计算机组织与体系结构实习 Lab 4

面向特定应用的体系结构优化

苏睿 1600017705

一、使用 x86 SIMD 指令对应用程序进行优化并分析

1. 要求概述

对单幅 YUV 图像的淡入淡出进行处理，程序基本步骤：

(1) 读入一幅 YUV420 格式的图像；

(2) YUV420 到 ARGB8888 转换(Alpha 分别取 1~255，for(A=1;A<255;A=A+3)，共 85 幅图像)；

A = Alpha

R = 1.164383 * (Y - 16) + 1.596027*(V - 128)

B = 1.164383 * (Y - 16) + 2.017232*(U - 128)

G = 1.164383 * (Y - 16) – 0.391762*(U - 128) – 0.812968*(V - 128)

(3) 根据 A 计算 alpha 混合后的 RGB 值（A*RGB/256）,得到不同亮度的 alpha 混合；

R = A*R/256

G = A*G/256

B = A*B/256

(4) 将 alpha 混合后的图像转换成 YUV420 格式，保存输出文件。

Y= 0.256788*R + 0.504129*G + 0.097906*B + 16

U= -0.148223*R - 0.290993*G + 0.439216*B + 128

V= 0.439216*R - 0.367788*G - 0.071427*B + 128

2. 编译执行

程序开发环境为 Ubuntu 18.04.3 LTS，使用的编程语言为 C++，编译器为 g++7.4.0 共包含 3 个文件 main.cpp, transform.h, transform.cpp，附有 makefile 文件，使用 make 指令进行编译，获得可执行文件 trans。

编译命令为 g++ main.cpp transform.cpp -o trans -std=c++11 -mavx2 -O2

trans 的执行方式：

./trans [yuvfile] [type]

其中 yuvfile 为输入 yuv 文件的路径，type 为运行所选的指令集，有以下选项-BASIC, -MMX, -SSE2, -AVX。

如 ./trans demo/dem1.yuv -MMX

程序对 Alpha 分别取 1~255，for(A=1;A<255;A=A+3)，得到 85 个 yuv 图像存放在 alpha 文件夹，输出图像转换所用总时间和每张图像的转换平均用时（不计文件读取和保存的时间）。

3. 程序实现

基本流程已在概述中描绘了，这里只简单说些细节。

(1) MMX 为 4 像素同时处理，SSE2 为 8 像素，AVX 为 16 像素。

(2) 由于输出文件是 YUV420 格式，因此从 RGB 转换到 YUV 时，所得到的 U、V 为周围四个像素计算得到的 U、V 的平均值。

(3) 在使用 SIMD 汇编时，从计算得到的__m64, __m128i, __m256i 提取数值到 Y,U,V 数组中，没有找到对应的汇编函数，因此使用了 union 结构，将 unsigned char[]和__mxxx 放入 union 中进行提取。

(4) 在将原始 YUV 数据加载到__m64, __m128i, __m256i 中时，可以使用 set 系列的汇编函数，但是前面都定义 union 了，不用白不用，而且整齐好看，所以就没有 set 汇编。

(5) 在汇编中虽然使用了 param1 = _mm_set1_epi16((short)(1.164383 * 256)); 这样的语句，看起来这个乘法没有使用 SIMD，但其实可以用计算器敲出来，而且在编译器 -O2 优化下应该不怎么影响性能，所以就懒得改了。

(6) 使用汇编转换出来的图像在颜色上有点问题，不清楚是因为在 1.164383 * 256 这样的地方进行了截取的缘故，还是某些位置对溢出的处理出现了问题，我太困了= =

4. 运行结果

程序以各种指令集运行在 dem1.yuv 和 dem2.yuv 下的结果：

```
surui@surui:~/Documents/architecture/SIMD_trans$ ./trans demo/dem1.yuv -BASIC
 total 85 pictures transform time: 0.057165s
                time per picture: 0.000673s
surui@surui:~/Documents/architecture/SIMD_trans$ ./trans demo/dem1.yuv -MMX
 total 85 pictures transform time: 0.031389s
                time per picture: 0.000369s
surui@surui:~/Documents/architecture/SIMD_trans$ ./trans demo/dem1.yuv -SSE2
 total 85 pictures transform time: 0.020177s
                time per picture: 0.000237s
surui@surui:~/Documents/architecture/SIMD_trans$ ./trans demo/dem1.yuv -AVX
 total 85 pictures transform time: 0.016348s
                time per picture: 0.000192s
```

```
surui@surui:~/Documents/architecture/SIMD_trans$ ./trans demo/dem2.yuv -BASIC
 total 85 pictures transform time: 0.048938s
                time per picture: 0.000576s
surui@surui:~/Documents/architecture/SIMD_trans$ ./trans demo/dem2.yuv -MMX
 total 85 pictures transform time: 0.030729s
                time per picture: 0.000362s
surui@surui:~/Documents/architecture/SIMD_trans$ ./trans demo/dem2.yuv -SSE2
 total 85 pictures transform time: 0.018540s
                time per picture: 0.000218s
surui@surui:~/Documents/architecture/SIMD_trans$ ./trans demo/dem2.yuv -AVX
 total 85 pictures transform time: 0.021499s
                time per picture: 0.000253s
```

