标题

一、背景说明

说到需要使用向量化指令来释放CPU性能，那么就不可避免地会想到那些有密集计算的代码，而在我完成过的一些项目中，首先让我印象深刻的就是图像信息处理中的各种图片处理算法；

由于图片的处理需要以像素为单位扫描全图，且对象若为全色图需要处理多个色彩通道，因此在处理过程中会产生大量嵌套循环，因此下面我就以处理较为复杂的对数化增强为例探索向量化指令能对程序的性能有多大的提升。

本次实验中所用到的图片格式为3860*2140 24位全色图，使用编译器GCC-8.1.0编译，使用的是AVX指令集，为了排除编译器自动优化对结果产生的影响，每次测试都会使用四种优化等级进行统计结果。

二、探索过程

2.1 基础方法

为了增强图像的可视信息，对图像中的像素进行基于对数的操作 公式如下：

$$L_d = \frac{\log\left(L_w + 1\right)}{\log\left(L_{max} + 1\right)}$$

L_d是显示亮度（我们所要求的值），L_w是真实世界的亮度（图片的当前值），L_{max}是场景中的最亮值（图片的最大值）公式很简单，实现起来也比较容易，这个映射保证了不管场景的动态范围是怎样的，其最大值都能映射到 1 ，其他的值能够比较平滑的递增。

在实现层面上，我们需要对整张图片的所有像素点进行遍历，因此需要多次的嵌套循环，基础实现代码如下：

```
if( infoHeader.biBitCount == 24 ){
    // 读取真彩图中的rgb数据，将之存到一维数组中
    Byte *RGBdata_24 = NULL;
    RGBdata_24 = (Byte *)calloc( infoHeader.biHeight * real_width, 1 );
    fread( RGBdata_24, 1, infoHeader.biHeight * real_width, fp );
    fclose( fp );
    if( strcmp( argv[1], "logarithm" ) == 0 ){
        timer.Start();

        if( !(logarithm = fopen( argv[3], "wb" )) ){
            printf( "Open Error!\n" );
            return -1;
        }
        double *pixel_y = (double *)calloc( infoHeader.biWidth *
infoHeader.biHeight, sizeof(double) );
        double *pixel_u = (double *)calloc( infoHeader.biWidth *
infoHeader.biHeight, sizeof(double) );
        double *pixel_v = (double *)calloc( infoHeader.biWidth *
infoHeader.biHeight, sizeof(double) );
        // 将rgb数据转换成yuv分别三个数组，同时找出最大的y并对y的值做规范处理到[0, 255]
        double Lmax = 0;
        for( int i = 0; i < infoHeader.biHeight; i++ ){
            for( int j = 0; j < infoHeader.biWidth; j++ ){
                double B = RGBdata_24[ i * real_width + j * 3 + 0 ];
```

```
                double G = RGBdata_24[ i * real_width + j * 3 + 1 ];
                double R = RGBdata_24[ i * real_width + j * 3 + 2 ];
                pixel_y[ i * infoHeader.biWidth + j ] = 0.2990 * R + 0.5870 * G
+ 0.1140 * B;
                pixel_u[ i * infoHeader.biWidth + j ] = -0.1471 * R - 0.2888 * G
+ 0.4360 * B ;
                pixel_v[ i * infoHeader.biWidth + j ] = 0.6150 * R - 0.5150 * G
- 0.1000 * B ;
                if( Lmax < pixel_y[ i * infoHeader.biWidth + j ] ) Lmax =
pixel_y[ i * infoHeader.biWidth + j ];
                if( pixel_y[ i * infoHeader.biWidth + j ] < 0 ) pixel_y[ i *
infoHeader.biWidth + j ] = 0;
                if( pixel_y[ i * infoHeader.biWidth + j ] > 255 ) pixel_y[ i *
infoHeader.biWidth + j ] = 255;
            }
        }
        // 遍历所有的Y值，对其作对数化增强处理--logarithm enhancement
        for( int i = 0; i < infoHeader.biHeight; i++ )
            for( int j = 0; j < infoHeader.biWidth; j++ )
                pixel_y[ i * infoHeader.biWidth + j ] = 255.0000 * (log(
pixel_y[ i * infoHeader.biWidth + j ] + 1 )) / (log( Lmax + 1 ));

        //--将规范化和对数化后的y和uv转换为rgb
        for( int i = 0; i < infoHeader.biHeight; i++ ){
            for( int j = 0; j < infoHeader.biWidth; j++ ){
                double temp = 0;
                temp = 1.0000 * pixel_y[ i * infoHeader.biWidth + j ] + 0.0000 *
pixel_u[ i * infoHeader.biWidth + j ] + 1.1398 * pixel_v[ i * infoHeader.biWidth
+ j ];
                if( temp < 0 ) temp = 0;
                if( temp > 255 ) temp = 255;
                RGBdata_24[ i * real_width + j * 3 + 2 ] = (Byte)temp;  //R
                temp = 1.0000 * pixel_y[ i * infoHeader.biWidth + j ] - 0.3947 *
pixel_u[ i * infoHeader.biWidth + j ] - 0.5806 * pixel_v[ i * infoHeader.biWidth
+ j ];
                if( temp < 0 ) temp = 0;
                if( temp > 255 ) temp = 255;
                RGBdata_24[ i * real_width + j * 3 + 1 ] = (Byte)temp;  //G
                temp = 1.0000 * pixel_y[ i * infoHeader.biWidth + j ] + 2.0321 *
pixel_u[ i * infoHeader.biWidth + j ] + 0.0000 * pixel_v[ i * infoHeader.biWidth
+ j ];
                if( temp < 0 ) temp = 0;
                if( temp > 255 ) temp = 255;
                RGBdata_24[ i * real_width + j * 3 + 0 ] = (Byte)temp;  //B
            }
        }
        //--将写好后的rgb数据连同之前的写进新的bmp
        fseek( logarithm, 0, SEEK_SET );
        fwrite( &bfType, sizeof(Word), 1, logarithm );
        fwrite( &fileHeader, sizeof(BMPFILEHEADER), 1, logarithm );
        fwrite( &infoHeader, sizeof(BMPINFOHEADER), 1, logarithm );
        fwrite( RGBdata_24, 1, infoHeader.biHeight * real_width, logarithm );

        timer.Stop();
        std::cout << "logarithm time: " << timer.GetElapsedMilliseconds() <<
"ms" << std::endl;

        free( RGBdata_24 );
```

```
        fclose( logarithm );
        printf( "OK! The logarithm enhancement of the image has been created!\n"
);
    }
```

