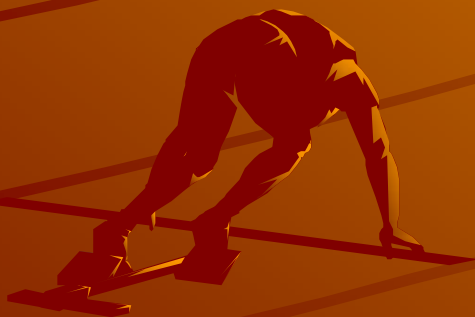


# 让你的软件飞起来

仅以此文献给那些在我的设计工作中  
所有给我提供过帮助的人



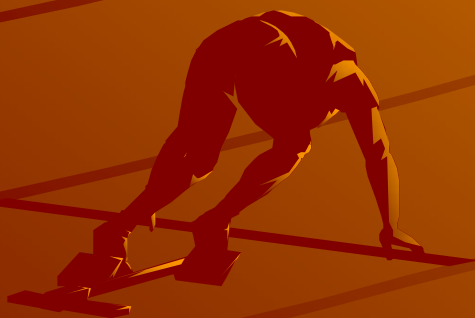
----捷报频传

[conquer\\_2007@163.com](mailto:conquer_2007@163.com)

2005.1.13

# 速度取决于算法

- ◆ 同样的事情，方法不一样，效果也不一样。比如，汽车引擎，可以让你的速度超越马车，却无法超越音速；涡轮引擎，可以轻松超越音障，却无法飞出地球；如果有火箭发动机，就可以到达火星。



# 代码的运算速度取决于以下几个方面

- ◆ 算法本身的复杂度，比如MPEG比JPEG复杂，JPEG比BMP图片的编码复杂。
- ◆ CPU自身的速度和设计架构
- ◆ CPU的总线带宽
- ◆ 您自己的代码的写法



本文主要介绍如何优化您自己的code，实现软件的加速

# 先看看我的需求

我们一个图象模式识别的项目，需要将RGB格式的彩色图像先转换成黑白图像。

图像转换的公式如下：

$$Y = 0.299 * R + 0.587 * G + 0.114 * B;$$

图像尺寸640\*480\*24bit，RGB图像已经按照RGBRGB顺序排列的格式，放在内存里面了。



例如，将这个喷火的战斗机引擎，转换为右边的黑白图片

# 我已经悄悄的完成了第一个优化

以下是输入和输出的定义：

```
#define XSIZE 640
#define YSIZE 480
#define IMGSIZE XSIZE*YSIZE
typedef struct RGB
{
    unsigned char R;
    unsigned char G;
    unsigned char B;
}RGB;
struct RGB in[IMGSIZE] //需要计算的原始数据
Unsigned char out[IMGSIZE] //计算后的结果
```



看得出来优化在  
哪里吗？

# 我已经悄悄的完成了第一个优化

```
#define XSIZE 640
#define YSIZE 480
#define IMGSIZE XSIZE*YSIZE
typedef struct RGB
{
    unsigned char R;
    unsigned char G;
    unsigned char B;
}RGB;
struct RGB in[IMGSIZE] //需要计算的原始数据
Unsigned char out[IMGSIZE] //计算后的结果
```

优化原则：

图像是一个2D数组，我用一个1维数组来存储。

编译器处理1维数组的效率要高过2维数组



# 先写一个代码

$$Y = 0.299 * R + 0.587 * G + 0.114 * B;$$

```
Void calc_lum()
{int I;
    for(i=0;i<IMGSIZE;i++)
    {double r,g,b,y;
    unsigned char yy;
    r=in[i].r; g=in[i].g; b=in[i].b;
    y=0.299*r+0.587*g+0.114*b;
    yy=y; out[i]=yy;
    }
}
```



这大概是能想得出来的最简单的写法了，实在看不出有什么毛病，好了，编译一下跑一跑吧。

# 第一次试跑

```
Void calc_lum()
{int i;
  for(i=0;i<IMGSIZE;i++)
  {double r,g,b,y;
   unsigned char yy;
   r=in[i].r; g=in[i].g; b=in[i].b;
   y=0.299*r+0.587*g+0.114*b;
   yy=y; out[i]=yy;
  }
}
```



