北 京 大 学

信息科学技术学院

本科生实验报告

实验课程：计算机组成与系统结构实习

实验名称：基于 SIMD 扩展指令的图像处理加速

撰 写 人：季凯航

学　　号：1400012727

组内成员：季凯航、李济洲、岳越

撰写时间：2016 年 12 月 24 日

目录

一、实验描述

本次实验（Lab 4.1）要求实现一款基于 SIMD 的 YUV 图像处理程序，包含以下两项功能：

- 单幅图像的淡入淡出处理 (Fade in and fade out one image)
- 两幅图像叠加的渐变实现 (Create a gradient which overlays two images)

并且，针对以上部分需要实现以下四个版本：

- 无 SIMD 指令 (Non-SIMD)
- 基于 MMX 指令集 (MMX)
- 基于 SSE2 指令集 (SSE2)
- 基于 AVX 指令集 (AVX)

二、实验准备

1．YUV

1.1 YUV 简介

YUV 是被欧洲电视系统所采用的一种颜色编码方法（属于 PAL），是 PAL 和 SECAM 模拟彩色电视制式采用的颜色空间。在现代彩色电视系统中，通常采用三管彩色摄影机或彩色 CCD 摄影机进行取像，然后把取得的彩色图像信号经分色、分别放大校正后得到 RGB，再经过矩阵变换电路得到亮度信号 Y 和两个色差信号 B－Y（即 U)、R－Y（即 V)，最后发送端将亮度和色差三个信号分别进行编码，用同一信道发送出去。这种色彩的表示方法就是所谓的 YUV 色彩空间表示。采用 YUV 色彩空间的重要性是它的亮度信号 Y 和色度信号 U、V 是分离的。

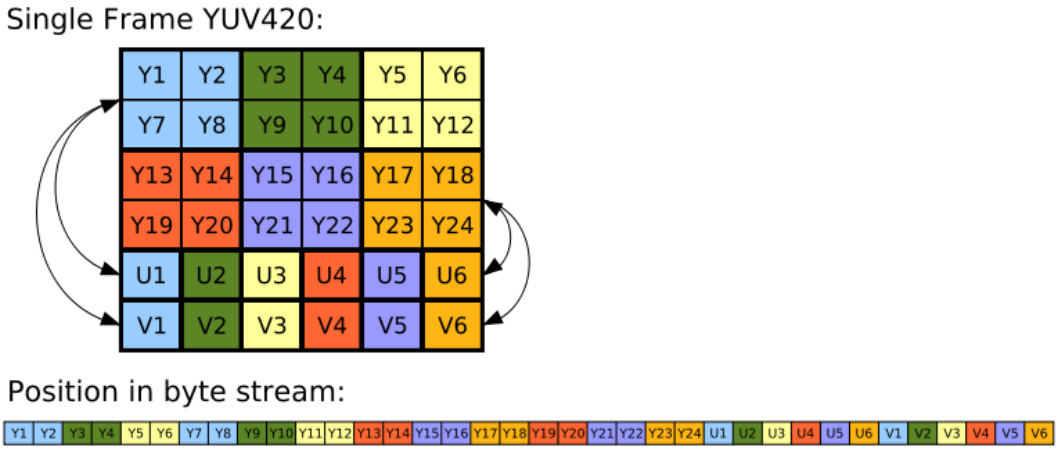
1.2 YUV420 简介

本次 lab 使用的格式为 YUV4:2:0。它指得是对每行扫描线来说，只有一种色度分量以 2:1 的抽样率存储。相邻的扫描行存储不同的色度分量，也就是说，如果一行是 4:2:0 的话，下一行就是 4:0:2，再下一行是 4:2:0...以此类推。对每个色度分量来说，水平方向和竖直方向的抽样率都是 2:1，所以可以说色度的抽样率是 4:1。对非压缩的 8 比特量化的视频来说，每个

