面向多簇超长指令字 DSP 的向量化优化算法①

徐华叶 [1,2], 郑启龙 [1,2], 丁陈飞 [1,2], 徐东鹏 [2]

[1](中国科学技术大学 计算机学院, 合肥 230027)
[2](安徽省高性能计算重点实验室, 合肥 230027)

摘 要: BWDSP 是一款针对高性能计算领域设计的处理器, 采用多簇超长指令字(VLIW)体系结构和 SIMD 架构, 同时也提供了很多向量化指令. 然而现有的编译框架无法对这些向量化指令提供支持, 因此本文提出了一种向量化优化算法, 可以显著提高一些在 DSP 领域有着广泛应用的计算密集型程序的性能. 最终实验结果表明,该优化算法能够平均取得 6.60 倍的加速比.

关键词: 向量化优化; 编译技术; 访存地址分析; 多簇体系 DSP; 超长指令字

Vectorization Algorithm for Multi-Cluster and VLIW DSP

XU Hua-Ye[1,2], ZHENG Qi-Long[1,2], DING Chen-Fei[1,2], XU Dong-Peng[1]

[1](School of Computer Science and Technology, USTC, Hefei 230027, China)
[2](Anhui High Performance Computing key laboratory at Hefei, USTC, Hefei 230027, China)

Abstract: This paper presents a vectorization algorithm for multi-cluster and VLIW(very long instruction word) DSP and this algorithm can significantly improve the performance of some compute-intensive programs which are widely used in DSP field. Via pre-handling the chain of the instruction and synthesizing the special instruction if needed, at last the algorithm synthesizing the vectorized instruction which the DSP provided. The experimental result shows this vectorization algorithm achieves 6.60 times performance improvement on average.

Key words: vectorization optimization; compiling technique; analysis of memory; multi-cluster DSP; VLIW

1 研究背景

BWDSP 是一款针对高性能计算领域设计的处理器, 可广泛应用于雷达、电子对抗、精确制导、通信保障、图像处理等信号处理领域. 它采用超长指令字(VLIW)[1]的分簇结构, 有四个执行簇和三个地址产生器[2], 而 BWCC 是一款基于可重定向基础编译设施 openimpact, 自主开发的面向 BWDSP 的 C 语言编译器, 后端分为指令注释、指令分簇、软件流水、寄存器分配、指令调度以及汇编代码生成六个模块[3].

BWDSP 采用了 SIMD 架构, 但并没有提供传统意义上的长向量化部件, 而是提供了很多向量化指令, 如:{x,y,z,t}R0 = R1 OP R2,这个指令表示在四个簇上同时执行一个 OP 指令. 而{x,y,z,t}Rs+1:s=[Un+=Um, Uk]则表示的是一个双字传输指令, 能够一次性读出[Un]和[Un]+1、[Un+2Uk]和[Un+2Uk+1]、[Un+2*2Uk]和[Un+2*2Uk+1] 、[Un+3*2Uk]和[Un+3*2Uk+1]共 8 个地址的数据, 并依次送到四个簇上的同名寄存器对上({x,y,z,t}Rs:s+1). 除此之外, BWDSP 还提供了其他诸如复数指令, MAX, MIN 等一些列向量化指令. 比如, 对于简单的向量加法运算(如图 1 所示).

```
for( i = 0 ; i < N ; ++ i )
  a[i] = b[i] + c[i] ;
```

图 1 向量加(vector addition)

一般而言编译器会将这个循环的循环体部分翻译成四个周期的汇编程序(在目标 DSP 中, load store , add 指令均为一个周期), 如图 2 所示(汇编以 BWDSP 的汇

① 基金项目:“核高基”重大专项(2012ZX01034-00-001)
收稿时间:2013-05-20;收到修改稿时间:2013-06-13

编形式表示,用运算符号表示 load, store, 运算等). 然而对于 BWDSP 而言, 对于上述循环最理想的情况是翻译成如图 3 所示的汇编程序, 利用向量化指令优化后, 相当于将 8 个循环体合成一个循环体. 另外由于 BWDSP 采用 VLIW 体系结构, 能够在一个指令周期中发射 16 条指令, 另外在一个簇上有 8 个 ALU, 所以两条加法指令可以并行执行的, 所以最终优化的结果是如图 4 所示.

```
R0 = b[i]      (load)
R1 = c[i]      (load)
R2 = R0 + R1   (add)
a[i] = R2      (store)
```

