面向多簇架构 DSP 的树匹配向量化算法①

郭连伟, 郑启龙, 黄胜兵, 徐华叶

(安徽省高性能计算重点实验室, 合肥 230026)
(中国科学技术大学 计算机学院, 合肥 230027)

摘 要: BWDSP 是针对高性能计算设计的一款新型的处理器, 采用多簇超长指令字体系结构和 SIMD 架构, 有丰富的指令集. 为充分利用 BWDSP 提供的向量化资源, 迫切需要提出一种向量化算法. 本文在 open64 基础上研究并实现了面向多簇超长指令字(VLIW)DSP 的 SIMD 编译优化算法. 算法基于 OPEN64 的中间语言 WHIRL, 能够充分地利用 BWDSP 丰富的硬件资源和向量化指令. 最终实验结果表明, 对于能够合成双字和单字的循环程序, 该优化算法能够平均取得 6 倍和 4 倍的加速比.

关键词: 单指令多数据; WHIRL 树; 多簇; 超长指令字; 指令并行

SIMD Algorithm Based on Tree Matching for Multi-cluster and VLIW DSP

GUO Lian-Wei, ZHENG Qi-Long, HUANG Sheng-Bing, XU Hua-Ye

(Anhui High Performance Computing Key Laboratory at Hefei, USTC, Hefei 230026, China)
(School of Computer Science and Technology, USTC, Hefei 230027, China)

Abstract: BWDSP is a new type of processor designed for high performance computing, using multi-cluster VLIW structure and SIMD architecture, including a rich instruction set. In order to make full use of the resources of BWDSP, a SIMD algorithm is to be proposed. In this paper, an algorithm for DSP SIMD compiler optimization based on open64 infrastructure is studied and implemented. This algorithm is based on WHIRL intermediate language of Open64 and can make full use of rich hardware resources and vector instruction set. The experimental result shows that the vectorization algorithm achieves 6 times performance improvement for double-word vectorization and 4 times performance for single-word vectorization on average.

Key words: SIMD; WHIRL tree; multi-cluster; VLIW; instruction-level parallelism

1 研究背景

高性能的 DSP 在雷达、精确制导、电子对抗、通信保障、图像处理等信号处理等多种领域有着重要的应用. BWDSP 正是一款针对高性能计算领域设计的可用于这些领域的处理器. 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]. 基于可重定向基础编译框架 open64[3], 自主开发了面向 BWDSP 的 C 语言编译器, 我们称为 OCC.

BWDSP 采用了 SIMD 架构[4], 但它没有提供传统意义上的长向量化部件. 为了支持并行, 它提供了很多 SIMD 指令, 包括双(单)字多簇同时存取指令、双(单)字多簇同时运算指令、簇间传输指令等, 这些指令在文献[2]中有详细介绍.

下面我们以简单的向量加法运算(如图 1 所示)为例来简单介绍 BWDSP 向量化指令的运用.

对通用编译器来说, 每次循环, 这个循环体会被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. 显然, 通用编译器没有利用 BWDSP 的体系结构的特殊性, 程序效率低下. 相反, 利用 BWDSP 的向量化指令, 可以将循环体程序翻译成如图 3 所示的汇编程序, 而循环步长则变为 N/8. 此外, BWDSP 采用 VLIW 体系

① 基金项目: "核高基"重大专项(2012ZX01034-00-001)
收稿时间:2015-02-06;收到修改稿时间:2015-04-26

结构, 在一个指令周期中能够发射 16 条指令, 并且一个簇上有 8 个 ALU, 所以图 3 中的两条加法指令可以并行执行, 最终优化的结果更加高效, 如图 4 中所示.

```
for( i = 0 ; i < N ; ++i ){
    A[i] = B[i] + C[i];
}
```

图 1 向量加程序

```
R0 = B[i]      (LOAD)
R1 = C[i]      (LOAD)
R2 = R0 + R1   (ADD)
A[i] = R2      (STORE)
```

