - 1);
    i = dim * (dim - 2);
    dst[j].red = (src[j].red + src[j + 1].red + src[i].red + src[i + 1].red) >> 2;
    dst[j].blue = (src[j].blue + src[j + 1].blue + src[i].blue + src[i + 1].blue) >> 2;
    dst[j].green = (src[j].green + src[j + 1].green + src[i].green + src[i + 1].green) >> 2;

    //오른쪽 아래(N,N)
    j = dim * dim -1;
    i = dim * (dim - 1) - 1;
    dst[j].red = (src[j - 1].red + src[j].red + src[i - 1].red + src[i].red) >> 2;
    dst[j].blue = (src[j - 1].blue + src[j].blue + src[i - 1].blue + src[i].blue) >> 2;
    dst[j].green = (src[j - 1].green + src[j].green + src[i - 1].green + src[i].green) >> 2;
```

```
/*사이드라인 처리*/
//위쪽 줄(0,1 ~ 0,N-1)
for (j = 1; j < dim - 1; j++)
{
    dst[j].red = (src[j].red + src[j - 1].red + src[j + 1].red +
                  src[j + dim].red + src[j + 1 + dim].red + src[j - 1 + dim].red) / 6;
    dst[j].blue = (src[j].blue + src[j - 1].blue + src[j + 1].blue +
                   src[j + dim].blue + src[j + 1 + dim].blue + src[j - 1 + dim].blue) / 6;
    dst[j].green = (src[j].green + src[j - 1].green + src[j + 1].green +
                    src[j + dim].green + src[j + 1 + dim].green + src[j - 1 + dim].green) / 6;
}

//아래 쪽 줄 (N,1 ~ N,N-1)
for (j = dim * (dim - 1) + 1; j < dim * dim - 1; j++)
{
    dst[j].red = (src[j].red + src[j - 1].red + src[j + 1].red +
                  src[j - dim].red + src[j + 1 - dim].red + src[j - 1 - dim].red) / 6;
    dst[j].blue = (src[j].blue + src[j - 1].blue + src[j + 1].blue +
                   src[j - dim].blue + src[j + 1 - dim].blue + src[j - 1 - dim].blue) / 6;
    dst[j].green = (src[j].green + src[j - 1].green + src[j + 1].green +
                    src[j - dim].green + src[j + 1 - dim].green + src[j - 1 - dim].green) / 6;
}

//왼쪽 줄 (1,0 ~ N-1,0)
for (j = dim; j < dim * (dim - 1); j += dim)
{
    dst[j].red = (src[j].red + src[j - dim].red + src[j + 1].red +
                  src[j + dim].red + src[j + 1 + dim].red + src[j - dim + 1].red) / 6;
    dst[j].blue = (src[j].blue + src[j - dim].blue + src[j + 1].blue +
                   src[j + dim].blue + src[j + 1 + dim].blue + src[j - dim + 1].blue) / 6;
    dst[j].green = (src[j].green + src[j - dim].green + src[j + 1].green +
                    src[j + dim].green + src[j + 1 + dim].green + src[j - dim + 1].green) / 6;
}

//오른쪽 줄 (1,N ~ N-1,N)
for (j = dim + dim - 1; j < dim * dim - 1; j += dim)
{
    dst[j].red = (src[j].red + src[j - 1].red + src[j - dim].red +
                  src[j + dim].red + src[j - dim - 1].red + src[j - 1 + dim].red) / 6;
    dst[j].blue = (src[j].blue + src[j - 1].blue + src[j - dim].blue +
                   src[j + dim].blue + src[j - dim - 1].blue + src[j - 1 + dim].blue) / 6;
    dst[j].green = (src[j].green + src[j - 1].green + src[j - dim].green +
                    src[j + dim].green + src[j - dim - 1].green + src[j - 1 + dim].green) / 6;
}

  /* 나머지 경우 (중앙 픽셀 처리) (1,1 ~ N-1,N-1) */
  tmpi = dim;
  for (i = 1; i < dim - 1; i++)
  {
      for (j = 1; j < dim - 1; j++)
      {
              tmp = tmpi + j; //해당 행의 열
              dst[tmp].red = (src[tmp].red + src[tmp - 1].red + src[tmp + 1].red + src[tmp - dim].red + src[tmp - dim - 1].red +
                              src[tmp - dim + 1].red + src[tmp + dim].red + src[tmp + dim + 1].red + src[tmp + dim - 1].red) / 9;
              dst[tmp].green = (src[tmp].green + src[tmp - 1].green + src[tmp + 1].green + src[tmp - dim].green + src[tmp - dim - 1].green +
                              src[tmp - dim + 1].green + src[tmp + dim].green + src[tmp + dim + 1].green + src[tmp + dim - 1].green) / 9;
              dst[tmp].blue = (src[tmp].blue + src[tmp - 1].blue + src[tmp + 1].blue + src[tmp - dim].blue + src[tmp - dim - 1].blue +
                              src[tmp - dim + 1].blue + src[tmp + dim].blue + src[tmp + dim + 1].blue + src[tmp + dim - 1].blue) / 9;
      }
      tmpi += dim; // 다음 행
  }
}
```

수정한 코드는 위와 같은 데 예를 들어 모서리의 경우 기존의 코드에서는 0,0에 대해서 0,0 값의 r,g,b와 그 오른쪽. 오른쪽아래, 아래쪽의 4개의 r,g,b 값을 각각 for문을 통해 sum 구조체의 변수에 값을 쌓고, 할당 시 /4도 수행했지만, 수정한 코드는 Loop Unrolling 한 것과 비슷하게 그 계산 과정을 한 번에 계산했다. 따라서 모서리, 사이드라인, 그 나머지의 중앙 값을 나눠서 계산하여 for문 수를 줄이고, /연산의 Cycle 동안 다음을 미리 계산하는 Loop Unrolling 효과로 수행 시간이 줄어들었다고 할 수 있다.

3) 실행 결과

```
Smooth: Version = naive_smooth: Naive baseline implementation:
Dim             256     512     1024    Mean
Your CPEs       38.2    41.6    42.2
Baseline CPEs   38.2    41.6    42.2
Speedup         1.0     1.0     1.0     1.0

Smooth: Version = smooth: Current working version:
Dim             256     512     1024    Mean
Your CPEs       12.3    12.6    13.0
Baseline CPEs   38.2    41.6    42.2
Speedup         3.1     3.3     3.2     3.2
```

총 결과 (Rotate = 4.4배 증가, Smooth = 3.2배 증가)

```
Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             512     1024    2048    Mean
Your CPEs       9.3     10.8    21.8
Baseline CPEs   9.3     10.8    21.8
Speedup         1.0     1.0     1.0     1.0

Rotate: Version = rotate: Current working version:
Dim             512     1024    2048    Mean
Your CPEs       2.0     3.0     4.3
Baseline CPEs   9.3     10.8    21.8
Speedup         4.6     3.6     5.1     4.4

Smooth: Version = naive_smooth: Naive baseline implementation:
Dim             256     512     1024    Mean
Your CPEs       38.2    41.6    42.2
Baseline CPEs   38.2    41.6    42.2
Speedup         1.0     1.0     1.0     1.0

Smooth: Version = smooth: Current working version:
Dim             256     512     1024    Mean
Your CPEs       12.3    12.6    13.0
Baseline CPEs   38.2    41.6    42.2
Speedup         3.1     3.3     3.2     3.2

Summary of Your Best Scores:
  Rotate: 4.4 (rotate: Current working version)
  Smooth: 3.2 (smooth: Current working version)
```