图 2 一般情况下向量加的汇编

```
{x,y,z,t}Rm:m+1 = [Un += 8 , 1]
{x,y,z,t}Rs:s+1 = [Uk += 8 , 1]
{x,y,z,t}Rp = Rm + Rs
{x,y,z,t}Rp+1 = Rm+1 + Rs+1
[Um += 8 , 1] = {x,y,z,t}Rp:p+1
注：其中 Un、Uk、Um 分别表示当前 a[i]、b[i]、c[i]的基地址
```

图 3 经向量化指令优化的汇编

```
{x,y,z,t}Rm:m+1 = [Un += 8 , 1]
{x,y,z,t}Rs:s+1 = [Uk += 8 , 1]
{x,y,z,t}Rp = Rm + Rs ‖ {x,y,z,t}Rp+1 = Rm+1 + Rs+1
[Um += 8 , 1] = {x,y,z,t}Rp:p+1
注：上述指令中“‖”表示一条中两个并行的指令槽，在同一个指令周期中执行。同样 Un、Uk、Um 分别表示当前 a[i]、b[i]、c[i]的基地址
```

图 4 进一步利用 VLIW 优化后的汇编

然而, IMPACT 的编译框架和传统的向量化编译优化无法对这些指令提供编译支持. 故本文利用编译制导, 提出一种向量化编译算法. 尽管该算法针对 BWDSP 目标平台, 但方法通用, 对于其他处理器平台同样适用.

2 面向多簇VLIW DSP的向量化优化算法

2.1 概述

目前实现自动向量化的方式有主要有两种: (1)基于循环的自动向量化: 通过分析循环, 对于无数据依赖的并行迭代, 直接生成对应的向量指令的方式实现向量化[4]; (2)基于基本块的自动向量化: 通过循环展开, 得到较大的基本块, 然后收集循环中的相同操作, 合成后端 CPU 支持的 SIMD 操作, 实现向量化[5].

上述两种方案主要面对通用处理器, 在工业级的编译器 gcc 和 open64 中都有完备的实现, 但是无法处理 BWDSP 这类特殊的体系结构和特殊的向量化指令. 本文针对针 BWDSP, 提出了一种基于编译制导的面向多簇 VLIW DSP 的向量化优化算法. 该算法的主要思想在于对循环体中的load和store指令以及运算指令进行代码移动, 并对循环步长进行处理, 最终合成 BWDSP 提供的向量化指令.

算法的处理过程为: 1.向量化合成之前的预处理 2.基于特殊指令的向量化处理 3.内存循环的向量化变换. 算法的伪代码如图 5 所示.

```
Vectorization(L_Func * fn){
    for(bb=fn->first_cb ; bb ; bb= bb->next_cb){
        if( bb->attr->name == “SIMD” ){
            Data_Flow_Analysis( bb ) ;
            // 数据流分析
            Change_Global_Array_Access( bb ) ;
    //        改变全局数组的访问方式
            Judge_Single_Double_Word( bb) ;
    //        合成单双字判断
            Vectorization_Transformation( bb ) ;
            // 内存循环的向量化变换
        }
    }
}
```

图 5 向量化优化算法的伪代码

2.2 向量化合成之前的预处理

在向量化指令合成之前需要进行的预处理包括以下部分:

(1) 数据流分析: 分析循环以及指令之间的依赖关系, 为内存循环向量化变量提供支持[6].

(2) 数组访问方式的统一: BWCC 对于全局数组和局部数组采取了不同的访问方式, 所以在向量化合成中需要对访问方式进行统一, 全部采取基址加偏移地址的方式.

(3) 合成单双字判断: BWDSP 提供了双字传输指

令和单字传输指令, 而在最终的向量化合成中到底使用双字还是单字需要根据循环的步长以及循环体中对连续地址的访问跨度来决定.

循环变量的改变我们可以分为两种, i = i + 1 和 i = i + C, 其中 C 为非 1 的常量(对于 i = i + V , V 为循环变量时, 循环无法进行向量化合成, 而 V 为循环不变量时, 情况和 i = i + C 一样).

