# 基于 Trimedia DSP 的 H.264 解码算法优化

林 冰 冯 艳 李学明
(北京邮电大学信息工程学院,北京 100876)

**摘 要** H.264 是最新的视频编码标准,具有非常优良的编码性能,但它的算法复杂度也很高,很难满足实时应用的需要。论文详细分析了影响 H.264 解码速度的因素,提出了基于 Trimedia DSP 平台的优化方案。该方案通过缩减不必要的判断、避免频繁的内存访问、优化内存的分配与使用、合理使用循环展开以及采用 DSP 专用指令等方法来提高 H.264 解码算法的运算速度。测试结果表明:优化后的代码运行速度平均提高了 8 倍,在主频为 200MHz 的 Trimedia DSP 上能实时解码 CIF 格式的 H.264 基本码流。

**关键词** H.264 代码优化 循环展开 指令级并行

**文章编号** 1002-8331-(2005)31-0041-05 **文献标识码** A **中图分类号** TP391

# H.264 Decoder Optimization for Trimedia DSP

**Lin Bing Feng Yan Li Xueming**
(School of Information Engineering of BUPT, Beijing 100876)

**Abstract**: H.264 is the latest video coding standard with high coding performance.But its computational complexity is also too high to be used in real time environment.In this paper,key factors,which have great influence to the decoding speed,have been examined.Then an optimization solution designed for Trimedia DSP platform is presented to speed up decoding of H.264.Many techniques are covered in this solution,including:reducing redundant flow control,minimizing memory access,taking care of the allocation and usage of memory,properly performing loop unrolling and utilizing DSP-specific instructions to improve Instruction Level Parallelism(ILP).Simulation results show this solution can accelerate the decoding speed by an average factor of about 8 times.Experimental result on Trimedia DSP with an operation frequency of 200MHz demonstrates that optimized code can decode CIF-format H.264 stream in real time.

**Keywords**: H.264,code optimization,loop unrolling,Instruction Level Parallelism

## 1 引言

H.264 是国际上最新的视频编码标准,它吸收了现有标准的优点,并采用了诸如:帧内预测、整数余弦变换、多模式多参考帧运动估计、自适应二进制算数编码等新技术,大大提高了视频编码的效率[1,2]。然而,这些新的技术在提高编码质量的同时,也极大地增加了算法的复杂度。经过对比,H.264 baseline 解码器的算法复杂度为现有的 H.263 baseline 解码器的 2.5 倍[3],无法满足视频会议、可视电话等实时应用的需要。在这种条件下,对 H.264 的算法进行优化,使之满足实时应用的需要成为目前的研究热点[4,5]。

作为新一代的视频编码标准,H.264 针对不同的应用定义了 3 个不同的框架(Profile):基本框架(Baseline Profile),主框架(Main Profile)和扩展框架(Extended Profile)[1]。图 1 给出了不同框架所包含的主要功能模块。

Baseline Profile 主要面向实时应用,包含了一些复杂度相对较低的编码工具,同时还支持一些低时延的容错工具。Main Profile 的主要目的是提高编码效率,因此它在编码时引入了 B 帧,内容自适应的二进制算术编码(Context Adaptive Binary Arithmetic Coding,CABAC)等模块。Extended Profile 为了兼顾编码效率、容错和重同步的要求,加入了相应的功能模块。本文主要关注 H.264 在实时通信中的应用,因此主要研究 Baseline Profile 的优化问题。为此我们首先对 Baseline Profile 解码算法

