基于寄存器优化的图象中低层处理

杨海波 姜骊黎 姚庆栋
(浙江大学信息与电子工程学系,杭州 310027)

摘 要 提出了一种在高性能 RISC 芯片上进行图象中低层处理的寄存器优化方法。使用该方法能使处理速度提高将近一倍,在 TMS320c40 上所做的实验表明应用该方法能取得较好的效果。

关键词 图象处理 循环展开 寄存器优化

0 引 言

众所周知图象处理和模式识别任务需要大量的运算,如何实现这些运算量大的任务一直是图象处理工作者关心的课题之一。解决这个问题的方法有 2 种:其一是为了取得实时性能可采用并行的思想加速运算[1];其二是当实验室计算机资源有限时,应仔细研究图象处理任务的特点,对程序结构进行尽量巧妙的组织,以充分利用硬件资源和减少冗余操作[2]。用并行的方法加速运算对硬件要求较高,如采用粗粒度并行需要多处理机结构[3,4],挖掘指令级并行要求芯片是超标量或超长指令形式[5,6];第 2 种方法适用于多数 RISC 结构的计算机程序优化以提高计算速度。

事实上在 RISC 结构的计算机上,传统的编译器对于寄存器的利用效率往往不高。因为传统的编译器对寄存器的分配以表达式为单位[7],对于不同的表达式寄存器被重新分配。通常情况下这种分配方法有利于克服寄存器的溢出,然而采用上述寄存器分配方案在某些情况下也存在着明显不足。如图 1 所示,当循环结构各次迭代的语句之间存在不变的内存数据时,上述寄存器分配方法往往会造成同一个数据反复从内存装入寄存器。图 1 中的变量 a 和 b 如果按传统寄存器的分配方法将装入同一寄存器。对于循环的各次迭代,变量 a 保持不变因而不必重新装入寄存器,但由于变量 b 与其共用同一寄存器覆盖了它的值,致使在循环的每次迭代中都必须重新装入,对于变量 b 存在类似的重复装入寄存器现象。显然当我们为 a 和 b 分配不同的寄存器时,上述重复装入寄存器的现象将被消除。循环结构各次迭代的语句之间存在不变内存数据的现象在图象中低层处理中经常遇到。我们知道相当多的图象中低层操作可看成是图象与窗口算子的卷积(或相关),假定图象尺寸为 $M \times M$,窗口算子尺寸为 $N \times N$,上述图象与窗口算子的卷积(或相关)可用图 2 的 C 语言程序段表示。

```
for(i = 0;i < m;i ++)
  {c1[i] = a * d1[i];
   c2[i] = b * d2[i];
  }
```

图 1 循环各次迭代的语句之间存在不变的内存数据

```
for(x = 0;x < M;x ++)
for(y = 0;y < M;y ++)
  {temp = 0;
    for(i = 0;i < N;i ++)
      for(j = 0;j < N;j ++)
        temp + = image[x + i][y + j] * mask[i][j];
   dest[x + N/2][y + N/2] = temp/norm;
  }
```

图 2 图象(尺寸为 $M \times M$)与窗口算子(尺寸为 $N \times N$)的卷积(或相关)的 C 语言程序段

图 2 中大括弧内的循环语句及循环体表示窗口算子与图象某一局部的卷积,我们称之为内部循环。大括弧外的循环语句及循环体表示窗口算子与整个图象的各个部分卷积,我们称之为外部循环。当我们展开内部循环时,窗口算子的各个分量对于外部循环的各次迭代是不变的内存数据。与图 1 所示程序的处理方

收稿日期:1998-07-14;收到修改稿日期:1998-11-23

法类似,当我们对窗口算子的各个分量分配不同的寄存器时,相当多的寄存器重复装入操作将被消除,从而使程序执行效率得到较大提高。值得指出的是这里应用了 RISC 机通常有较多寄存器的概念。当然这种方法也会产生一些问题,我们在第 1 节针对小模板、中模板、大模板的不同情形讨论了本文提出的寄存器优化方法,在第 2 节我们给出了实例与讨论。

1　具体方法

1.1　小模板算子

小模板算子以 Sobel 算子为例,假如图象尺寸为 256×256,未优化的程序段如图 3 所示。

```
for(x = 0;x < 256;x ++)
  for(y = 0;y < 256;y ++) {
    temp = 0;
      for(i = 0;i < 3;i ++)
        for(j = 0;j < 3;j ++)
          temp + = image[x + i][y + j] * mask[i][j];
      dest[x + 3/2][y + 3/2] = temp/norm;
  }
```

