西安交通大學

XI'AN JIAOTONG UNIVERSITY

计算机组织与结构实验
利用 MMX 加速图像处理

软件 42

陈铮

2141601026

2016 年 12 月 6 日

目录

第一章　综述

1.1　实验题目

利用 MMX [1]加速图片的渐入渐出效果。

渐入渐出 (Fade In & Fade Out) 是一种常见的图像处理特效。

其原理用一个加权平均公式即可表示：

$$p = p_1 w + p_2(1-w) = w(p_1 - p_2) + p_2, (w \in [0,1]) \tag{1.1}$$

其中，p, p_1, p_2 都是颜色，称 p_1, p_2 为源色，p 为合成色。

这个公式表示如何**按照比例 w 融合两种颜色**。

渐入渐出过程是从 0%(100%) 开始迭代递增 (减) w 时伴随着的 p 的变化过程。在这个过程中，一种源色在合成色中的比例不断增加直至 100%，另一种则不断减少直至 0%。

1.2　实验设计

在计算机中，图像以像素阵列的形式存储，只要能按像素读写图片，就能实现颜色融合。进而配合延时控制，即可实现渐入渐出效果。

不使用 MMX 也可以实现渐入渐出，但 MMX 可以利用**并发**加速**颜色融合**这个过程。

以 24 位位图作为研究对象，因为其色元数据具有 8 位对齐的特征 [2]，其每个像素用 24 bits (3 Bytes) 表示，默认格式为 RGB。红 (Red)、绿 (Green)、蓝 (Blue) 依次用像素的一个 Byte 来表示，取值范围为 0 - 255。

按照算法，p, p_1, p_2 都是一个颜色向量。三种原色的融合要分开做，不能一起做。一个简单的反例是，白色 (0xFFFFFF) 与黑色 (0x000000) 以 50% 的比例融合的时候，如果将颜色直接代入融合公式：

(0xFFFFFF - 0x000000) / 2 + 0x000000 = 0x7FFFFF

得到的是青色 (0x7FFFFF)，这与预期的灰色 (0x7F7F7F) 不符。

正确的做法是将三原色分别代入融合公式：

(0xFF - 0x00) / 2 + 0x00 = 0x7F

这样才能得到灰色 (0x7F7F7F)。

所以颜色融合本质上是**向量操作**，涉及向量的加减以及数乘运算。

MMX 指令集正好就提供了并发进行向量操作的指令, excited!

[1] MMX: 由英特尔开发的一种 SIMD 多媒体指令集，共有 57 条指令。它于 1996 年集成在英特尔奔腾（Pentium）MMX 处理器上，以提高其多媒体数据的处理能力。[Fur97]

[2] MMX 只能正确处理已经对齐的数据，否则需要耗费大量的时间做数据对齐。

第二章　普通的图像操作

2.1　算法设计

同时遍历两个位图的每一个像素的色元，按照公式 1.1 进行颜色合成。
对于 64×64 像素的 24 位位图，需要进行 $3 \times 64 \times 64 = 12288$ 次迭代。

2.2　核心代码实现

在实现时使用了 ATL 的 CImage 类。

代码 2.1: 普通的图像融合

```
1  void ImageFusion(CImage &from, CImage &to, CImage &dest, WORD fading) {
2      int width = dest.GetWidth(), height = dest.GetWidth();
3      BYTE *pFrom = (BYTE*)from.GetPixelAddress(0, height - 1);
4      BYTE *pTo = (BYTE*)to.GetPixelAddress(0, height - 1);
5      BYTE *pDest = (BYTE*)dest.GetPixelAddress(0, height - 1);
6      int t = 3 * height * width;
7      while (t--) {
8          *pDest = (*pFrom * (0x8000 - fading) + *pTo * fading) / 0x8000;
9          pTo++;
10         pFrom++;
11         pDest++;
12     }
13 }
```

第三章　MMX 指令集

3.1　概述

3.1.1　关键字

- Byte: 字节 (8 bits)
- Word: 字 (16 bits)
- Doubleword: 双字 (32 bits)
- Quadword: 四字 (64 bits)
- Double Quadword: 八字 (128 bits)
- Pack: 压缩
- Unpack: 解压
- Saturation: 饱和