图 2 向量加循环体的汇编翻译

```
{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]的基址, {x, y, z, t}表示簇选择
```

图 3 BWDSP 向量加的理想优化结果

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

图 4 BWDSP 向量加优化最终理想的结果

SIMD 编译优化要解决的主要问题是如何有效地把各种用高级语言编写的多种形式的代码转换成多媒体扩展集中对应的 SIMD 指令, 使生成的 SIMD 代码尽可能高效[5]. 目前, 针对 SIMD 的研究主要包括: SIMD 编译优化的特点和效果的研究[6], 对可进行 SIMD 向量化操作的识别[7,8,9], 访存优化[10,11]等.

Open64 和传统的向量化编译优化无法对 BWDSP 提供的向量化资源提供编译支持. 本文在 open64 基础设施上, 提出并实现了基于 WHIRL 中间语言的 SIMD 向量化算法, 能够很好地利用 BWDSP 的向量化资源, 提高循环程序的效率.

2 基于WHIRL树的SIMD算法

2.1 算法概述

Open64 编译器的中间表示(Intermediate Representation, IR)采用的是独立中间表示 whirl 语言. Whirl 中间语言采用层次结构, Open64 的优化策略是尽可能在高层 Whirl 进行优化, 各个层次的 whirl 共享数据流、控制流的分析结果, 层次之间避免重复优化工作[12].

基于 whirl 树的向量化算法的基本思想是对嵌套循环中的最内层循环进行分析, 对于满足条件的循环, 经过特殊向量化处理(如果需要)、预处理、SIMD 指令合成、SIMD 后续归约、SIMD 后续处理等阶段, 最终合成 BWDSP 的向量化指令.

图 5 是基于 Whirl 树的向量化算法的处理流程.

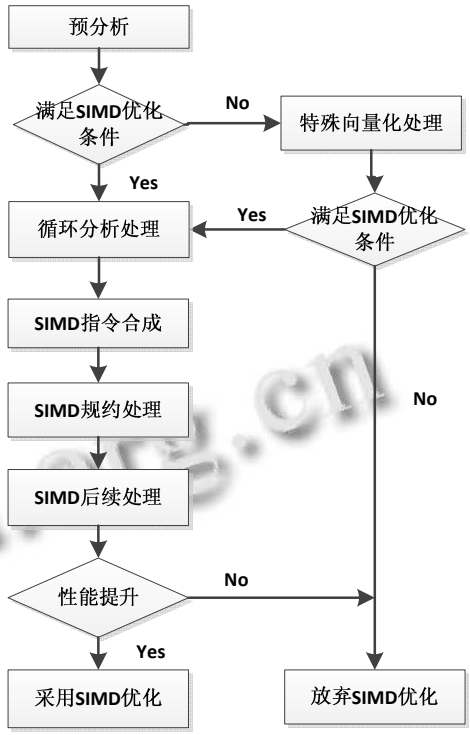

图 5 基于 WHIRL 树的 SIMD 算法流程

2.2 预分析

SIMD 优化能够大幅提高代码的效率, 然而有些循环是无法合成向量化指令的, 所以在处理之前进行预分析, 判断预处理的循环能否进行向量化. 对于具有下列特点的循环, 可能无法进行 SIMD 优化:

(1) 循环体中出现含有函数调用, 显式或者隐式 goto 语句, return、exit 等退出函数体的语句, OpenMP

函数调用, 或者嵌入式汇编的其中一种或多种情况时, 可能无法进行 SIMD 优化.

(2) 目前, OCC 的 SIMD 优化仅针对最内层循环, 对于不是最内层的循环, 不进行 SIMD 优化.

(3) 有些循环边界变量过于复杂, 可能无法标准化, 或者循环索引在退出循环后依然活跃 , 这种情况下 OCC 将放弃 SIMD 优化.

(4) SIMD 优化主要针对循环体中的数组进行优化, 如果循环体中没有连续的数组引用, 则无法进行向量化优化.