将每幅图像平均转换时间记录如下：

/秒	BASIC	MMX	SSE2	AVX
dem1.yuv	0.000673(100%)	0.000369(54.8%)	0.000237(35.2%)	0.000192(28.5%)
dem2.yuv	0.000576(100%)	0.000362(62.8%)	0.000218(37.8%)	0.000253(43.9%)

我们可以看到，各种汇编的优化下图像转换的速度确实提高了不少。

(1) 原以为 MMX 是 4 像素同时处理，那么耗时会变为 1/4 左右，看来我想多了，外层循环的时间和数据载入提取的时间也占不少。

(2) 以上数据没有多次循环取均值，所以 dem2.yuv 的 AVX 看起来比 SSE2 慢，其实大多数情况还是 AVX 快于 SSE2 的。

二、设计自定义扩展指令对 SIMD 应用优化并分析

1. 设计若干 32 位宽的扩展指令。

没太看懂题目，感觉 AVX 的指令还挺全面的，但就前面的 YUV 图像转换任务而言，在将数据转化到__m256i 以及将__m256i 转换回数据时缺少高效的指令。

设计的指令有

(1) vphcrwb 对以 16 位存储的数据，每个截取低 8 位，存储在新寄存器的低 128 位中，高 128 位置零。

对应 c 函数为__m256i _mm256_hcr_epi16(__m256i a);

```
FOR j := 0 to 15
    i := j*8
    k:= j*16
    dst[i+7:i] := a[k+7:k]
ENDFOR
dst[255:128] := 0
```

(2) vphclwb 对以 8 位存储的数据，将低 16 个数据每个以零扩展成 16 位，存储在新寄存器中。

对应 c 函数为__m256i _mm256_hcl_epi16(__m256i a);

```
FOR j := 0 to 15
    i := j*8
    k:= j*16
    dst[k+15:k] := a[i+7:i]
ENDFOR
```

(3) vpdupw 对以 16 位存储的数据，将低 8 个数据复制，并按照 aabbcc...的方式存放在新寄存器中。

对应 c 函数为__m256i _mm256_dup_epi16(__m256i a);

```
FOR j := 0 to 7
    i:= j*16
    dst[i*2+15:i*2] := a[i+15:i]
    dst[i*2+31:i*2+16] := a[i+15:i]
ENDFOR
```

(4) vpavgw 对以 16 位存储的数据，将相邻两数据取均值，共得到 8 个结果，存储在新的寄存器中。

对应 c 函数为__m256i _mm256_avg_epi16(__m256i a);

```
FOR j := 0 to 7
    i:= j*2*16
    k:=j*16
    dst[k+15:k] := (a[i+15:i] + a[i+31:i+16]) >> 1
ENDFOR
dst[255:128] := 0
```

(5) vpstorer 将寄存器的低 128 位存储到给定地址上；vpstorep 将寄存器的低 64 位存储到给定地址上。

对应 c 函数为 void _mm256_store_r(char* p, __m256i a)

void _mm256_store_p(char* p, __m256i a)

(6) vploadr 读取给定地址的 128 位到寄存器; vploadp 读取给定地址的 64 位到寄存器。

对应 c 函数为 void _mm256_load_r(char* p, __m256i a)
void _mm256_load_p(char* p, __m256i a)

2. 以上指令的助剂符含义分别为

vphcrwb：vp 好像是 256 位指令头，这里直接照搬；hcr 意为 half cut right，就是指将数据的一半截取并向右靠拢；wb 为数据的长度，w 指 16 位，b 指 8 位，即由 16 位截断为 8 位，此处也可以扩展出 dw,db 等。

vphclwb：vp 同上；hcl 意为 half cut left，就是将数据扩展，并向左靠拢，和上面对称；wb 同上。

vpdupw：vp 同上；dup 意为 duplicate，表示将数据复制；w 指 16 位数据，同样可以扩展出 w, b 等。

vpavgw：vp 同上；avg 意为 average，表示对相邻数据取均值；w 同上。

vpstorer, vpstorep, vploadr, vploadp 中的 r 和 p 随便写的...