由 2x2 个 2 行 2 列相邻的像素组成的宏像素需要占用 6 字节内存。

1.3 YUV420 文件格式

如下图所示，以单帧的 6*4 YUV420 为例，其存放码流为[Y1 Y2 Y3 Y4 …Y24 U1 U2 U3 …U6 V1 V2 V3… V6]。



对于多帧的 YUV420 文件，每一帧直接跟在上一帧后即可。

2．YUV420 转 RGB888

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.164383 & 2.017232 & 0 \\ 1.164383 & -0.391762 & -0.812968 \\ 1.164383 & 0 & 1.596027 \end{bmatrix} \begin{bmatrix} Y-16 \\ U-128 \\ V-128 \end{bmatrix}$$

YUV420 转 RGB888 满足以上公式。要注意的是，此处应当有饱和运算。从公式中不难看出，YUV420 中的 Y、U、V 三通道都应为 uint8_t。[1]

3．RGB888 转 YUV420

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.256788 & 0.504129 & 0.097906 \\ -0.148223 & -0.290993 & 0.439216 \\ 0.439216 & -0.367788 & -0.071427 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix}$$

RGB888 转 YUV420 满足以上公式。同样，此处也应有饱和运算。

[1] 而老一版中给出的转换公式，要求 Y 为 uint8_t，而 U 和 V 皆应存储为 int8_t。

三、程序结构

考虑到本次实验需要使用特殊的指令集，故使用 C++作为开发语言，使用 VC++ WIN32 作为编译器。

共设立 4 个命名空间：Non_Simd、MMX、SSE、AVX。各命名空间下的函数为其指令集上的实现。

以下介绍程序中的主要类（略去次要成员变量和成员函数）和主要函数。

1．class YUV[2]

该类主要用于 yuv 格式数据存储，包含了一些相关输入输出的成员函数。

类成员变量：

```
uint8_t* y8;
uint8_t* u8;
uint8_t* v8;
```

2．class RGB[3]

该类主要用于 rbg 格式数据存储，包含了一些相关输入输出（BMP24）的成员函数。

类成员变量：

```
uint8_t* r8;
uint8_t* g8;
uint8_t* b8;
```

3．function yuv2rgb()[4]

该函数用于将 YUV 格式转换为 RGB 格式。读入 YUV；输出 RGB。

读入：`const YUV* src_yuv`

输出：`const RGB* dst_rgb`

有四个实现版本，分属于四个命名空间：

```
void Non_Simd::yuv2rgb(const RGB* dst_rgb, const YUV* src_yuv);
void MMX::yuv2rgb(const RGB* dst_rgb, const YUV* src_yuv);
void SSE::yuv2rgb(const RGB* dst_rgb, const YUV* src_yuv);
```

[2] 详情请见 yuv.h
[3] 详情请见 rgb.h
[4] 详情请见 yuv2rgb.cpp

```
void AVX::yuv2rgb(const RGB* dst_rgb, const YUV* src_yuv);
```

4. function rgb2yuv()[5]

该函数用于将 RGB 格式转换为 YUV 格式。读入 RGB；输出 YUV。

读入：const RGB* src_rgb

输出：const YUV* dst_yuv

有四个实现版本，分属于四个命名空间：

```
void Non_Simd::rgb2yuv(const YUV* dst_yuv, const RGB* src_rgb);
void MMX::rgb2yuv(const YUV* dst_yuv, const RGB* src_rgb);
void SSE::rgb2yuv(const YUV* dst_yuv, const RGB* src_rgb);
void AVX::rgb2yuv(const YUV* dst_yuv, const RGB* src_rgb);
```

5. function alpha_blend()[6]

该函数用于将 RGB 格式进行透明变换。读入 RGB 和指定 alpha 值，输出变换后的 RGB。

读入：const RGB* src_rgb, const uint8_t alpha

输出：const RGB* dst_rgb

```
void Non_Simd::alpha_blend(const RGB* dst_rgb,
                           const RGB* src_rgb, const uint8_t alpha);
void MMX::alpha_blend(const RGB* dst_rgb, const RGB* src_rgb, const uint8_t alpha);
void SSE::alpha_blend(const RGB* dst_rgb, const RGB* src_rgb, const uint8_t alpha);
void AVX::alpha_blend(const RGB* dst_rgb, const RGB* src_rgb, const uint8_t alpha);
```