3.1.2　饱和

饱和 (Saturation) 是 MMX 的一个很有特点的运算性质，它能在运算结果溢出时直接取上下界的值作为运算结果。

在无符号饱和字运算中，上溢时取结果为 0xFFFF，下溢时取结果为 0x0000。

在带符号饱和字运算中，上溢时取结果为 0x7FFF (32767D)，下溢时取结果为 0x8000 (-32768D)。

以 0x4000 * 0x4000 = 0x10000000 为例，一般的字运算直接截断高位字取低位字得到 0x0000 作为结果，而无符号饱和运算得到 0xFFFF，有符号饱和运算得到 0x7FFF。

3.1.3　压缩

压缩 (打包)/解压缩 (解包) 在 MMX 中是一对很重要的操作，它们与 MMX 的效率密不可分。

以 PACKUSWB [1] (无符号饱和压缩字到字节) 与 PUNPCKLBW [2] 指令为例说明在 MMX 中的压缩/解压缩操作。

指令的格式如下：

```
1 PACKUSWB mm mm/m64
2 PUNPCKLBW mm mm/m32
```

[1] PACKUSWB: Pack with Unsigned Saturation from Word to Byte

[2] PUNPCKLBW: Packed Unpack Low-order Byte to Word

mm 是 MMX 专用的 64 位寄存器格式，有 mm0, mm1, ..., mm7，共 8 个 64 位寄存器。m64 则表示 64 位内存空间格式，m32 则表示 32 位的内存空间格式。

第一个操作数被称为“目的操作数”，只能通过寄存器寻址；第二个操作数被称为“源操作数”，可以通过寄存器寻址也可以通过内存寻址。

压缩是采取“有损压缩”的策略。

PACKUSWB 的机理是先将 64 位操作数视作 4 个字的序列，分别进行无符号饱和压缩，得到 4 个字节；那么 2 个 64 位操作数就可以得到 8 个字节，即一个 64 位数据，然后赋值到目的操作数寄存器中，其中由源操作数得到的 4 个字节在结果中占据高 32 位，而目的操作数压缩得到的 4 个字节在结果中占据低 32 位。

解压缩采取“交叉组合”(Interleave) 的策略。

PUNPCKLBW 的机理是先将 64 位操作数的低 32 位取出，看作 4 个字节；那么 2 个 64 位操作数可以得到 2 组 4 个字节的数据；交叉放置这些字节得到 4 个字，即一个 64 位数据，然后赋值到目的操作数寄存器中，其中由源操作数得到的字节占据字的高位，由目的操作数得到的字节占据字的低位。

举一个具体的例子：

```
1  ; mm0: 7FFF8000FFFF002B, mm1: 002D1121DEADBEEF
2  packuswb mm0, mm1
3  ; 002D|1121|DEAD|BEEF|7FFF|8000|FFFF|002B
4  ;   2D|  FF|  00|  00|  FF|  00|  00|  2B
5  ; mm0: 2DFF0000FF00002B
6  punpcklbw mm0, mm1
7  ; interleave mm1, mm0
8  ; mm1's lower doubleword: DE|AD|BE|EF
9  ; mm0's lower doubleword: FF|00|00|2B
10 ; DE|FF|AD|00|BE|00|EF|2B
11 ; mm0: DEFFAD00BE00EF2B
```

3.2 指令说明

3.2.1 指令清单

拷贝指令

- movd: Move Doubleword 拷贝双字 (32 bits)
- movq: Move Quadword 拷贝四字 (64 bits)

压缩指令

解压缩 (Packed Unpack)

- punpcklbw: Packed Unpack Low-order Bytes to Word
- punpcklwd: Packed Unpack Low-order Words to Doubleword
- punpckldq: Packed Unpack Low-order Doublewords to Quadword
- punpcklqdq: Packed Unpack Low-order Quadwords to Double Quadword
- punpckhbw: Packed Unpack High-order Bytes to Word