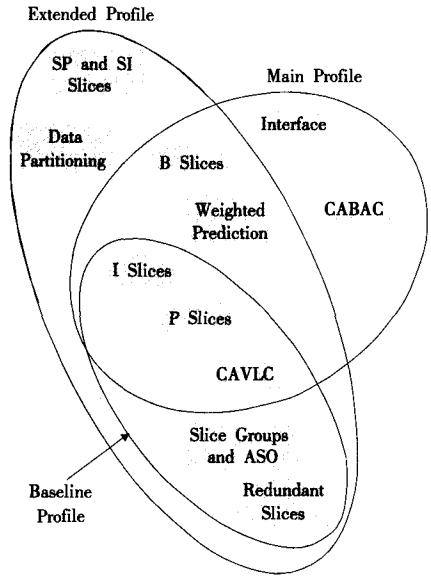


**图 1 三种 H.264 框架所包含的主要功能模块**

**基金项目**:国家自然科学基金资助(编号:60172050)

**作者简介**:林冰,男,硕士,主要研究方向为嵌入式应用,多媒体通信。冯艳,女,硕士,主要研究方向为视频编码,多媒体通信。李学明,男,教授,主要研究方向为视频编码,图像处理,移动多媒体应用。

进行了详细分析，了解影响算法复杂度的关键环节，在此基础上提出了提高运算速度的优化方案，并进行了测试。下面将首先介绍影响 H.264 解码速度的因素。

## 2 影响解码速度的主要因素

为了定量分析解码算法中影响运算速度的关键因素，我们用 Philips 公司为 Trimedia 系列 DSP 芯片开发的仿真分析工具 Tmsim 和 Tmprof 对 H.264 解码算法进行了仿真测试。Tmsim 可以模拟算法在 DSP 上实际运行的情况，并生成相应的运行文件。Tmprof 是一个统计软件，它根据 Tmsim 运行的结果，以函数为单位统计出每个函数执行的次数以及总的指令周期[6]。

在测试过程中我们采用 JM8.1a 版本的参考实现软件对三个标准测试序列(Foreman、Missa、Mobile)进行编码[7]，得到相应的 H.264 码流。在编码过程中，设定运动搜索范围为 16，5 个预测参考帧，I 帧和 P 帧图像的量化参数固定为 28，熵编码采用 CAVLC。

表 1 给出标准测试序列编码的帧速率以及编码后输出的比特速率。表 2 给出了解码算法解码这三个测试码流的运行结果(只给出了耗时最多的四个模块)。从表 2 可以看出，1/4 像素插值、单个宏块熵解码、整数余弦变换和去除块效应滤波器是 H.264 解码过程中耗时最多的几个模块。对于不同的测试序列，不同模块所耗费的比例有一定的差异，比如：1/4 像素插值在解码 Foreman 序列时能占到总运算量的 25%，而在 Missa 中却只占 15%。尽管如此，这几个模块的排列次序没有特别大的变化，并且在总的解码时间所占比例都超过 40%。由此不难看出，为提高算法运行速度，必须对这些耗时特别多的模块进行优化。

**表 1 测试序列编码参数**

| 序列名称 | 序列长度/帧 | 图像格式 | 编码帧速率/(帧/s) | 输出比特率/(kb/s) |
|---|---|---|---|---|
| Foreman | 300 | QCIF | 10 | 36.39 |
| Missa | 99 | CIF | 15 | 70.12 |
| Mobile | 300 | CIF | 30 | 1 669.03 |

**表 2 测试序列中最耗时的模块**

| 序列名称 | 功能模块 | 指令周期总数 | 百分比/% | 合计/% |
|---|---|---|---|---|
| Foreman | 1/4 像素插值 | 376 002 171 | 24.97 | 49.98 |
| | 单个宏块熵解码 | 182 287 177 | 12.11 | |
| | 整数余弦反变换 | 146 137 172 | 9.70 | |
| | 去除块效应滤波器 | 51 600 598 | 3.20 | |
| Missa | 1/4 像素插值 | 708 345 014 | 15.74 | 46.37 |
| | 单个宏块熵解码 | 677 228 891 | 15.04 | |
| | 整数余弦反变换 | 557 647 239 | 12.39 | |
| | 去除块效应滤波器 | 144 242 543 | 3.20 | |
| Mobile | 1/4 像素插值 | 1 640 379 240 | 20.58 | 40.24 |
| | 单个宏块熵解码 | 757 581 479 | 9.50 | |
| | 整数余弦反变换 | 577 954 776 | 7.25 | |
| | 去除块效应滤波器 | 231 629 395 | 2.91 | |

下面我们以 1/4 像素插值为例，详细讨论 H.264 解码算法的优化。整个优化主要从两个方面来进行。首先是根据 H.264 解码算法的特点，缩减不必要的判断，合理分配和使用内存，尽可能减少内存访问次数，进而提高运行效率。另一方面，我们根据 Trimedia DSP 的特点对代码进行合理组织，帮助编译器生成并行度高的代码；同时，尽可能使用DSP 的专用指令进一步提高运算速度。

## 3 H.264 解码算法的优化

### 3.1 1/4 像素插值的实现方法

1/4 像素插值模块是解码算法中耗时最多的模块，也是优化的重点，其主要功能是：根据运动向量在参考图像中获得相应的预测图像块，以便用于运动补偿。由于 H.264 中的像素精度可以达到 1/4 像素，因此需要根据整数像点的值计算出分数位置的像素值。

在 H.264 中，1/2 像素点的值需要通过六抽头滤波器[1，-5，20，20，-5，1]对原始整像素点插值得到[3]；1/4 像素点的值需要在 1/2 像素值和整像素值的基础上使用双线性滤波器[1，1]后得到。图 2 是非整数像素位置的示意图。其中大写字母表示整数位置的像素，小写字母表示分数位置的像素。