编码直接就算了= =

3. 这里还是使用 c 函数编写对应的 SIMD 转换函数。

```
void avx_transform()
{
    memset(newY, 0, sizeof(newY));
    memset(newU, 0, sizeof(newU));
    memset(newV, 0, sizeof(newV));

    for(int i = 0; i < pic_h; ++i)
    {
        for(int j = 0; j < pic_w; j += 16)
        {
            __m256i y, u, v;
            _mm256_load_r(Y + i * pic_w + j, y);
            _mm256_load_p(U + (i/2) * (pic_w/2) + (j/2), u);
            _mm256_load_p(V + (i/2) * (pic_w/2) + (j/2), v);
            y = _mm256_hcl_epi16(y);
            u = _mm256_hcl_epi16(u);
            v = _mm256_hcl_epi16(v);
            u = _mm256_dup_epi16(u);
            v = _mm256_dup_epi16(v);

            __m256i i16 = _mm256_set1_epi16(16), i128 = _mm256_set1_epi16(128);
            y = _mm256_sub_epi16(y, i16);
            u = _mm256_sub_epi16(u, i128);
            v = _mm256_sub_epi16(v, i128);

            __m256i param1, param2, param3, param4, param5;
            param1 = _mm256_set1_epi16((short)(1.164383 * 256));
            param2 = _mm256_set1_epi16((short)(1.596027 * 256));
```

```
            param3 = _mm256_set1_epi16((short)(2.017232 * 256));
            param4 = _mm256_set1_epi16((short)(0.391762 * 256));
            param5 = _mm256_set1_epi16((short)(0.812968 * 256));

            __m256i r, g, b;
            r = g = b = _mm256_mullo_epi16(y, param1);
            r = _mm256_adds_epi16(r, _mm256_mullo_epi16(v, param2));
            b = _mm256_adds_epi16(b, _mm256_mullo_epi16(u, param3));
            g = _mm256_subs_epi16(g, _mm256_add_epi16(_mm256_mullo_epi16(u, param4), _mm256_mullo_epi16(v,
param5)));

            r = _mm256_srai_epi16(r, 8);
            g = _mm256_srai_epi16(g, 8);
            b = _mm256_srai_epi16(b, 8);

            param1 = _mm256_set1_epi16((short)alpha);
            r = _mm256_srai_epi16(_mm256_mullo_epi16(r, param1), 8);
            g = _mm256_srai_epi16(_mm256_mullo_epi16(g, param1), 8);
            b = _mm256_srai_epi16(_mm256_mullo_epi16(b, param1), 8);

            param1 = _mm256_set1_epi16((short)(0.256788 * 256));
            param2 = _mm256_set1_epi16((short)(0.504129 * 256));
            param3 = _mm256_set1_epi16((short)(0.097906 * 256));
            y = _mm256_add_epi16(_mm256_mullo_epi16(r, param1), _mm256_mullo_epi16(g, param2));
            y = _mm256_adds_epi16(y, _mm256_mullo_epi16(b, param3));
            y = _mm256_srai_epi16(y, 8);
            y = _mm256_add_epi16(y, i16);

            param1 = _mm256_set1_epi16((short)(0.148223 * 256));
            param2 = _mm256_set1_epi16((short)(0.290993 * 256));
            param3 = _mm256_set1_epi16((short)(0.439216 * 256));
            u = _mm256_sub_epi16(_mm256_mullo_epi16(b, param3), _mm256_mullo_epi16(r, param1));
            u = _mm256_subs_epi16(u, _mm256_mullo_epi16(g, param2));
            u = _mm256_srai_epi16(u, 8);
            u = _mm256_add_epi16(u, i128);

            param1 = _mm256_set1_epi16((short)(0.439216 * 256));
            param2 = _mm256_set1_epi16((short)(0.367788 * 256));
            param3 = _mm256_set1_epi16((short)(0.071427 * 256));
            v = _mm256_sub_epi16(_mm256_mullo_epi16(r, param1), _mm256_mullo_epi16(g, param2));
            v = _mm256_subs_epi16(v, _mm256_mullo_epi16(b, param3));
            v = _mm256_srai_epi16(v, 8);
            v = _mm256_add_epi16(v, i128);
```

```
            y = _mm256_hcr_epi16(y);
            u = _mm256_avg_epi16(u);
            u = _mm256_hcr_epi16(u);
            v = _mm256_avg_epi16(v);
            v = _mm256_hcr_epi16(v);
            _mm256_store_r(newY + i * pic_w + j, y);
            _mm256_store_p(newU + (i/2) * (pic_w/2) + (j/2), u);
            _mm256_store_p(newV + (i/2) * (pic_w/2) + (j/2), v);
        }
    }
}
```

4. 定性分析采用自己设计的 SIMD 扩展指令后可以获得最大指令减少数（相对于未使用 SIMD 指令），以及可以获得的潜在性能提升。

对于每个循环，修改之前需要循环载入和提取数据，共需要 16 * 3 * 2 = 96 条指令；修改之后一次性你载入和提取数据，共需要 8 + 8 = 16 条指令。

每个循环可以减少 80 条指令，还是比较可观的。