这个代码分别用VC6.0和GCC编译，生成2个版本，分别在PC上和我的embedded system上面跑。

速度多少？说出来吓死你！



# 第一次试跑的成绩

在PC上，由于存在硬件浮点处理器，CPU频率也够高，计算速度为20秒

我的embedded system，没有以上2个优势，浮点操作被编译器分解成了整数运算，运算速度为120秒左右

这只是一副图像的运算速度！！



# 去掉浮点运算

上面这个代码还没有跑，我已经知道会很慢了，因为这其中有大量的浮点运算。只要能不用浮点运算，一定能快很多。

$$Y = 0.299 * R + 0.587 * G + 0.114 * B;$$

那这个公式怎么能用定点的整数运算替代呢？



# 0.299 \* R 可以如何化简？

$$Y = 0.299 * R + 0.587 * G + 0.114 * B;$$

$$Y = D + E + F;$$

$$D = 0.299 * R;$$

$$E = 0.587 * G;$$

$$F = 0.114 * B;$$

我们就先简化算式D吧！

RGB的取值范围都是0~255,都是整数，只是这个系数比较麻烦，不过这个系数可以表示为：

$$0.299 = 299/1000$$

$$\text{所以 } D = (R * 299) / 1000$$

$$Y = (R * 299) / 1000 + (G * 587) / 1000 + (B * 114) / 1000$$

再简化为：

$$Y = (R * 299 + G * 587 + B * 114) / 1000$$

这一下，能快多少呢？



# 化简后的成绩

Embedded system  
上的速度45秒

PC上的速度2秒



# 0.299 \* R 进一步化简

$$Y = 0.299 * R + 0.587 * G + 0.114 * B$$

$$Y = (R * 299 + G * 587 + B * 114) / 1000$$

这个式子好像还有点复杂，可以再砍掉一个除法运算。

前面的算式D可以这样写：

$$0.299 = 299 / 1000 = 1224 / 4096$$

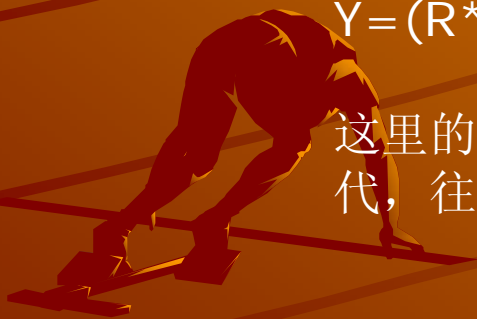
$$\text{所以 } D = (R * 1224) / 4096$$

$$Y = (R * 1224) / 1000 + (G * 2404) / 4096 + (B * 467) / 4096$$

再简化为：

$$Y = (R * 1224 + G * 2404 + B * 467) / 4096$$

这里的/4096除法，因为它是2的N次方，所以可以用移位操作替代，往右移位12bit就是把某个数除以4096了



# 0.299 \* R进一步化简

$$Y = (R * 1224 + G * 2404 + B * 467) / 4096$$

```
Void calc_lum()
```

```
{int I;
```

```
    for(i=0;i<IMGSIZE;i++)
```

```
    {int r,g,b,y;
```

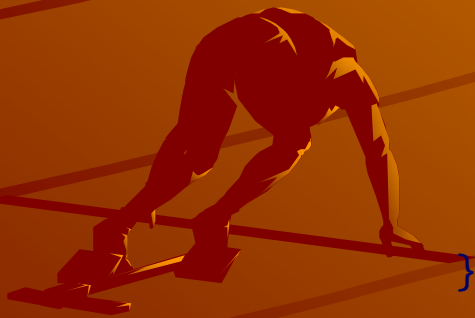
```
    r=1224*in[i].r;  g=2404*in[i].g;  b=467*in[i].b;
```

```
    y=r+g+b;
```

```
    y=y>>12; //这里去掉了除法运算
```

```
    out[i]=y;
```

```
}
```



这个代码编译后，又快了20%

# 还是太慢！

虽然快了不少，还是太慢了一些，  
20秒处理一幅图像，地球人都不能  
接受！



但是目前这个式子好像优化到极限了，要想突破音障，只能拆掉活塞发动机，安装涡轮引擎！

# 仔细端详一下这个式子！

$$Y = 0.299 * R + 0.587 * G + 0.114 * B;$$

$$Y = D + E + F;$$

$$D = 0.299 * R;$$

$$E = 0.587 * G;$$

$$F = 0.114 * B;$$

仔细端详一下这个式子！

RGB的取值有文章可做，RGB的取值永远都大于等于0，小于等于255，我们能不能将D, E, F都预先计算好呢？然后用查表算法计算呢？

我们使用3个数组分别存放DEF的256种可能的取值，然后。。。





# 查表数组初始化

```
Y = 0.299 * R + 0.587 * G + 0.114 * B;  
Y=D+E+F;  
D=0.299*R;  
E=0.587*G;  
F=0.114*B;
```

```
Int D[256], E[256], F[256]; //查表数组
```

```
Void table_init()
```

```
{int I;
```

```
for(i=0;i<256;i++)
```

```
{D[i]=i*1224; D[i]=D[i]>>12;
```

```
E[i]=i*2404; E[i]=E[i]>>12;
```

```
F[i]=i*467; F[i]=F[i]>>12;
```

```
}
```


```
}
```