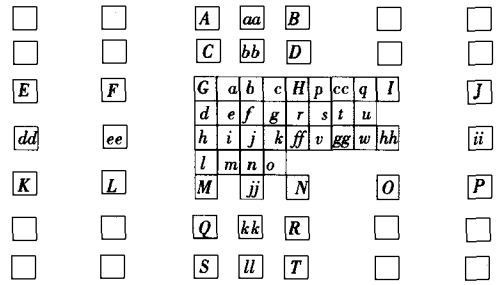


**图 2 1/2、1/4 像素位置示意图**

举例来说，为了得到 1/2 像素位置 $b$ 和 $h$ 的值，需要用 6 阶滤波器在水平方向和垂直方向对邻近像素进行滤波得到中间结果 $b_1$ 和 $h_1$，然后除以 32 来得到。即：

$$b_1=E-5F+20G+20H-5I+J \quad (1)$$

$$h_1=A-5C+20G+20M-5Q+S \quad (2)$$

$$b=(b_1+16)>>5 \quad (3)$$

$$h=(h_1+16)>>5 \quad (4)$$

最后还需要对 $b$ 和 $h$ 的值进行限幅，使值在[0，255]之间。

1/4 像素点 $a$、$c$、$i$、$k$、$d$、$l$、$f$ 和 $n$ 的值，需要根据水平方向或垂直方向上与之邻近的整像素点和 1/2 像素点或两个 1/2 像素点的值，通过求平均值来获得，比如：$a$ 的值等于整像素点 $G$ 和 1/2 像素点 $b$ 的平均，$f$ 的值等于两个 1/2 像素点 $b$ 和 $j$ 的平均，即：

$$a=(G+b)>>1，f=(b+j)>>1 \quad (5)$$

1/4 像素点 $e$、$g$、$m$ 和 $o$ 用对角线方向邻近的两个 1/2 像素点做平均得到，比如：

$$e=(b+h)>>1 \quad (6)$$

### 3.2 减少不必要的判断分支

我们仔细分析了 JM 的实现代码，发现在插值时，由于要进行滤波，JM 的实现代码对于每一个像素点，都要判断它是否超出图像的上、下、左、右四个边界，并根据不同的情况做不同的处理。事实上，这种判断对于处于边界上的点而言是必须的，但对于绝大部分处于图像内部的像素点而言，这些判断完全是多余的，并且会耗费大量的计算时间。为此，朱冬冬等人提出图 3(b)所示的方案[8]。

该方案的基本思路是：由于插值时需要对超出边界的点使

用边界值进行代替,因此在插值之前我们可以在图像四周补上16个像素的"边",这样在滤波时根本不需要任何的边界判断就可以直接进行滤波处理,得到1/4像素的值。但该方案也有不足之处:需要占用更多的内存,且不利于数据的批量操作。对于CIF、QCIF格式的图像而言,"补"上这样的边后,图像存储容量分别会增加10.4%和21.2%。更为重要的是:由于图像数据对应的内存空间是不连续的,因此解码后的图像在用于预测1/4像素值之前,需要以行为单位拷贝到内存的指定区域,并对边界像素进行赋值操作。实验结果表明,在Trimedia DSP上进行这类操作的效率比较低,在一定程度上抵消了优化的效果。为此我们提出了图3(a)所示的解决方案。

该方案将整个参考帧分为9个不同的区域,对于每个运动向量,我们首先根据它的位置判断它属于哪个区域,然后调用针对不同的区域的插值函数。落入中间区域的运动向量不存在超出参考帧边界的问题,因此完全没有进行边界判断的必要;落入上下左右四块的运动向量分别只需要判断一次是否超出了对应的边;而落入四角区域的运动向量也只需要进行两次相应的边界判断。测试结果显示:对于QCIF格式的序列,大约90%的运动向量位于中心区域,不需要做任何判断;而对于CIF以及4CIF格式的序列这个比例更大,优化效果非常显著。

| | | |
|---|---|---|
| 判断两次 | 只判断一次,是否超出上边界 | 判断两次 |
| 只判断一次,是否超出左边界 | 不需要判断 | 只判断一次,是否超出右边界 |
| 判断两次 | 只判断一次,是否超出下边界 | 判断两次 |

(a)

| | | |
|---|---|---|
| 同左上角的值16×16 | 与上边界值相同,向上扩展16个像素 | 同右上角的值16×16 |
| 与左边界值相同,向左扩展16个像素 | 原来的参考帧 | 与右边界值相同,向右扩展16个像素 |
| 同左下角的值16×16 | 与下边界值相同,向上扩展16个像素 | 同右下角的值16×16 |

(b)