图 3　Sobel 算子的原始程序段

本方法的第一步是内部循环展开。内部循环展开不但对于本优化方法是必不可少的一个环节,而且本身意义重大。首先循环开销例如循环控制变量的修改、循环条件的判断等被删除;其次,内部循环展开使得原先的指令数较少的循环体指令序列变成指令数众多的非循环体指令序列,各次迭代间的指令级并行得到充分的暴露,这样许多指令调度优化和经典优化可被使用。在上述例子中,二维数组引用可被简化成一维数组引用,甚至简单的变量引用。temp 为累加器,原先的算法中每做一次乘积与累加都必须把累加值保存在 temp 变量中,因为在通常的编译器中,此处的乘积与累加共用一个寄存器,原因是加法运算的一个加数为乘积结果,它已放在寄存器中,所以加法运算不必再另外申请寄存器。在原来的内部循环的循环体中,必须把该值返回给 temp 内存变量,否则下次进行乘积运算时,上一次累加值将被覆盖,其原因是前后两次迭代中寄存器的分配情况完全一致。在新的方案中,我们顺便把累加值另分配给一个寄存器,以免乘加操作的部分和频繁存入内存。因为这里的乘积运算是相互独立的,所以在超标量或超长指令机中,这里的内部循环展开还能导致并行加速。内部循环展开后如图 4 所示。

```
for(x = 0;x < 256;x ++)
  for(y = 0;y < 256;y ++) {
    temp = 0;
    temp + = image[x][y] * mask[0][0];
    temp + = image[x][y + 1] * mask[0][1];
    temp + = image[x][y + 2] * mask[0][2];
    temp + = image[x + 1][y] * mask[1][0];
    temp + = image[x + 1][y + 1] * mask[1][1];
    temp + = image[x + 1][y + 2] * mask[1][2];
    temp + = image[x + 2][y] * mask[2][0];
    temp + = image[x + 2][y + 1] * mask[2][1];
    temp + = image[x + 2][y + 2] * mask[2][2];
  dest[x + 1][y + 1] = temp/norm; }
```

图 4　Sobel 算子原始程序段的内部循环展开

注意此处应为 temp 变量分配一个寄存器。新方法的第二步是小模板的每一个元素(或分量)都各自分配一个寄存器,并且这种分配必须在外部循环语句前进行。为清晰起见,用图 5 的高级语言程序段表示。

```
r0 = mask[0][0];
r1 = mask[0][1];
r2 = mask[0][2];
r3 = mask[1][0];
r4 = mask[1][1];
r5 = mask[1][2];
r6 = mask[2][0];
r7 = mask[2][1];
r8 = mask[2][2];
for(x = 0;x < 256;x ++)
  for(y = 0;y < 256;y ++)
    {temp = 0;
      temp + = image[x][y] * r0;
      temp + = image[x][y + 1] * r1;
      temp + = image[x][y + 2] * r2;
      temp + = image[x + 1][y] * r3;
      temp + = image[x + 1][y + 1] * r4;
      temp + = image[x + 1][y + 2] * r5;
      temp + = image[x + 2][y] * r6;
      temp + = image[x + 2][y + 1] * r7;
      temp + = image[x + 2][y + 2] * r8;
    dest[x + 1][y + 1] = temp/norm;
    }
```

图 5　Sobel 算子原始程序段的寄存器优化

由于循环次数非常大,所以循环外部的寄存器装载开销可以忽略不计。原先的运算主要为模板元素装载、求积与求和,现在省去装载,并且求和所占的计算量相对较小,故仅由削减寄存器装载就可减少将近一

半的计算量。如果结合其它优化手段如循环开销消除、数组元素引用优化等,总收益为提高 4 倍左右的计算速度。对于设计新的编译器,上述思想可作为优化器(optimizer)的一部分加以实现,对于不支持上述优化的编译器,上述思想只能在编译器支持的高级语言程序中借助于插入汇编来实现。

1.2 中模板算子

模板算子一次装入寄存器并滞留其中的一个困难是对寄存器的个数有一定的要求,在模板较大时可能存在寄存器组容量不够问题。假如模板还未足够大,以至于通用寄存器组的容量足够而空闲的通用寄存器容量不够,则可在执行外部循环语句前,把被其它指令占用的通用寄存器压入寄存器堆栈加以保存,在执行完外部循环语句后加以恢复。由于入栈和出栈操作同循环的次数相比微乎其微,所以由此引起的性能下降可以略去不计,即收益仍可看作与小模板算子的优化一样。

1.3 大模板算子

以 10×10 大模板 Marr 算子为例,假如由于程序其它部分对寄存器的占用导致 Marr 算子的各个分量最多只能使用 25 个寄存器,则我们就把该大模板均分为 4 个子模板分 4 次计算,并把中间结果暂时存入输出数组中,10×10 Marr 算子的原始程序段如图 6 所示。

```
for(x = 0;x < 256;x ++)
 for(y = 0;y < 256;y ++)
    {temp = 0;
      for(i = 0;i < 256;i ++)
        for(j = 0;j < 256;j ++)
          temp + = image[x + i][y + j] * mask[i][j];
      dest[x + 5][y + 5] = temp/norm;
    }
```