当循环变量的改变为 i = i + 1 时, 如果循环体中对于数组元素的访问为 a[i]或者 a[i+C]时, 此时可以进行双字合成, 向量化地址的形式类似于[Ua+Ui,1], 合成之后的循环步长为 i = i + 8; 如果循环体中对于数组元素的访问形式为 a[C*i + C]或者 a[C*i], 则此时只能合成单字, 最终步长的改变为 i = i + 4. 如果循环体中对数组元素的访问存在连续地址的访问, 比如同时存在对 a[i]和 a[i+1]的访问, 如果同时进行单字合成的话则地址分布存在冲突, 所以对于这种情况只能放弃合成. 而对于 a[C*i]和 a[C*i+1](此时 C 必须为偶数)则可以进行单字合成, 循环步长变为 i = i + 4.

对于 i = i + C 的形式, 如果循环体中对于数组元素的访问形式为单一访问(即只存在 a[i]或者其他形式), 则均只能合成单字; 而当循环体中对数组元素的访问形式为连续地址的访问时(即同时存在 a[i]和 a[i+1]形式或者类似形式), 可以对每个访问形式同时进行单字合成, 最终的步长变为 i = i + 4*C 的形式.

2.3 内存循环向量化变换

目前的循环向量化只考虑内存循环, 对最内存循环体进行以下变换:

(1) 首先对内存循环体中的所有 load 和 store 操作进行前移和后移,便于后面操作的识别和精简[7], 并且对这些操作标记 SIMD 属性. 对于循环控制块中的所有 load 操作, 做顺控制流的正向处理, 尽可能的前移, 但是不越过同类操作(load)、过程调用(JSR), 并且和越过的指令之间不存在依赖关系[4](在数据流分析中进行判断). 同理, 对于循环控制块中的 store 操作做同样处理. 由于后移, 控制条件更为复杂, 除不能越过同类操作(store)、过程调用(JSR)以及存在依赖关系的指令, 还不能跨越控制流转移操作, 如跳转指令(JUMP)、返回(RTS)等操作. 通过 load 和 store 指令的前移和后移, 将指令链汇聚成加载操作片和存储操作片, 为后面的向量化指令合成提供方便.

以图 6 为例来说明对于 load、store 指令的前移和后移. 图 6(a)中为循环体代码, (b)中为移动之前的指令链, (c)中为 load、store 前移和后移之后的指令链.

```
for( i = 0 ; i < N ; ++ i ){
    a[2i] += b[i];
  a[2i + 1] += c[i] ;
}
```

(a) 循环体代码

```
R0 = b[i]       (load)
R1 = a[2i]      (load)
R2 = R1 + R0    (add)
a[2i] = R2      (store)
R3 = c[i]       (load)
R4 = a[2i + 1]  (load)
R5 = R4 + R3    (add)
a[2i +1] = R5   (store)
```

(b) 移动前指令链

```
R0 = b[i]   (load)
R1 = a[2i]  (load)
R3 = c[i]   (load)
R4 = a[2i +1] (load)
R2 = R1 + R0 (add)
R5 = R4 + R3 (add)
a[2i] = R2   (store)
a[2i +1] = R5 (store)
```

(c) load、store 前移及后移后指令链

图 6　load、store 的前移与后移

(2) 对 load 操作进行扩展. 根据第一步的步长判来决定是否进行双字扩展. 如果进行双字扩展,首先对 load 的各个寄存器进行双字寄存器对扩展, 最后进行四簇扩展.

(3) 对运算操作进行扩展. 根据上一步的 load 扩展来决定运算指令的扩展. 如果 load 进行了双字扩展, 则必须首先拷贝并前插一个新的运算指令, 然后对这两条运算指令均进行四簇同时扩展. 如果 load 没有双字扩展的话则只需对这一条运算指令进行四簇同时扩展.

(4) 对 store 操作进行扩展. store 的扩展与第二步的 load 保持一致即可.

(5) 对循环步长进行扩展如果 load 和 store 进行了双字扩展和四簇同时扩展,则循环步长变为 8,如果仅仅进行了四簇同时扩展则只循环步长只需扩展为 4.