图3 减少边界判断的两种方案

## 3.3 数据本地化,减少读写内存的次数

DSP/CPU在进行数据运算时,如果操作数据在寄存器中,DSP/CPU可直接运算,速度最快;如果数据在Cache中,由于Cache是片内存储,因此速率也比较快;如果数据存储在外部内存中时,DSP/CPU需要通过外部总线去读取数据,这需要消耗大量的指令周期。从这个意义上说,尽可能地减少内存操作次数是提高运行速度的重要手段之一。由于一般DSP/CPU的片内数据Cache通常只有十几KB,不能保存完整的一幅图像,这样在运算中就可能需要多次访问内存获取需要的数据。如果每次操作都需要访问内存中的数据,显然会浪费很多的指令周期。为此可以把当前需要的小块数据一次性读入缓存,待所有运算结束后再将结果一次性写入内存,从而避免频繁的内存操作,提高运算速度。

前面我们提到,在计算1/2像素值时需要使用相邻6个整数像素点的值,如图4所示。图中4个$a$表示4个1/2像素点,分别用$a_0,a_1,a_2,a_3$表示,9个$x$表示9个整数像素点,分别用$x_0$至$x_8$表示。计算$a_0$需要使用$x_0$至$x_5$,计算$a_1$需要使用$x_1$至$x_6$,依此类推。在JM的实现方式中,对每一个1/2像素点都读取了6个相邻整数像素的值,即:需要做6次内存读操作。事实上,我们发现:计算$a_0$所需要的整数像素点与计算$a_1$所需要的整数像素点有5个是完全相同的。计算图4所示的4个1/2像素点时,总共只需要9个整数像素的值。这就提示我们,如果先一次性将这9个点的值读入DSP的寄存器或缓存中,然后再计算4个1/2像素的值,这样总共只需要9次内存访问,比JM的实现方法节约4×6-9=15次内存读取操作。

在具体实现时,我们以4×4的块为基本单位,并将4×4的块看成四行。为减少内存访问次数,我们在堆栈中临时分配一个4×9数组,并在计算开始前图像数据从内存读入到本地数组中,内存访问次数为4×9=36。接下来,对于每一行我们均采用上面的方法来处理,计算出4个1/2像素的值。事实上,计算出这16个1/2像素的值JM的方法需要4×4×6=96次内存访问,采用这种方法后每个块可节约60次内存访问。

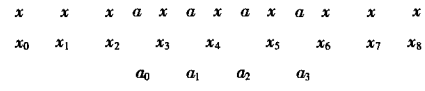


图4 1/2像素插值需要的9个整数像素点

上面的例子还仅仅计算了水平方向1/2像素的值,如果需要计算1/4像素的值,改进后的方法可以利用当前计算出的1/2像素的值和堆栈数组中的整数像素值进行计算,根本不需要再访问内存。而JM的实现代码则需要二次读入整像素的值。由此可见,通过数据的本地化,可以减少不必要的内存访问,提高运算速度。

## 3.4 采用紧凑的数据结构

在JM的实现代码中,大量使用了结构体来保存有关的参数。由于解码算法涉及几十个基本参数,导致几个常用的数据结构体都很庞大,占用了很大的内存空间。更为重要的是:这些结构体经常作为参数在不同的函数间传递。由于结构体比较大,当某个函数运行结束后,系统为保证后续代码的运行,通常需要从缓存中删除结构体,增大了结构体的缓存失效概率,使得系统需要经常性地对内存中的结构体进行读写,降低了运行速度。为此,我们对常用的结构体进行了调整:

(1)仅保留密切相关的数据。比如:所有与运动矢量有关的参数保存在一个结构体中,其它无关的参数则用另外一个结构体来保存,这样在进行与运动矢量有关的操作时无需读入无关的数据。

(2)尽可能采用最紧凑的存储方式,比如:图像数据的范围在0~255之间,因此完全可以用unsigned char类型来表示,从而大幅度减小结构体的尺寸。

## 3.5 优化内存管理

除内存读写需要耗费较多的时间之外,内存的分配和管理也会影响算法的运行速度。在JM的实现代码中,有很多地方

使了高维数组,而且数组在内存中也不是连续分配的,因此不利于使用 memcpy 和 memset 进行批量内存操作,为此,我们修改了 JM 中的内存分配方法,保证数组分配到的内存在物理上是连续的,便于批量操作(参见图 5)。

```
int * *p;//p[m][n]

p=(int * *)malloc(m * sizeof(int * ));
for(int i=0;i<m;i++)
  p[i]=(int * )malloc(n * sizeof(int));
```

```
int * *p;//p[m][n]

p=(int * *)malloc(m * sizeof(int * ));
int * temp=(int * )malloc(m * n * sizeof
          (int));
for(int i=0;i<m;i++)
  p[i]=temp+n * i;
```