对上图进行运行时间统计如下：

方法	O0耗时	O1耗时	O2耗时	O3耗时
基础方法	822.139ms	495.828ms	492.408ms	487.934ms

2.2 向量化指令AVX

2.2.1 Loop 1

下面我们需要对代码中的循环进行向量化，这里我首先对下面这段嵌套循环进行了拆解：

```
for( int i = 0; i < infoHeader.biHeight; i++ ){
    for( int j = 0; j < infoHeader.biWidth; j++ ){
        double B = RGBdata_24[ i * real_width + j * 3 + 0 ];
        double G = RGBdata_24[ i * real_width + j * 3 + 1 ];
        double R = RGBdata_24[ i * real_width + j * 3 + 2 ];
        pixel_y[ i * infoHeader.biWidth + j ] = 0.2990 * R + 0.5870 * G + 0.1140
* B;
        pixel_u[ i * infoHeader.biWidth + j ] = -0.1471 * R - 0.2888 * G +
0.4360 * B ;
        pixel_v[ i * infoHeader.biWidth + j ] = 0.6150 * R - 0.5150 * G - 0.1000
* B ;
        if( Lmax < pixel_y[ i * infoHeader.biWidth + j ] )
            Lmax = pixel_y[ i * infoHeader.biWidth + j ];
        if( pixel_y[ i * infoHeader.biWidth + j ] < 0 )
            pixel_y[ i * infoHeader.biWidth + j ] = 0;
        if( pixel_y[ i * infoHeader.biWidth + j ] > 255 )
            pixel_y[ i * infoHeader.biWidth + j ] = 255;
    }
}
```