3 支持向量化优化的寄存器分配算法

算法的设计与实现对于寄存器分配模块的影响最大. 算法中设计到同簇寄存器对以及四簇同名寄存器的分配. 一个簇上连续的两个寄存器, 比如xRm,xRm+1,可以简写为 xRm:m+1 称为同簇寄存器对, 而类似于{x,y,z,t}R0 的形式则被称为四簇同名寄存器. BWDSP 的向量化指令以及双字指令的合成主要在于同簇寄存器对以及四簇同名寄存器的分配.

BWCC 的寄存器分配[3]采用启发式的图着色算法[8,9], 无法满足这种分簇结构 DSP 的向量化指令对于寄存器分配的要求. 因此在先有的寄存器分配算法的基础上进行了修改, 以便满足特殊寄存器形式的分配:

(1) 在向量化算法中, 对最终合成的操作指令的伪寄存器的属性进行标记(如需要分配同簇寄存器对或者四簇同名寄存器);

(2) 分配阶段, 首先考虑向量化算法、复数优化算法等对于寄存器的特殊形式的要求, 并先行分配;

(3) 在(2)的基础上, 将数据相关性强的指令尽可能分配同一个簇上的寄存器, 避免产生过多的簇间传输指令;

(4) 遍历指令链, 插入必要的簇间传输指令.

4 实验结果

实验选取了四个具有代表性的循环结构的程序(图 7 的卷积运算(convolution), 图 1 的向量运算, 图 6(a)的优化后只能合成单字的循环结构以及点数为 1024 的 fft_radix4)来验证算法的有效性. 这些例子在 DSP 领域中都有着非常广泛的应用.

对于这四种程序在BWCC下进行两种不同模式(无向量化优化和向量化优化)的测试, 结果如表 1 所示.

表 1 优化前和优化后的时钟周期数(cycles)

程序(N=1000)	优化前 (cycles)	优化后 (cycles)	加速比
卷积运算	6002	759	7.91
向量运算	6000	750	8
合成单字的循环	10000	2000	5
fft_radix4(1024)	148738	27154	5.48
平均			6.60

从上表可以看出, 对于这四种结构的程序, 使用本文提出的向量化优化算法, 平均能够得到 6.60 倍的加速比.

```
for( i = 0 ; i < N ; ++ i )
    sum += a[i] *b[i]
```

图 7 卷积运算

5 结论与展望

本文面向多簇 VLIW DSP, 利用其特殊的向量化指令和超长指令字特性, 提出了一种向量化优化算法. 该算法首先对需要进行向量化变换的循环进行预处理, 然后对内存循环进行向量化变换, 对于计算密集型的程序(如卷积运算、向量运算等)有很大的提升, 得到了 6.60 倍的加速比. 然而, 目前的算法尚只能合成部分向量化指令, BWDSP 提供的很多向量化指令都还无法合成, 所以在今后的工作中还需要继续研究向量化优化算法, 尽最大可能对 BWDSP 提供编译上的支持.

参考文献

1 Fisher JA, Faraboschi P, Young C. 嵌入式计算:体系结构,编译器和工具的 VLIW 方法.北京:机械工业出版社.2006:337–395.

2 CETC 38, BWDSP 100 Software User Manual.

3 邱鹏飞.BWDSP100 编译器的研制及优化技术研究[硕士学位论文].合肥:中国科学技术大学,2011.

4 Allen R, Kennedy K.张兆庆,乔如良等译.现代体系结构的优化编译器.北京:机械工业出版社.2004:23–44.

5 Larsen S, Amarasinghe S. Exploiting superword level parallelism with multimedia instruction sets. Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation. Vancouver, ACM Press. 2000.145–156.

6 Aho AV, Lam MS, Sethi R, Ullman JD.赵建华,郑滔,戴新宇译.编译原理.北京:机械工业出版社.2009:382–404.

7 Xu DP, Zheng QL. An address-based compiling Optimization for FFT on multi-cluster DSP. Proc. Of The International Symposium on Parallel Architectures. 2012. 60–64.

8 Chaitin GJ. Register allocation and spillin via graph coloring. ACM SIGPLAN Notices. 1982. 201–207.

9 Chow F, Hennessy JL. The priority-based coloring approach to register allocation. ACM Trans. Programming Languages and Systems. 1990. 501–536.