图 5 分配连续内存便于批量操作

其次,在 JM 的实现代码中,通常是在一个函数开始的地方先释放内存,然后重新分配内存,这样函数每执行一次就会有一次内存的分配与释放。对于那些经常执行的函数而言,这种方法显然会影响代码运行的速度,因为内存的分配和释放需要消耗很多的指令周期。为此,我们将在程序开始时就为函数分配所需的内存,在函数体内不再重复地分配和释放内存,直到程序结束时才释放内存(参见图 6)。

```
void func(int * p){
 ..................
  if(p! =null)free(p);
  p=(int * )malloc(m * sizeof(int));
 ..................
};
```

```
void main(){
 ..... .
   int * p=malloc(m * sizeof(int));
   func(p);
 .........
   free(p);
};
void func(int * p){
 ..................
};
```

图 6 避免频繁分配内存与释放内存

另外,我们对在函数体内部多次使用的常量数组添加了 static 修饰符,数组只在函数首次被调用执行时初始化,函数结束时不释放,降低了系统开销。图 7 给出了一个这方面的例子。在这个例子中,数组 a[10][4]保存的是一些运算需要的常数。在左边的代码中,函数每次执行的时候,系统均需要为其分配空间,并赋值。而优化之后,我们添加了 static 修饰符,这样只在函数第一次运行时才需要分配空间和赋值,此后就不需要赋值,从而节约大量的时间。

```
void func(int * p){
int a[10][4]=
 {{10,20,30,40},
  {1,2,3,7},
  ..... };

  int x=a[0][3];
  .....
};
```

```
void func(int * p){
 static int a[10][4]=
 {{10,20,30,40},
  {1,2,3,7},
  ..... };

  int x=a[0][3];
  .....
};
```

图 7 使用静态参数来保存常数

## 4 基于 Trimedia DSP 的代码优化

Trimedia DSP 是 Philips 公司推出的多媒体处理器。我们的优化是在 Trimedia PNX1502 处理器芯片上进行的。该芯片的主频为 200MHz,片内有 32K 字节的指令缓存和 16K 字节的数据缓存。Trimedia DSP 的微处理器内核采用了超长指令字(VLIW,Very Long Instruction Word)结构,一个指令周期最多并行完成 5 次算术操作。根据 Trimedia DSP 的这些特点,我们对 H.264 解码算法进行了有针对性的优化。

### 4.1 合理使用循环展开技术,提高代码并行度

循环是程序设计中的常用技术,它可以使代码清晰,可读性强。在每次循环体内的操作完成后程序都需要判断循环条件是否满足,进而决定是否进行下一次循环。我们知道,判断操作通常比普通加法操作耗费更多的指令周期。循环的次数越多,用于循环条件判断的周期也就越多。循环展开(loop unrolling)作为一种常用的代码优化方法,其目的就是减少循环判决的次数,从而提高运行速度。图 8 是一个循环展开的例子:

```
for(int i=0;i<1000;i++) {
  x[i]=y[i]+z[i];
}
```

```
for(int i=0;i<1000;i+=4) {
  x[i]=y[i]+z[i];
  x[i+1]=y[i+1]+z[i+1];
  x[i+2]=y[i+2]+z[i+2];
  x[i+3]=y[i+3]+z[i+3];
};
```

图 8 循环展开示例

在这个例子中,左边的循环需要执行 1 000 次,也就意味着需要进行 1 000 次条件判断。右边的例子中,通过循环展开,循环只需要执行 250 次,判决的次数只有原来的 1/4。当然,循环展开后代码长度会增加,这可能会导致指令缓存失效概率增加,进而抵消由于循环展开所带来的优势。

事实上,在 Trimedia DSP 平台上进行循环展开不仅可以减少循环带来的开销,更为重要的是:循环展开后的代码有利于编译器生成更高效的代码[4]。以图 8 为例,在没有进行循环展开之前,循环体内有 1 次加法,需要 1 个指令周期来计算。进行循环展开后,循环体内尽管有 4 次加法,但这 4 次加法所使用的操作数之间没有依赖关系。比如:计算 x[i]需要使用 y[i]和 z[i],而计算 x[i+1]需要的操作数是 y[i+1]和 z[i+1],与前面一次加法的结果 x[i]无关,这表明计算 x[i],x[i+1]涉及的加法操作彼此无关,可同时进行。由于 Trimedia DSP 可以在 1 个指令周期内完成最多 5 次操作,这样循环体内的 4 次加法操作可以在 1 个指令内完成。也就是说,展开后虽然循环体内有 4 条语句,但经过编译后仍然可以用 1 条指令来完成所有的操作。在一般情况下 Trimedia DSP 的编译器不会生成这样高效的代码,但循环展开后的代码由于操作相同,并且彼此无关,因而便于编译器生成高效的代码。