6. function image_overlay()[7]

该函数用于将两个 RGB 格式依照给定的透明度进行叠加。读入两个 RGB 和指定 alpha 值，输出变换后的 RGB。

读入：const RGB* src_rgb_1, const RGB* src_rgb_2, const uint8_t alpha

[5] 详情请见 rgb2yuv.cpp
[6] 详情请见 alpha_blend.cpp
[7] 详情请见 image_overlay.cpp

输出：const RGB* dst_rgb

```
void Non_Simd::image_overlay(const RGB* dst_rgb,
                const RGB* src_rgb_1, const RGB* src_rgb_2, const uint8_t alpha);
void MMX::image_overlay(const RGB* dst_rgb,
                const RGB* src_rgb_1, const RGB* src_rgb_2, const uint8_t alpha);
void SSE::image_overlay(const RGB* dst_rgb,
                const RGB* src_rgb_1, const RGB* src_rgb_2, const uint8_t alpha);
void AVX::image_overlay(const RGB* dst_rgb,
                const RGB* src_rgb_1, const RGB* src_rgb_2, const uint8_t alpha);
```

四、实现细节

1．Non-SIMD

1.1 yuv2rgb

依照公式使用浮点数乘实现。

1.2 rgb2yuv

依照公式使用浮点数乘实现。

1.3 alpha_blend

使用整数乘，之后进行右移操作。即(Input * alpha) >> OFFSET

1.4 image_overlay

使用整数乘，之后进行右移操作。即(Input1 * alpha + Input2 * (~alpha)) >> OFFSET

2．MMX

使用 mmintrin.h。

2.1 yuv2rgb

虽然所有的数据存储都为 8 位的 uint8_t 类型，但需要注意的是，在运算过程中会发生上下溢出。故实际应当使用 int16_t 进行运算。需要特殊说明的是，若不使用饱和运算指令，

则运算结果可能为负，故不应当使用 uint16_t。

注意到 MMX 并不支持浮点数运算，故使用整数运算代替。事实上，在 yuv 转 rgb 这一过程中，经常使用整数运算用于代替浮点运算加速或是获取更好的硬件系统兼容性。(例如 TENSORFLOW)

__m64 是 Microsoft 为 MMX 指令集而设计的数据类型。可以支持 4 个 int16_t 或 uint16_t 存储于其中同时进行加法、减法、乘法、位移运算。其中需要特殊说明其乘法指令_m_pmulhw 和_m_pmullw。这两条指令为将两个__m64 中的 8 个 int16_t 对应两两相乘，前者输出乘法结果的高 16 位，后者输出乘法结果的低 16 位。

运用以上特性，我们可以将所有转换系数乘以2^{16}后保存为整数。再将这些整系数与待处理相乘,保留其结果的高 16 位即可,低 16 位直接抛弃。原本使用整数运算完成这一操作，需要先将系数整体乘以2^t（t 为选定的正整数），乘完后再乘以待运算数据，之后将结果算术右移 t 位。而现在使用_m_pmulhw 指令并将 t 取定为 16 则可以省去这右移操作。

尽管这一方式非常简便，但我们需要注意到，并非所有的系数都可以直接乘以2^{16}后保存。如 yuv 转 rgb 中，U 通道相对于 R 通道的系数为 2.017232。它乘以2^{16}将会超过 int16_t 的上界，甚至 uint16_t 的上界。为解决这一问题，只能将其减去 2，在之后的运算中，手动让被乘数左移 1 位并加至结果中。这一方法虽然运行速度非常快，但是缺少代码移植能力。