(5) 当迭代次数太少, SIMD 优化没有足够的效率提升, 不进行优化.

(6) 目前, OCC 的 SIMD 优化支持的向量化操作包括 load、store、加法、减法、乘法等, 当循环体中没有被支持的向量化操作时, 不进行 SIMD 优化.

在判定具有上述一种或几种特定的嵌套循环, 并不是完全放弃向量化优化, 而是将其转入特殊向量化化处理阶段, 而对于满足向量化优化条件的则直接进入预处理阶段.

2.3 特殊向量化处理

预分析中 SIMD 优化条件比较苛刻, 这可能会导致这样的情况: 一些代码表面上不满足向量化优化的条件, 但是经过循环分布、循环交换、以及特殊指令合成等特殊向量化处理手段后, 可以进行向量化.

特殊向量化处理的经典理论有循环交换(Loop Interchange)[13]和循环分布(Loop Fission)[14]. 当阻碍向量化的依赖关系由最内层循环而非外层循环携带时, 可以考虑交换内外层循环, 使嵌套循环中的内层循环不再携带依赖关系, 从而满足向量化优化的条件(具体可见图 6).

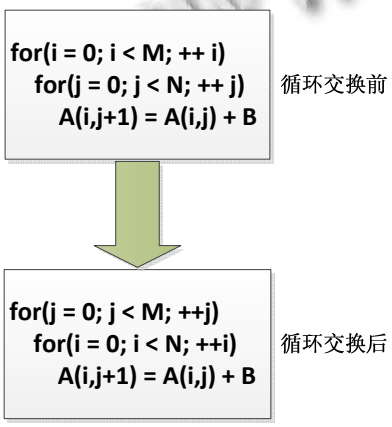

图 6　循环交换

循环分布将一个循环分拆为多个循环, 包括存在依赖关系和不存在依赖关系的循环, 对于不存在依赖关系的那些循环进行 SIMD 优化. 循环分布示意图见图 7.

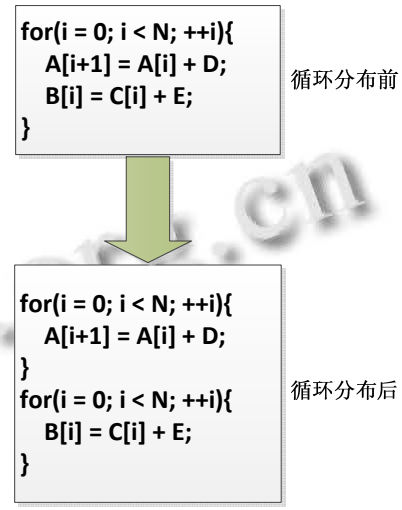

图 7　循环分布

除了循环分布和循环交换等经典的向量化优化方法, 针对 BWDSP 提供的特殊体系架构和指令集, 我们可以对循环体中可以合成特殊指令的部分操作先进行指令合成, 从而消除依赖, 最后满足 SIMD 优化实施条件后再进行 SIMD 优化. 下面, 以 max、min 指令合成为例来说明.

如图 8 所示, 在合成特殊指令之前, 由于 for 循环中含有 if 语句, 翻译后的 whirl 语言会存在跳转指令, 所以一般情况下无法进行 SIMD 优化. 但是仔细观察后可以看出, 循环体中实际为求最大值, 可以合成 BWDSP 中的 max 指令后再进行 SIMD 优化.

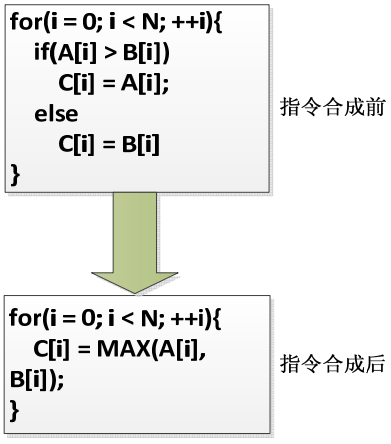

图 8　MAX 指令合成

2.4 预处理

经过预分析阶段后, OCC 将对满足 SIMD 优化条件的循环进行预处理. 预处理包括数据流分析、对数组分析、归约分析、规约分析等.