阅读之后可以发现，这里我们是先取出图片某个像素的RGB三通道数据，再将之转化到YUV颜色空间中，同时对Y通道的像素数据进行对数化分析和归约处理，防止数据越界；

向量化后的代码如下：

```
for( int i = 0; i < infoHeader.biHeight; i++ ){
    for( int j = 0; j < infoHeader.biWidth; j += 4 ){
        __m256d base_r_vec, base_g_vec, base_b_vec, temp_vec;
        __m256d mul_r_vec, mul_g_vec, mul_b_vec, res_vec;
        base_r_vec = _mm256_set_pd(RGBdata_24[i*real_width+j*3+2],
RGBdata_24[i*real_width+j*3+5], RGBdata_24[i*real_width+j*3+8],
RGBdata_24[i*real_width+j*3+11]);
        base_g_vec = _mm256_set_pd(RGBdata_24[i*real_width+j*3+1],
RGBdata_24[i*real_width+j*3+4], RGBdata_24[i*real_width+j*3+7],
RGBdata_24[i*real_width+j*3+10]);
        base_b_vec = _mm256_set_pd(RGBdata_24[i*real_width+j*3+0],
RGBdata_24[i*real_width+j*3+3], RGBdata_24[i*real_width+j*3+6],
RGBdata_24[i*real_width+j*3+9]);
        // for -> y
```

```
        temp_vec = _mm256_mul_pd(_mm256_set1_pd(0.2990), base_r_vec);
        temp_vec = _mm256_add_pd(temp_vec, _mm256_mul_pd(_mm256_set1_pd(0.5870),
base_g_vec));
        temp_vec = _mm256_add_pd(temp_vec, _mm256_mul_pd(_mm256_set1_pd(0.1140),
base_b_vec));
        _mm256_store_pd(pixel_y + i * infoHeader.biWidth + j, res_vec);
        res_vec = _mm256_setzero_pd();
        temp_vec = _mm256_setzero_pd();
        // for -> u
        temp_vec = _mm256_mul_pd(_mm256_set1_pd(-0.1471), base_r_vec);
        temp_vec = _mm256_add_pd(temp_vec,
_mm256_mul_pd(_mm256_set1_pd(-0.2888), base_g_vec));
        temp_vec = _mm256_add_pd(temp_vec, _mm256_mul_pd(_mm256_set1_pd(0.4360),
base_b_vec));
        _mm256_store_pd(pixel_u + i * infoHeader.biWidth + j, res_vec);
        res_vec = _mm256_setzero_pd();
        temp_vec = _mm256_setzero_pd();
        // for -> v
        temp_vec = _mm256_mul_pd(_mm256_set1_pd(0.6150), base_r_vec);
        temp_vec = _mm256_add_pd(temp_vec,
_mm256_mul_pd(_mm256_set1_pd(-0.5150), base_g_vec));
        temp_vec = _mm256_add_pd(temp_vec,
_mm256_mul_pd(_mm256_set1_pd(-0.1000), base_b_vec));
        _mm256_store_pd(pixel_v + i * infoHeader.biWidth + j, res_vec);
        res_vec = _mm256_setzero_pd();
        temp_vec = _mm256_setzero_pd();
        if( Lmax < pixel_y[ i * infoHeader.biWidth + j ] ) Lmax = pixel_y[ i *
infoHeader.biWidth + j ];
        if( pixel_y[ i * infoHeader.biWidth + j ] < 0 ) pixel_y[ i *
infoHeader.biWidth + j ] = 0;
        if( pixel_y[ i * infoHeader.biWidth + j ] > 255 ) pixel_y[ i *
infoHeader.biWidth + j ] = 255;
    }
}
```

同时进行运行时间统计：

方法	O0耗时	O1耗时	O2耗时	O3耗时
基础方法	822.139ms	495.828ms	492.408ms	487.934ms
向量化loop1	968.772ms	717.619ms	725.71ms	727.789ms