其函数流程为：

计算 R 通道；

计算 G 通道；

计算 B 通道；

不同的通道计算流程相似，以下以 U 通道叠加至 R 通道为例：

```
// U Channel to R Channel : R += (U - 128) * 2.017232
data = _m_psubw(*src_u, OFFSET_128);        // data = U - 128
tmp = _m_pmulhw(data, U2R);                 // tmp = data * 0.017232
*dst_r = _m_paddsw(*dst_r, tmp);            // R += tmp
tmp = _m_psllwi(data, 1);                   // tmp = data << 1
*dst_r = _m_paddsw(*dst_r, tmp);            // R += tmp
```

2.2 rgb2yuv

与 2.1 中原理一致。

其函数流程为：

计算 Y 通道；

计算 U 通道；

计算 V 通道；

要注意的是，Y 的存储是 U 的四倍、V 的四倍。除此以外，不同通道的计算流程大致相同，下以计算 Y 通道为例：

```
// R Channel to Y Channel : Y = R * 0.256788
*dst = _m_pmulhw(*src_r, R2Y);          // Y = R * 0.256788

// G Channel to Y Channel : Y += G * 0.504129
tmp = _m_pmulhw(*src_g, G2Y);           // tmp = G * 0.004129
*dst = _m_paddsw(tmp, *dst);            // Y += tmp
tmp = _m_psrlwi(*src_g, 1);             // tmp = G >> 1;
*dst = _m_paddsw(tmp, *dst);            // Y += tmp

// B Channel to Y Channel  : Y += B * 0.097906
tmp = _m_pmulhw(*src_b, B2Y);           // tmp = B * 0.097906
*dst = _m_paddsw(tmp, *dst);            // Y += tmp

// Add offset : Y += 16
*dst = _m_paddsw(*dst, OFFSET_16);      // Y += 16
```

2.3 alpha_blend

相较于 2.1 和 2.2 较为简单。

以下给出每轮循环中的核心代码：

```
// Alpha_blend : Output = (Input * Alpha) >> 8
tmp = _m_pmullw(*src, alpha);           // tmp = Input * Alpha
*dst = _m_psrlwi(tmp, 8);               // Output = tmp >> 8
```

2.4 image_overlay

本质是 2.3 的加强版。

以下给出每轮循环中的核心代码：

```
// Image_overlay : Output = (Input1 * Alpha + Input2 * _Alpha) >> 8
tmp1 = _m_pmullw(*src1, alpha);         // tmp1 = Input1 * Alpha
tmp2 = _m_pmullw(*src2, _alpha);        // tmp2 = Input2 * _Alpha
*dst = _m_paddusw(tmp1, tmp2);          // Output = tmp1 + tmp2
*dst = _m_psrlwi(*dst, 8);              // Output = Output >> 8
```

3．SSE2

使用 emmintrin.h。

3.1 yuv2rgb

注意到 SSE2 虽然支持双精度浮点数运算，但考虑到速度因素，仍使用整数运算代替。

__m128i 是 Microsoft 为 SSE2 指令集而设计的整数数据类型。可以支持 8 个 int16_t 或 uint16_t 存储于其中同时进行加法、减法、乘法、位移运算。其中需要特殊说明其乘法指令 _mm_mulhi_epi16 和 _mm_mullo_epi16，它们对应了 MMX 中的 _m_pmulhw 和 _m_pmullw。这两条指令为将两个 __m128i 中的 8 个 int16_t 对应两两相乘，前者输出乘法结果的高 16 位，后者输出乘法结果的低 16 位。

其函数流程与 2.1 一致。

不同的通道计算流程相似，以下以 U 通道叠加至 R 通道为例：

```
// U Channel to R Channel : R += (U - 128) * 2.017232
data = _mm_subs_epi16(*src_u, OFFSET_128);  // data = U - 128
tmp = _mm_mulhi_epi16(data, U2R);           // tmp = data * 0.017232
*dst = _mm_adds_epi16(*dst, tmp);           // R += tmp
tmp = _mm_slli_epi16(data, 1);              // tmp = data << 1
*dst = _mm_adds_epi16(*dst, tmp);           // R += tmp
```