- punpckhwd: Packed Unpack High-order Words to Doubleword
- punpckhdq: Packed Unpack High-order Doublewords to Quadword
- punpckhqdq: Packed Unpack High-order Quadwords to Double Quadword

饱和压缩 (Pack with Saturation)
饱和策略分为**带符号饱和**与**无符号饱和**。

- packuswb: Pack with Unsigned Saturation Word to Byte
- packusdw: Pack with Unsigned Saturation Doubleword to Word
- packsswb: Pack with Signed Saturation Word to Byte
- packssdw: Pack with Signed Saturation Doubleword to Word

运算指令

MMX 的运算主要是指“压缩运算”。

一**般来说**，指令名以 p 开头 (packed)，然后接着操作名 (add, sub, mul 等)，然后是运算属性 (u: Unsigned, s: Staturation, h: High-order, l: Low-order 等)，最后是操作数的长度 (b: Byte, w: Word, d: Doubleword, q: Quadword, dq: Double Quadword)。

以 add 为例：

- paddb: Packed Add Bytes
- paddw: Packed Add Words
- paddd: Packed Add Doublewords
- paddq: Packed Add Quadwords
- paddsb: Packed Add with Signed Saturation Bytes
- paddsw: Packed Add with Signed Saturation Words
- paddusb: Packed Add with Unsigned Saturation Bytes
- paddusw: Packed Add with Unsigned Saturation Words

对于 mul 来说：

- pmuldq: Packed Multiply Doublewords to Quadword
- pmulhrsw: Packed Multiply High with Round and Scale Words *
- pmulhuw: Packed Multiply High-order result of Unsigned Words
- pmulhw: Packed Multiply High-order result of signed Words
- pmulld: Packed Multiply Low-order result of signed Doublewords
- pmullw: Packed Multiply Low-order result of signed Words
- pmulludq: Packed Multiply Low-order result of Unsigned Doublewords to Quadwords

第四章　MMX 图像操作

4.1　算法设计

在 24bpp [1]位图中，一个色元恰好占据一个字节，而寄存器的长度数倍于此，一个色元单独使用一个寄存器，于时间、空间都有些浪费，希望能用一个寄存器，一个指令，同时处理多个色元。而且数据正好是按字节对齐的，非常符合 MMX 的优化条件。

由公式 1.1得色元融合需要对色元做一个乘法，因此处理色元乘法需要一个字长的空间，因此 64 位寄存器最多可能一次性处理 4 个色元。

可以一次取出 4 个色元放置在 64 位寄存器的低 32 位，利用之前 3.1.3 提过的解压缩指令，将其分散到 64 位寄存器上的 4 个字中，然后按照公式 1.1 给出的算法进行运算。

设 $\max F = 32767$，对于渐变值 $F \in [0, \max F]$，给定像素色元 $A, B \in [0, 255]$

$$\begin{aligned} \frac{FA + (\max F - F)B}{\max F} &= \frac{2FA + 2(\max F - F)B}{2\max F} \\ &\approx (\frac{FA}{2(\max F + 1)} + \frac{(\max F - F)B}{2(\max F + 1)}) \times 2 \\ &= 2(\mathrm{mulhw}(A, F) + \mathrm{mulhw}(B, \max F - F)) \end{aligned} \tag{4.1}$$

使渐变系数为一个有符号非负整型字的，最大值为 0x7FFF，渐变分为 32768 个级别。

在乘法中，两个字相乘得到一个双字长数据，pmulhw 是有符号乘法取结果双字中的高位字，相当于相乘以后除以 65536，因此在算法公式 4.1 中，要刻意凑出分母 65536，这样才可以替换出 pmulhw。