2.2.2 Loop 2

对第二个循环进行向量化，结果如下图：

```
for( int i = 0; i < infoHeader.biHeight; i++ )
    for( int j = 0; j < infoHeader.biWidth - 4; j += 4 ) {
        //pixel_y[ i * infoHeader.biWidth + j ] = 255.0000 * (log( pixel_y[ i *
infoHeader.biWidth + j ] + 1 )) / (log( Lmax + 1 ));
        double y = pixel_y[ i * infoHeader.biWidth + j ];
        __m256d const_vec, base_vec, temp_vec, res_vec;
        base_vec = _mm256_set_pd(pixel_y[i*infoHeader.biWidth+j],
pixel_y[i*infoHeader.biWidth+j+1], pixel_y[i*infoHeader.biWidth+j+2],
pixel_y[i*infoHeader.biWidth+j+3]);
        temp_vec = _mm256_setzero_pd();
        const_vec = _mm256_add_pd(_mm256_set1_pd(1.0), _mm256_set1_pd(Lmax));
        temp_vec = _mm256_add_pd(base_vec, _mm256_set1_pd(1.0));
        res_vec = _mm256_div_pd(temp_vec, const_vec);
        _mm256_store_pd(pixel_y + i * infoHeader.biWidth + j, res_vec);
    }
```

同时进行运行时间统计：

方法	O0耗时	O1耗时	O2耗时	O3耗时
基础方法	822.139ms	495.828ms	492.408ms	487.934ms
向量化loop1	968.772ms	717.619ms	725.71ms	727.789ms
向量化loop2	410.923ms	148.221ms	134.068ms	139.274ms

2.3 Parallelism

优化循环的方法除了向量化，还有多线程的并行处理，这段代码的多线程优化，一个潜在的策略是将图像的每一行分配给不同的线程进行处理。每个线程负责处理部分行，通过这种方式可以提高并行性。

```
void processPixelRange(int startRow, int endRow, int biWidth, double* pixel_y,
double* Lmax) {
    for (int i = startRow; i < endRow; ++i) {
        for (int j = 0; j < biWidth; ++j) {
            // 读取 pixel_y[i * biWidth + j] 的值
            double currentPixelY;
            {
                std::lock_guard<std::mutex> lock(LmaxMutex);
                currentPixelY = pixel_y[i * biWidth + j];
            }

            // 计算新的 pixel_y 值
            double newPixelY = 255.0 * (log(currentPixelY + 1.0) / log(*Lmax +
1.0));

            // 写入新的 pixel_y 值
            {
                std::lock_guard<std::mutex> lock(LmaxMutex);
                pixel_y[i * biWidth + j] = newPixelY;
            }
        }
    }
}
...
...
```

```
// 确定线程数量
const int numThreads = std::thread::hardware_concurrency();
std::vector<std::thread> threads;

// 分配任务给各个线程
int rowsPerThread = infoHeader.biHeight / numThreads;
for (int i = 0; i < numThreads; ++i) {
    int startRow = i * rowsPerThread;
    int endRow = (i == numThreads - 1) ? infoHeader.biHeight : (i + 1) *
rowsPerThread;
    threads.push_back(std::thread(processPixelRange, startRow, endRow,
infoHeader.biWidth, pixel_y, &Lmax));
}
// 等待所有线程完成
for (std::thread& t : threads) {
    t.join();
}
```

同时进行运行时间统计：

方法	O0耗时	O1耗时	O2耗时	O3耗时
基础方法	**822.139ms**	**495.828ms**	**492.408ms**	**487.934ms**
向量化loop1	968.772ms	717.619ms	725.71ms	727.789ms
向量化loop2	**410.923ms**	**148.221ms**	**134.068ms**	**139.274ms**
多线程loop2	2364.07ms	1849.19ms	1831.17ms	1841.75ms

三、效果分析

3.1 向量化角度

从使用向量化指令以提升性能的角度，大部分算法使用AVX指令之后，相比常规的迭代程序都有非常可观的性能进 步。在上面的循环+优化等级组合中，有一半使用AVX后带来了两倍左右的加速比例；尽管256bit的理论上限有4倍左右，但也不错了。这可以解释为何在类似的累和运算使用得非常多的科学计算领域，比如Matlab和Numpy，向 量化指令得到了非常广泛的应用。

然而，有一个特别引人注目的数据，即向量化loop1的结果，竟然比基础的不进行任何优化的方法还慢了15%左右，这就十分让人费解了，按理来说，向量化后，即使不会带来显著的提升也应该不会对原方法产生负面影响才对。