图 6 10×10 Marr 算子的原始程序段

采用本优化方法改进后的程序段如图 7 所示。

```
push(r0);
push(r1);
  …
 push(r24); /* r25 留作 temp 类加变量,其它诸如数组
               元素引用的寄存器也已经被留出 */
r0 = mask[0][0];
r1 = mask[0][1];
  …
r24 = mask[4][4];
for(x = 0;x < 256;x ++)
  for(y = 0;y < 256;y ++)
    {temp = 0;
       temp + = image[x][y] * r0;
       temp + = image[x][y + 1] * r1;
         …
       temp + = image[x + 4][y + 4] * r24;
      dest[x + 5][y + 5] + = temp/norm;
    }
  r0 = mask[0][5];
  r1 = mask[0][6];
    …
  r24 = mask[4][9];
for(x = 0;x < 256;x ++)
  for(y = 0;y < 256;y ++)
    {temp = 0;
      temp + = image[x][y + 5] * r0;
      temp + = image[x][y + 6] * r1;
        …
      temp + = image[x + 4][y + 9] * r24;
      dest[x + 5][y + 5] + = temp/norm;
    }
      …
      …
      r0 = mask[5][5];
      r1 = mask[5][6];
           …
      r24 = mask[9][9];
    for(x = 0;x < 256;x ++)
      for(y = 0;y < 256;y ++)
        {temp = 0;
          temp + = image[5][5] * r0;
          temp + = image[5][6] * r1;
            …
          temp + = image[9][9] * r24;
        dest[x + 5][y + 5] + = temp/norm;
        }
```

图 7 10×10 Marr 算子程序段的寄存器优化

上述算法中分成 4 次装载寄存器引入的开销微乎其微,所增加的主要开销由中间结果暂存于输出数组引起。由于循环体语句众多,所以增加这样一次操作尽管其绝对计算量较大,但相对计算量仍可近似不计。

2 实例与讨论

上述思想不仅可用于卷积(或相关)运算,而且还可用于其它许多图象处理任务,如运动估计、霍夫变换等。为验证本方法的性能,我们在 TMS320C40 上运

行了几个基于寄存器优化的图象处理中低层程序,实验表明采用新方法后性能提高将近一倍(见表1)。

表1

处理时间(ms)	Sobel 算子	5*5 卷积	10*10 Marr 算子	霍夫变换
基于普通优化	54.4	132.2	611.2	2610.6
增加本优化技术	29.3	70.7	331.6	1521.1

应该指出使用本方法也有一定的局限性,其主要缺点可概括如下:

(1)设计编译器时增加本优化方法将使编译器的优化程序更加复杂,当编译器不支持上述优化时,本方法需采用在高级语言程序中插入汇编行实现。

(2)本文所论述的寄存器优化只是其中一种优化方法,所以单独使用一般只能加速一倍左右,想取得更满意的加速效果一般应与其它优化方法同时使用。

(3)当内部循环的循环次数明显小于外部循环的循环次数时,增加本优化方法才能取得满意效果,否则加速效果不明显。

参考文献

1 Wang Cho-Li *et al*. High-performance computing for vision. In: Proceedings of the IEEE, 1996, 84(7).

2 Baglietto P *et al*. Image processing on high-performance RISC systems. In: Proceedings of the IEEE, 1996, 84(7).

3 Wang K H. Advanced Computer Architecture. McGraw-Hill Book Company, 1993.

4 严学强. 基于高性能 DSP-TMS320C40 的多处理机系统的研究与设计:[学位论文]. 杭州: 浙江大学, 1997.

5 Smith J E *et al*. The microarchitecture of superscalar processors. In: Proceedings of the IEEE, 1995, 83(12).

6 李彭城. 90年代计算机技术热点——RISC 计算机. 北京:北京理工大学出版社.

7 亚　文,张　勇,建　平. 编译原理与实践. 北京:中国科学院希望高级电脑技术公司.

杨海波　浙江大学信息与电子工程学系博士生,主要研究领域为并行编译器,并行操作系统,计算机视觉,知识工程等。

姜骊黎　浙江大学信息与电子工程学系博士生,主要研究领域为图象理解,专家系统等。

Middle and Low Level Image Processing Based on Register Optimising

Yang Haibo, Jiang Lili and Yao Qingdong

(*Department of Information and Electronic Engineering, Zhejiang University, Hangzhou* 310027)

Abstract　An approach about register optimising has been presented in this paper, which can be used to develop middle and low level image processing program more effectively. Double speeds can be attained with this method. Experiment based on TMS320C40 shows that a good effect has been reached with this method.

Keywords　Image processing, Loop unrolling, Register optimising