3.2 rgb2yuv

与 3.1 中原理一致。

其函数流程与 2.2 一致。

下以计算 Y 通道为例：

```
// R Channel to Y Channel : Y = R * 0.256788
*dst = _mm_mulhi_epi16(*src_r, R2Y);        // Y = R * 0.256788

// G Channel to Y Channel : Y += G * 0.504129
tmp = _mm_mulhi_epi16(*src_g, G2Y);         // tmp = G * 0.004129
*dst = _mm_adds_epi16(tmp, *dst);           // Y += tmp
tmp = _mm_srli_epi16(*src_g, 1);            // tmp = G >> 1;
*dst = _mm_adds_epi16(tmp, *dst);           // Y += tmp

// B Channel to Y Channel  : Y += B * 0.097906
tmp = _mm_mulhi_epi16(*src_b, B2Y);         // tmp = B * 0.097906
*dst = _mm_adds_epi16(tmp, *dst);           // Y += tmp

// Add offset : Y += 16
*dst = _mm_adds_epi16(*dst, OFFSET_16);     // Y += 16
```

3.3 alpha_blend

其函数流程与 2.3 一致。

以下给出每轮循环中的核心代码：

```
// Alpha_blend : Output = (Input * Alpha) >> 8
tmp = _mm_mullo_epi16(*src, alpha);          // tmp = Input * Alpha
*dst = _mm_srli_epi16(tmp, 8);               // Output = tmp >> 8
```

3.4 image_overlay

本质是 3.3 的加强版。其函数流程与 2.4 一致。

以下给出每轮循环中的核心代码：

```
// Image_overlay : Output = (Input1 * Alpha + Input2 * _Alpha) >> 8
tmp1 = _mm_mullo_epi16(*src1, alpha);        // tmp1 = Input1 * Alpha
tmp2 = _mm_mullo_epi16(*src2, _alpha);       // tmp2 = Input2 * _Alpha
*dst = _mm_add_epi16(tmp1, tmp2);            // Output = tmp1 + tmp2
*dst = _mm_srli_epi16(*dst, 8);              // Output = Output >> 8
```

4. AVX

使用 immintrin.h。

4.1 yuv2rgb

注意到 AVX 虽然支持双精度浮点数运算，但考虑到速度因素，仍使用整数运算代替。

__m256i 是 Microsoft 为 AVX 指令集而设计的整数数据类型。可以支持 16 个 int16_t 或 uint16_t 存储于其中同时进行加法、减法、乘法、位移运算。其中需要特殊说明其乘法指令 _mm256_mulhi_epi16 和 _mm256_mullo_epi16，它们对应了 MMX 中的 _m_pmulhw 和 _m_pmullw。这两条指令为将两个 __m256i 中的 16 个 int16_t 对应两两相乘，前者输出乘法结果的高 16 位，后者输出乘法结果的低 16 位。

其函数流程与 2.1 一致。

不同的通道计算流程相似，以下以 U 通道叠加至 R 通道为例：

```
// U Channel to R Channel : R += (U - 128) * 2.017232
data = _mm256_subs_epi16(*src_u, OFFSET_128);    // data = U - 128
tmp = _mm256_mulhi_epi16(data, U2R);             // tmp = data * 0.017232
*dst = _mm256_adds_epi16(*dst, tmp);             // R += tmp
tmp = _mm256_slli_epi16(data, 1);                // tmp = data << 1
*dst = _mm256_adds_epi16(*dst, tmp);             // R += tmp
```

4.2 rgb2yuv

与 4.1 中原理一致。

其函数流程与 2.2 一致。

下以计算 Y 通道为例：

```
// R Channel to Y Channel : Y = R * 0.256788
*dst = _mm256_mulhi_epi16(*src_r, R2Y);      // Y = R * 0.256788

// G Channel to Y Channel : Y += G * 0.504129
tmp = _mm256_mulhi_epi16(*src_g, G2Y);       // tmp = G * 0.004129
*dst = _mm256_adds_epi16(tmp, *dst);         // Y += tmp
tmp = _mm256_srli_epi16(*src_g, 1);          // tmp = G >> 1;
*dst = _mm256_adds_epi16(tmp, *dst);         // Y += tmp

// B Channel to Y Channel  : Y += B * 0.097906
tmp = _mm256_mulhi_epi16(*src_b, B2Y);       // tmp = B * 0.097906
*dst = _mm256_adds_epi16(tmp, *dst);         // Y += tmp

// Add offset : Y += 16
*dst = _mm256_adds_epi16(*dst, OFFSET_16);   // Y += 16
```