分析我进行向量化的过程，首先，我们可以关注到loop1的整体代码思路，是希望能将图片的像素数据从RGB颜色空间映射到YUV颜色空间，同时由于图片的原始RGB数据是线性存储在一维数组中的，这就导致每一组像素的RGB都需要临时取出在进行三次映射，这或许是原本的代码比较慢的缘故；
而我进行向量化时，首先关注到的是，存储RGB数据信息的数组是BYTE类型的，

```
Byte *RGBdata_24 = NULL;
```

因此当我们进进行向量化时，首先需要考虑AVX的指令能否处理8-bit的SIMD指令，经过查询后结果是不行，因此我首先需要将这些数据扩展到双精度类型，**这会带来一定的位数损失，也是进一步优化的方向**；
然后我又考虑，将RGB三通道用向量化的方式同时操作，这样也达到了SIMD的效果，然而，RGB三个数据完美避开了AVX指令集中对于多个数据同时操作的规定，要么多一个，要么少一个，因此这条路

也行不通；

最后我想到，RGB转到YUV的操作中新的颜色空间的每一个分量都需要RGB来运算，这似乎就是一种横向的**数据依赖**，因此我没法将RGB分开进行向量化，所以我只需要分开YUV进行计算即可，因此最后我选择了同时计算四个像素点的YUV来加速运算。

对loop1向量化思路如上，那为什么会产生“恶化”的效果呢?

观察向量化后的代码可以发现，原本数据的读取和存储都只需要一次：

```
double B = RGBdata_24[ i * real_width + j * 3 + 0 ];
double G = RGBdata_24[ i * real_width + j * 3 + 1 ];
double R = RGBdata_24[ i * real_width + j * 3 + 2 ];
pixel_y[ i * infoHeader.biWidth + j ] = 0.2990 * R + 0.5870 * G + 0.1140 * B;
pixel_u[ i * infoHeader.biWidth + j ] = -0.1471 * R - 0.2888 * G + 0.4360 * B ;
pixel_v[ i * infoHeader.biWidth + j ] = 0.6150 * R - 0.5150 * G - 0.1000 * B ;
```

向量化后，则首先需要从一维数组中读出，存入`__m256`型变量中，在进行运算，存入`res_vec`中，再存入`double`数组中，读写操作的增多在循环的增幅下可能是制约向量化效果的重要因素；

其次， 有时候，过度使用向量化指令可能会导致过多的指令插入，增加了指令缓存的压力，反而也会降低性能。

3.2 并行与分支预测角度

原本是想用多种方法优化一下这段代码的，但是最后发现，多线程的优化策略最后并没有取得比较好的效果；对于原因，我暂时只能想到电脑自身的性能限制，无法进行无限的线程分发；

其次，由于代码中对于同一个数组需要进行大量的读写操作，可以说所有的图像操作和循环操作都集中在几个一维数组上，而图片的数据量又非常大，导致存储与内存中的数组也非常大，当我们频繁索引这个数组中的元素时就会导致索引的耗时过长；

最后，我想，所有线程都需要结束后才能得到我们想要的正确结果，但正因为一些数组上的读写限制，可能并不能在实际操作中实现完美的多线程并发；

至于更深的原因探究，我还没什么想法……

四、实验体会

除了上面进行的向量化的分析和并行的分析，其实可以发现我还考虑了编译器的优化选项，因为编译器也是会自动优化的，但在实验过程中我发现了许多意料之外的现象，比如编译器优化非但不会影响加速比，反而使向量化的效果更佳显 著；特定的代码使用特定的优化等级（不一定是最高等级）编译时，运行效率可能暴增五六个数量级等等。这些令 人惊喜的小收获引领我极高效率的完成了本次探究。

本次最大的体会是“纸上得来终觉浅”，想法总是苍白的，只有在实验的过程中不断发现并解决问题才能将其变得充 实立体；实验过程中的发现和收获也不是看书可以得到的。

五、经验教训

代码需要仔细分析，并不是所有优化方法都会对代码产生正面的效果

六、参考文献

无