# 使用查表数组

```
Y = 0.299 * R + 0.587 * G + 0.114 * B;  
Y=D+E+F;  
D=0.299*R;  
E=0.587*G;  
F=0.114*B;
```

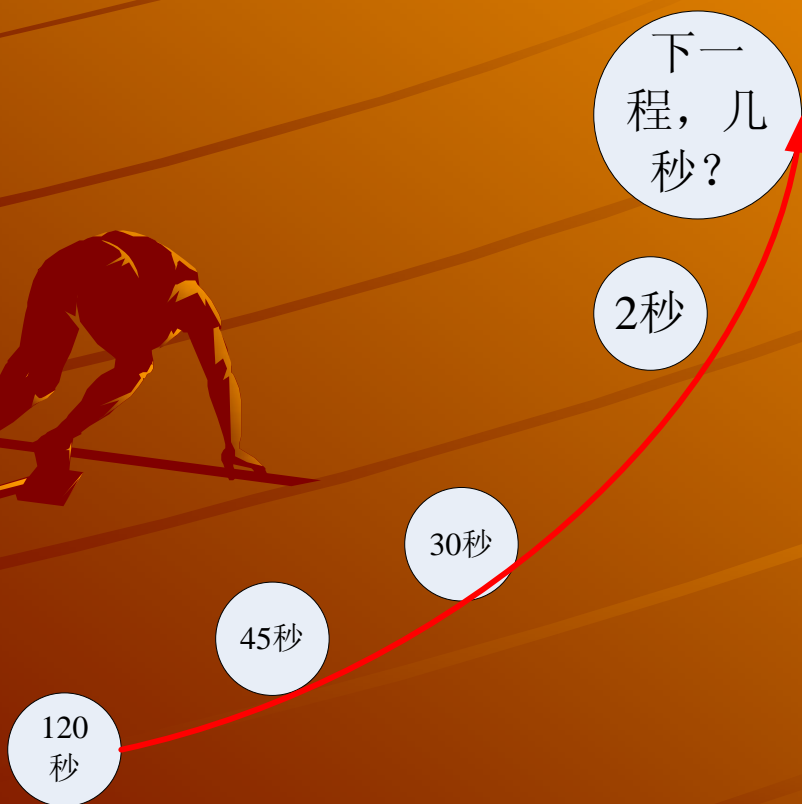
```
Void calc_lum()  
{int i;  
    for(i=0;i<IMGSIZE;i++)  
    {int r,g,b,y;  
      r=D[in[i].r]; g=E[in[i].g]; b=F[in[i].b]; //查表  
      y=r+g+b;  
      out[i]=y;  
    }  
}
```



# 突破音障!

这一次的成绩把我吓出一身冷汗，执行时间居然从30秒一下提高到了2秒！在PC上测试这段代码，眼皮还没眨一下，代码就执行完了。

一下提高15倍，爽不爽？

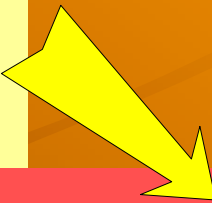


## 还能再快吗？

# 踩足油门，向2马赫进军！

很多embedded system 的32bitCPU，都至少有2个ALU，能不能让2个ALU都跑起来？

```
Void calc_lum()
{int I;
    for(i=0;i<IMGSIZE;i++)
    {int r,g,b,y;
    r=D[in[i].r]; g=E[in[i].g]; b=F[in[i].b]; //查表
    y=r+g+b;
    out[i]=y;
    }
}
```



```
Void calc_lum()
{int I;
    for(i=0;i<IMGSIZE;i+=2) //一次并行处理2个数据
    {int r,g,b,y, r1,g1,b1,y1;
    r=D[in[i].r]; g=E[in[i].g]; b=F[in[i].b]; //查表
    y=r+g+b;
    out[i]=y;
    r1=D[in[i+1].r]; g1=E[in[i+1].g]; b1=F[in[i+1].b]; //查表
    y1=r1+g1+b1;
    out[i+1]=y1;
    }
}
```