### 4.2 尽可能使用 DSP 专用指令

由于 Trimedia DSP 是专门用于多媒体处理的 DSP,它提供了很多与多媒体相关的特殊指令。比如:在图像处理中我们经常需要将处理后的图像值限制在[0,255]的范围之内。一般的情况下我们需要作两次比较来实现限幅。在 Trimedia DSP 中,它有专门的指令可在 1 个周期内完成限幅操作。因此使用 DSP 的专用指令或函数是提高运算速度的重要手段。

在上面提到的 1/4 像素插值中,插值需要完成式(1)(2)所示的滤波操作,而一次滤波操作需要进行 4 次乘法和 5 次加法。Trimedia 提供的 ifir8ui 指令可以一次对 2 组各 4 字节的数据对应相乘相加(见图 9)[6]。从图 9 中可以看到,寄存器 rsrc1

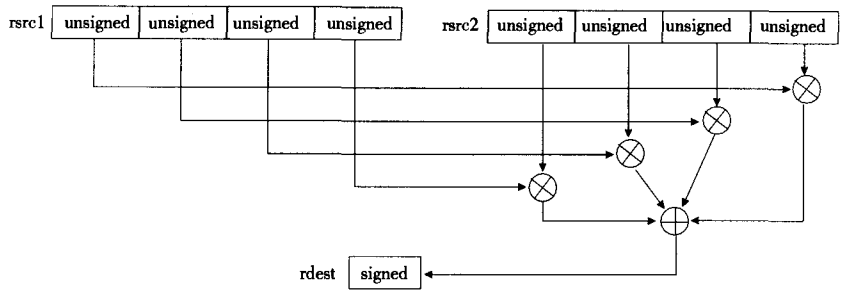


**图 9 Trimedia ifir8ui 指令示意图**

中保存了 4 个无符号数，每个数占用 1 个字节，而 rsrc2 中有 4 个有符号数，函数 ifir8ui(rsrc1,rsrc2)可以在一个指令周期内完成 4 次乘法和 3 次加法。这就提示我们可以用它来快速插值。比如，将式(1)中 $F$、$G$、$H$、$I$ 四个像素的值放入寄存器 rsrc1 中，将系数-5、20、20、-5 放入 rsrc2 中，然后调用 ifir8ui 指令就可以计算出 $-5F+20G+20H-5I$ 的值，只要再加上 $E$、$J$ 的值就可以得到滤波的结果。当然，这里有一个问题：如何将 $F$、$G$、$H$、$I$ 四个像素的值放入一个寄存器？事实上，Trimedia 是 32 位的 DSP，每次可读入 32 比特的数据，如果图像数据刚好在 32 位的边界上，那么一次读入的 32 位数据就对应 4 个像素的值。如果这种条件不满足，Trimedia 还提供了专门的数据打包指令，比如 packbytes 和 pack16lsb，可以字节或字为单位将数据组合成各种形式。在我们的实现代码中就使用了这类指令。

## 5 测试结果

为验证优化的效果，我们利用 Tmsim 和 Tmprof 软件对表 1 所示的 3 个 H.264 码流进行了解码测试，表 3 比较了几个关键函数优化前后所需要的指令周期数，最后一列“节省比例”的计算公式为：

$$\text{节省比例}=\frac{\text{优化前指令周期}-\text{优化后指令周期}}{\text{优化前指令周期}}\times 100\%$$

从表 3 可以看出：优化后的 1/4 像素插值、单个宏块熵解码和整数余弦变换可以缩减 90%左右的指令周期，即：优化后的周期数只有优化前的十分之一左右，效果非常显著。去除块效应滤波器由于需要根据图像内容来判断滤波器系数[9]，很多优化技术无法使用，优化效果不太明显，只能缩减 30%~40%。

**表 3 优化前后主要模块的性能比较**

| 序列名称 | 功能模块 | 优化前周期数 | 优化后周期数 | 节省比例/% |
|---|---|---|---|---|
| Foreman | 1/4 像素插值 | 376 002 171 | 34 643 695 | 90.88 |
| | 单个宏块熵解码 | 182 287 177 | 17 552 506 | 90.37 |
| | 整数余弦反变换 | 146 137 172 | 14 955 579 | 89.77 |
| | 去除块效应滤波器 | 51 600 598 | 28 759 026 | 44.27 |
| Missa | 1/4 像素插值 | 708 345 014 | 73 732 757 | 89.59 |
| | 单个宏块熵解码 | 677 228 891 | 84 965 599 | 87.45 |
| | 整数余弦反变换 | 557 647 239 | 57 592 478 | 89.67 |
| | 去除块效应滤波器 | 144 242 543 | 78 865 985 | 45.23 |
| Mobile | 1/4 像素插值 | 1 640 379 240 | 170 201 159 | 89.62 |
| | 单个宏块熵解码 | 757 581 479 | 109 678 037 | 85.52 |
| | 整数余弦反变换 | 577 954 776 | 61 386 659 | 89.38 |
| | 去除块效应滤波器 | 231 629 395 | 164 311 343 | 29.06 |