4.3 alpha_blend

其函数流程与 2.3 一致。

以下给出每轮循环中的核心代码：

```
// Alpha_blend : Output = (Input * Alpha) >> 8
tmp = _mm256_mullo_epi16(*src, alpha);       // tmp = Input * Alpha
*dst = _mm256_srli_epi16(tmp, 8);            // Output = tmp >> 8
```

4.4 image_overlay

本质是 4.3 的加强版。其函数流程与 2.4 一致。

以下给出每轮循环中的核心代码：

```
// Image_overlay : Output = (Input1 * Alpha + Input2 * _Alpha) >> 8
tmp1 = _mm256_mullo_epi16(*src1, alpha);     // tmp1 = Input1 * Alpha
tmp2 = _mm256_mullo_epi16(*src2, _alpha);    // tmp2 = Input2 * _Alpha
*dst = _mm256_add_epi16(tmp1, tmp2);         // Output = tmp1 + tmp2
*dst = _mm256_srli_epi16(*dst, 8);           // Output = Output >> 8
```

五、实验结果

1. 正确性

输入为图 1 和图 2：

图 1 dem1.yuv (Frame 1/1)

图 2 dem2.yuv (Frame 1/1)

对于 Task 1: 单幅图像的淡入淡出处理，以 dem1.yuv 为输入，输出见图 3、图 4、图 5。

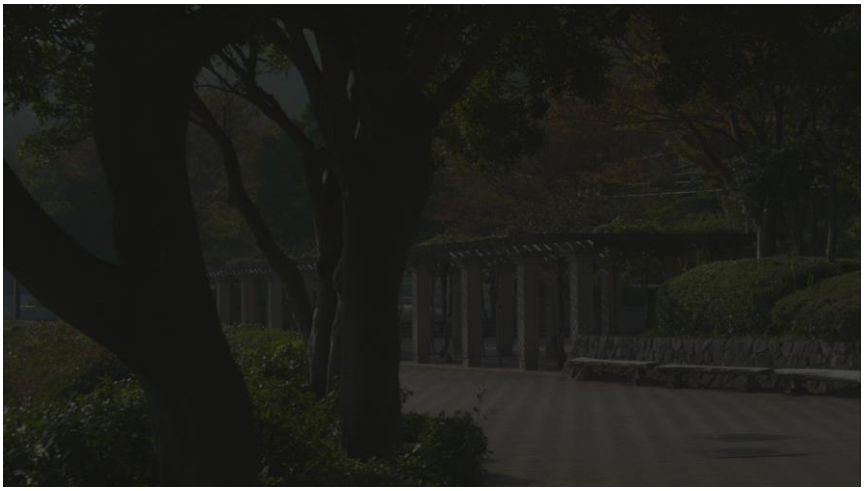


图 3 alpha_blend.yuv (Frame 25/85)