# 并行计算

```
Void calc_lum()  
{int I;
```

```
for(i=0;i<IMGSIZE;i+=2) //一次并行处理2个数据
```

```
{int r,g,b,y, r1,g1,b1,y1;
```

```
r=D[in[i].r]; g=E[in[i].g]; b=F[in[i].b]; //查表
```

```
y=r+g+b;
```

```
out[i]=y;
```

```
r1=D[in[i+1].r]; g1=E[in[i+1].g]; b1=F[in[i+1].b]; //查表
```

```
y1=r1+g1+b1;
```

```
out[i+1]=y1;
```

```
}
```

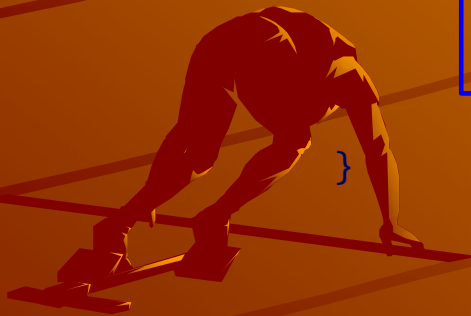
```
}
```

一次并行处理2组数据  
所以这里一次加2

给第一个ALU执行

给第二个ALU执行

2个ALU处理的数据不能有数据依赖，也就是说：  
某个ALU的输入条件不能是别的ALU的输出，这样  
才可以并行



这一次的成绩是：



1S

# 加足燃料，进军3马赫！

```
Int D[256], E[256], F[256]; //查表数组
Void table_init()
{int I;
    for(i=0;i<256;i++)
    {D[i]=i*1224; D[i]=D[i]>>12;
    E[i]=i*2404; E[i]=E[i]>>12;
    F[i]=i*467; F[i]=F[i]>>12;
    }
}
```

看看这个代码，

到这里，似乎已经足够快了，但是我们反复实验，发现，还有办法再快！  
可以将

```
Int D[256], E[256], F[256]; //查表数组
```

更改为：

```
Unsigned short D[256], E[256], F[256]; //查表数组
```

这是因为编译器处理int类型和处理unsigned short类型的效率不一样



# 再改动一下

```
Unsigned short D[256], E[256], F[256]; //查表数组
Inline Void calc_lum()
{int i;
    for(i=0;i<IMGSIZE;i+=2) //一次并行处理2个数据
    {int r,g,b,y, r1,g1,b1,y1;
    r=D[in[i].r]; g=E[in[i].g]; b=F[in[i].b]; //查表
    y=r+g+b;
    out[i]=y;
    r1=D[in[i+1].r]; g1=E[in[i+1].g]; b1=F[in[i+1].b]; //查表
    y1=r1+g1+b1;
    out[i+1]=y1;
    }
}
```



将函数声明为inline,这样编译器就会将其嵌入到母函数中,可以减少CPU调用子函数所产生的开销



这2个小小的改进带来的效益！

**SPEED=**

**0.5S**



现在，我们已经达到了客户的要求！

# 其实，我们还可以飞出地球的！

如果加上以下措施，应该还可以更快：

- 把查表的数据放置在CPU的高速数据CACHE里面
- 把函数`calc_lum()`用汇编语言来写

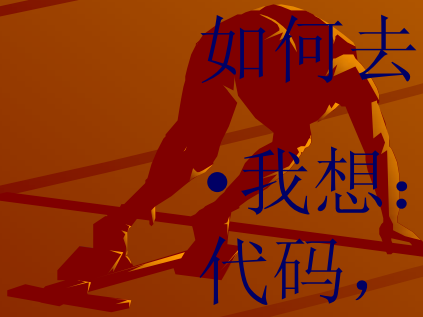



# 其实，CPU的潜力是很大的

- 不要抱怨你的CPU，记住一句话：“只要功率足够，砖头都能飞！”

- 同样的需求，写法不一样，速度可以从120秒变化为0.5秒，说明CPU的潜能是很大的！看你如何去挖掘。

- 我想：要是Microsoft的工程师都像我这样优化代码，我大概就可以用486跑windows XP了！



如果您觉得本文足够的精彩，请  
发一个mail问候我一下：（想扔  
鸡蛋的就免了 ）

[conquer\\_2007@163.com](mailto:conquer_2007@163.com)



您的鼓励是我努力工作的最大动力，  
也是我不竭的创新动力！

# The End



----捷报频传

[conquer\\_2007@163.com](mailto:conquer_2007@163.com)

2005.1.13