表 4 给出了 H.264 解码算法优化前后总体性能的比较。由表 3、表 4 我们不难看出：综合运用上述方法不仅大幅度地提高了关键模块的运算速度，也大幅度提高了算法的总体性能。优化后的 H.264 算法的总指令周期至少可减少 84%。

**表 4 优化前后总体性能比较**

| 序列名称 | 指令周期总数 | | | 序列平均硬件解码速度/(帧/s) | |
|---|---|---|---|---|---|
| | 优化前 | 优化后 | 节省比例/% | 优化前 | 优化后 |
| Foreman | 1 505 809 796 | 215 533 771 | 85.69 | 13.9 | 112.2 |
| Missa | 4 501 442 618 | 538 064 354 | 88.05 | 5.1 | 39.8 |
| Mobile | 7 970 744 603 | 1 236 832 762 | 84.48 | 2.6 | 16.5 |

表 4 还给出了在实际的 DSP 硬件上的测试结果。我们使用的硬件平台是具有 PCI 接口的 DSP 开发板，板上有主频为 200MHz 的 PNX1502 DSP 芯片，外围有 8MB 的内存和相应的音频、视频输入输出接口。从表 4 可以看出，对于 QCIF 格式的 Foreman 序列，在优化之前每秒平均只能解码 13.9 帧图像，而优化后每秒可以解码 112.2 帧图像。对于 CIF 格式的 Missa 序列，优化前每秒只能解码 5.1 帧，而优化后每秒可解码 39.8 帧，对于 CIF 格式的 Mobile 序列，优化前每秒只能解码 2.6 帧，优化后能达到 16.5 帧，平均的解码速度提高了 8 倍，效果还是非常显著的。

## 6 结论

本文详细分析了影响 H.264 解码速度的主要因素，并给出了基于 Trimedia DSP 平台的优化方案。本方案综合采用了以下优化方法：缩减不必要的判决分支；数据本地化，减少内存操作次数；尽可能使用连续内存，避免内存的重复分配与释放；合理使用静态数据，减少不必要的内存操作；使用循环展开技术，减少循环开销，提高代码并行效率；尽可能使用高效指令等等。

算法仿真结果表明，经过优化的代码指令周期可比优化前减少 84%以上。在实际硬件芯片上的测试也证明这种方案能将 H.264 的解码速度提高 8 倍左右，基本满足中低速率实时视频应用的需要。(收稿日期：2005 年 6 月)

## 参考文献

1.ITU-T.Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification(ITU-T Rec.H.264 | ISO/IEC 14496-10 AVC)[S].2002

2.T Wiegand,G J Sullivan,G Bϕntegaard et al.Overview of the H.264/AVC video coding standard[J].IEEE Transactions on Circuits and Systems for Video Technology,2003;13(7):560~576

3.M Horowitz,A Joch,F Kossentini et al.H.264/AVC baseline profile decoder complexity analysis[J].IEEE Transactions on Circuits and Systems for Video Technology,2003;13(7):704~716

4.Y Dai,Q Li,Q Zhang et al.SIMD-aware loop unrolling for embed-

(下转 89 页)

```
……
CorEnd[num2][0]=x+(ls*cos((theta+shoulder)/180*Pi)+le*cos((theta+shoulder+elbow)/180*Pi)+lh*cos((theta+shoulder+elbow+hand)/180*Pi));
CorEnd[num2][1]=H+dist;
CorEnd[num2][2]=z-(ls*sin((theta+shoulder)/180*Pi)+le*sin((theta+shoulder+elbow)/180*Pi)+lh*sin((theta+shoulder+elbow+hand)/180*Pi));
……
```

## 4.5 动态仿真的实现

OpenGL 提供了双缓存，可以用来制作动画。也就是说，在显示前台缓存内容中的一帧画面时，后台缓存正在绘制下一帧画面，当绘制完毕，则后台缓存内容便在屏幕上显示出来，而前台正好相反，又在绘制下一帧画面内容。这样循环反复，屏幕上显示的总是已经画好的图形，于是看起来所有的画面都是连续的。双缓存技术可以生成平滑的动画，从一幅图像变化到下一幅图像的时间极短，人眼是不会感到这种变化的。