4.2　代码实现

代码 4.1: MMX 图像融合

```
1  void ImageFusionMMX(CImage &from, CImage &to, CImage &dest, WORD fading) {
2      int width = dest.GetWidth(), height = dest.GetWidth();
3      DWORD *pFrom = (DWORD*)from.GetPixelAddress(0, height - 1);
4      DWORD *pTo = (DWORD*)to.GetPixelAddress(0, height - 1);
5      DWORD *pDest = (DWORD*)dest.GetPixelAddress(0, height - 1);
6      DWORD *pDestEnd = (DWORD*)dest.GetPixelAddress(width - 1, 0);
7      WORD fade1[4], fade2[4];
8      fade1[0] = fade1[1] = fade1[2] = fade1[3] = 0x7fff - fading;
9      fade2[0] = fade2[1] = fade2[2] = fade2[3] = fading;
10     LPWORD fade_ptr1 = fade1, fade_ptr2 = fade2;
11     _asm {
```

[1]24 bits per pixel: 每像素 24 位，即 24 位位图。

```
12          pxor mm7, mm7
13          mov ebx, fade_ptr1
14          movq mm2, [ebx]
15          mov ebx, fade_ptr2
16          movq mm3, [ebx]
17      }
18      while (pDest <= pDestEnd) {
19          _asm {
20              mov ebx, pFrom
21              movd mm0, [ebx]
22
23              mov ebx, pTo
24              movd mm1, [ebx]
25
26              punpcklbw mm0, mm7
27              punpcklbw mm1, mm7
28
29              pmulhw mm0, mm2
30              pmulhw mm1, mm3
31              paddw mm0, mm1
32              paddw mm0, mm0
33              packuswb mm0, mm7
34
35              mov edi, pDest
36              movd[edi], mm0
37          }
38          pFrom++;
39          pTo++;
40          pDest++;
41      }
42      _asm EMMS
43  }
```

4.3 性能基准测试

代码 4.2: 基准测试

```
1  void ImageFusionCPUBenchmark(CImage src1, CImage src2, WORD fading) {
2      CImage temp1, temp2;
3      int width = min(src1.GetWidth(), src2.GetWidth());
4      int height = min(src1.GetHeight(), src2.GetHeight());
5      temp1.Create(width, height, 24);
6      temp2.Create(width, height, 24);
7      while (1) {
8          ImageFusion(src1, src2, image3, fading);
```

```
9           ImageFusionMMX(src1, src2, image4, fading);
10      }
11  }
```

利用 Visual Studio 的性能探查器可以检测 CPU 使用率。

运行测试约 1 分钟后终止检测，得到函数运行独占样本数：

函数名	独占样本数
ImageFusion	45049
ImageFusionMMX	15633

得到近似加速比：

$$S = \frac{45049}{15633} = 2.88^{[2]}$$

[2]实际数据根据不同的实验环境会有所变化，正常范围是 2.0 - 4.0。

第五章　实验总结

一开始有些手足无措，去参考了一下网上老学长的实验报告，只言片语说得我依然懵逼。

但大致了解是要用 Win32 绘制一个界面，依照过去的 Win32 编程经验快速搞了一个框架以后开始研究核心的 MMX 内联汇编部分。

有很多没见过的指令，各种搜索资料。

后来找到了一些很好用的参考资料：

- x86 Instruction Set Reference [rje]

然后结合 Visual Studio 进行单步调试探索一番后弄清楚了各个操作的效果。

最后思考了一番就知道如何设计与普通图像融合等价的 MMX 图像融合算法了。

最后一边写实验报告一边完善理论，顺便尝试优化了算法，使得加速比变高了。

整篇报告除了那个 CImage，没有涉及其他 Win32 编程，是希望能尽可能减少 Win32 在 MMX 中的分量，因为它们本来就不是相关的东西。

5.1　版权声明

本文使用 LaTeX 排版，源代码已开源至 GitHub

URL: https://github.com/zccz14/ComputerOrganizationAndArchitectureMMX

参考文献

[Fur97] B. Furht. *Multimedia Technologies and Applications for the 21st Century: Visions of World Experts*. The Springer International Series in Engineering and Computer Science. Springer US, 1997. ISBN: 9780585287676. URL: https://books.google.com/books?id=2E%5C_lBwAAQBAJ.

[rje] rjeschke. *x86 Instruction Set Reference*. URL: http://x86.renejeschke.de/.

代码清单