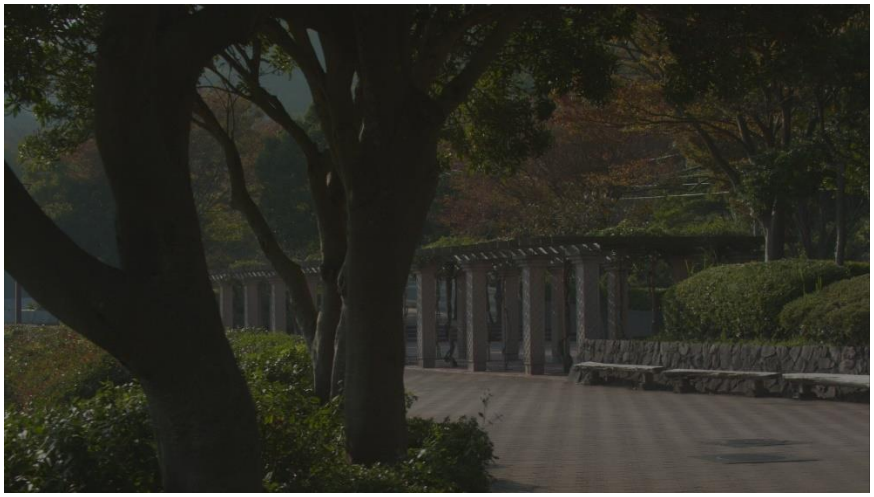


图 4 alpha_blend.yuv (Frame 50/85)

图 5 alpha_blend.yuv (Frame 75/85)

对于 Task 2: 两幅图像的渐变叠加实现，以 dem1.yuv 和 dem2.yuv 为输入，输出见图 36、图 7、图 8。

图 6 image_overlay.yuv (Frame 22/85)

图 7 image_overlay.yuv (Frame 42/85)

图 8 image_overlay.yuv (Frame 62/85)

此处需要注明的是，以上给出的截图皆为 MMX/SSE2/AVX 的运行结果，它们三者的运行结果一致，使用 windows 下的 cmp 命令找不到差异。

而若使用 Non-SIMD 运行，则可能会在像素的通道上存在不大于±1的肉眼难以识别的差异。造成这一差异的主要原因是 Non-SIMD 使用的是浮点运算，与其它三者的近似方式不同。

2. 性能

以下为本次测评的相关参数：

测试平台：64 位 Windows 10 企业版 2015 LTSB

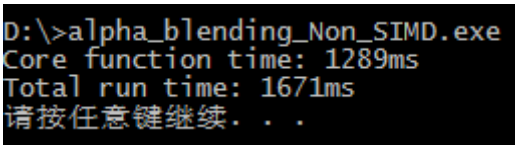
编译器：cl.exe

编译优化选项：-Ox

CPU：Intel i7-6700K

RAM: 16GB DDR4 2133MHz

程序在退出前，会在 stderr 中输出运行时间，如下所示：



其中 Core function time 指的是相关本 Lab 关心的函数部分的运行总时长；Total run time 指的是程序的总运行时长。这二者的时长差异主要是因为文件输入输出。

以下仅对前者进行分析。采用记录三次运行时长取平均的方式记录各个配置的运行时长。以下为数据记录：（单位为 ms）

Non_SIMD	1	2	3	Average
Alpha Blend	1279	1289	1289	1285.7
Image Overlay	1702	1693	1680	1691.7

Non-SIMD 关键函数运行总时长记录表

MMX	1	2	3	Average	Improvement Compared with Non-SIMD	Improvement Compared with the previous one
Alpha Blend	686	677	678	680.3	89.0%	89.0%
Image Overlay	722	723	733	726.0	133.0%	133.0%

MMX 关键函数运行总时长记录表

SSE	1	2	3	Average	Improvement Compared with Non-SIMD	Improvement Compared with the previous one
Alpha Blend	620	619	631	623.3	106.3%	9.1%
Image Overlay	671	673	661	668.3	153.1%	8.6%

SSE2 关键函数运行总时长记录表

AVX	1	2	3	Average	Improvement Compared with Non-SIMD	Improvement Compared with the previous one
Alpha Blend	599	606	612	605.7	112.3%	2.9%
Image Overlay	651	648	643	647.3	161.3%	3.2%

AVX 关键函数运行总时长记录表

可以发现，SIMD 的速率提升非常之明显，但是 SSE2 相较于 MMX 的性能提升、AVX 相较于 SSE2 的性能提升并不那么地显著。这与实际实现时仅类似于循环展开有关。

六、 附件表

1. 代码及说明文档

见附件，或见 https://github.com/mimicji/YUV-Processor

2. Demo

见 http://pan.baidu.com/s/1slMtMP3