程序设计主要步骤如下：

(1)程序首先进行初始化操作，完成移动机械手初始位姿的设定、变量初值的设定、视点位置及光源等的设定。

(2)触发定时器，可以通过菜单、按钮、鼠标等来触发定时器。定时器设定的时间不宜过短，要为规划算法及图像的绘制留下充足的时间。

(3)在定时器响应函数 CPPMMView::OnTimer 中，要完成路径规划算法的实现，并调用 CPPMMView::Draw()来完成连续的动画。当然如果是针对静态环境的规划算法可以在 OnTimer 外来完成规划，并将规划结果存入一数组，再把路径的中间点(移动机械手广义坐标)按顺序在定时器响应函数内调用 Draw()来完成动画。如果是针对动态环境的规划则需要在 OnTimer 中实时地将规划结果发给 Draw()来完成动画。Draw()函数绘制所有的场景，其源程序如下：

```
void CPPMMView::Draw()
{
  glClearColor(1.0f,1.0f,1.0f,0.0f);
  glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
//清空颜色缓冲区
  DrawPlane();//绘制运动平面
  //绘制障碍
  DrawObstacle(3,0,4,1);
  DrawObstacle(-5,0,4,1);
  DrawObstacleCube(-7,-6,8.0,2.0,4.0);
  DrawRobot(x,z,theta,dist,shoulder,elbow,hand);//机器人
  DrawLineP(x,0,z);//绘制移动平台底面中心轨迹
  //绘制机械手末端轨迹
  DrawLineE(x,z,theta,dist,shoulder,elbow,hand);
  glFlush();
}
```

## 4.6 仿真结果

仿真结果如图 6 所示，其中长方体及圆柱体为障碍物，可以根据仿真的需要设定障碍物的数目及位置，这里仅画出 3 个进行说明。图中的两条轨迹分别为平台底面中心及机械手末端轨迹。可以通过调整视点从不同的角度观察仿真结果。

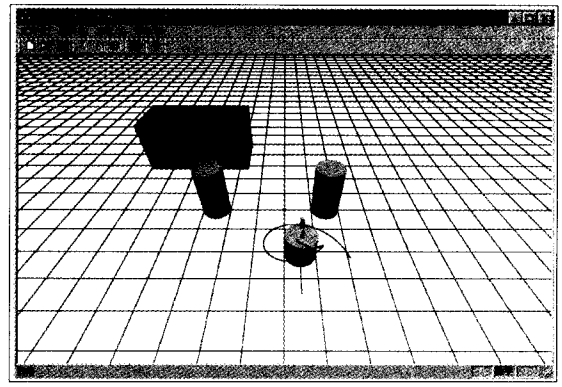

图 6 仿真结果

## 5 结论

本文利用 OpenGL 设计并实现了移动机械手的路径规划仿真平台，以三维动画的方式对移动机械手的运动进行了仿真，仿真结果生动形象，为路径规划算法的验证奠定了基础，对路径规划的研究起到了很好的辅助作用。(收稿日期：2005 年 6 月)

## 参考文献


1.K Nagatani,T Hirayama et al.Motion Planning for Mobile Manipulator with Keeping Manipulability[C].In:Proceeding of the 2002 IEEE/RSJ International Conference on Intelligent Robots and Systems,2002:1663~1668

2.R M Murray,S Shankar Sastry.Nonholonomic Motion Planning:Steering Using Sinusoids[J].IEEE TANSACTIONS ON AUTOMATIC CONTROL,1993;38(5):700~716

3.吴斌.OpenGL 编程权威指南[M].第三版，中国电力出版社，2001

4.费广正.Visual C++高级编程技术 OpenGL 篇[M].中国铁道出版社，2000


(上接 45 页)


ded code optimization[C].In:Proceedings of the SPIE,2003;5241:157~168

5.Lappalainen V,Hallapuro A,Timo D H.Complexity of optimized H.26L video decoder implementation[J].IEEE Transactions on Circuits and Systems for Video Technology,2003;13(7):717~925

6.Philips TriMedia Documentation Set[EB/OL].http://www.trimedia.philips.com,1999

7.JVT Software Version 8.1[EB/OL].http://iphome.hhi.de/suehring/tml/download/old_jm/jm81a.zip,2004

8.朱冬冬，丁嵘，尹亚光等.H.264 软件解码器的优化[J].电视技术，2003;(12):4~6

9.P List,A Joch,J Lainema et al.Adaptive deblocking filter[J].IEEE Transactions on Circuits and Systems for Video Technology,2003;13(7):614~619