(1)数据流分析: 分析嵌套的循环之间和指令之间的依赖关系, 建立 du 链和数据流图.

(2)统一数组访问方式: 对数组的访问方式全部统一为基址加偏移地址的方式.

(3)对数组对齐分析: 检查数组引用的连续性与循环变量的跨度之间是否保持一致.

(4)合成单双字判定: BWDSP 分别提供了单字和双字两种不同的指令形式. 而最终合成的指令采用哪种形式则需要根据循环步长和循环体对连续地址的访问跨度来判定.

循环步长可以分为两种形式, i = i + 1 和 i = i + C, 其中 C 为非 1 的常量. 对于 i = i + V, 其中 V 为循环变量时, 循环无法合成向量化指令.

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(C > 1)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.

(5)归约分析: 对循环体进行分析, 判定是否需要进行归约处理.

2.5 SIMD 指令合成

经过预处理之后, 对于循环体中的 whirl node 的处理分为两部分, 首先将普通的单数据流节点变换成多数据流节点(四簇同时操作), 然后对相关的循环边界进行处理.

(1)单簇操作扩展成四簇操作

BWDSP 的向量化指令的多数据流体现在四簇同时存取及进行运算. 在 whirl 树上, 这些指令通常的体现都是单簇指令(单数据流), 所以在 SIMD 指令合成阶段, 对这些单簇操作替换成多(四)簇操作.

(2)循环变量边界处理

对 whirl 树进行了从单簇到四簇的替换后, 循环边界变量发生了变化, 这时候需要对循环变量进行相关的处理. 如果循环体中操作数刚好为 4 的倍数, 并且经过预处理后全部可以正常进行 SIMD 优化, 则不再需要对循环边界进行特殊处理, 只需要处理循环步长的递增值; 如果循环边界不为 4 的整数倍, 在 SIMD 优化后, 有多余的循环体没有处理, 则不仅需要对循环步长进行处理, 并且需要在向量化后的循环体后新增加一个小的收尾循环.

对于收尾循环的生成过程包括以下步骤:

(a) 拷贝优化之前的 WN 操作符到新循环体, 分别拷贝向量化的循环上界以及优化之前的循环上界作为新循环的循环下届和上界.

(b) 更新新循环下界、上界的 d-u 链.

(c) 将新生成的循环插入 SIMD 优化循环的父节点的孩子节点中, 和优化过的循环成为兄弟节点.

(d) 更新相关的依赖信息.

2.6 SIMD 后续规约处理

对于部分循环体, 需要在完成 SIMD 优化后进行特殊的归约处理. 如下面对数组求和的程序, 需要继续归约处理. 为便于描述, 假定 N 为 4 的整数倍, 采用高级语言和 BWDSP 易读汇编语句结合的方式来说明对归约求和的优化结果. 对于归约求和, 先用单字取值指令取出连续的 a[i]、a[i+1]、a[i+2]、a[i+3]分别放到四个簇的 R0 上, 然后四个簇同时求和. 最后, 插入簇间传输指令, 将 y、z、t 上的 R0 分别传输到 x 簇上的 R1、R2、R3 上, 最后归约求出正确的和. 归约求和的程序和归约处理后的伪汇编如图 9 所示.

归约处理的步骤包括如下:

(1)分析循环体是否需要归约, 如果需要进行特殊归约处理, 进行记录, 保存相关变量(在预处理中处理);

(2)在循环体前新定义部分结果变量, 在 whirl 树上增加相关定义语句;

(3)在循环体中进行 SIMD 优化, 对部分结果变量进行向量化求值;

(4)如果在 remainder_loop 中有部分未处理, 需要增加一个部分结果用来存储在 remainder_loop 中的计算结果;

(5)最后对所有的部分结果进行归约处理, 并且更新 du 链信息.

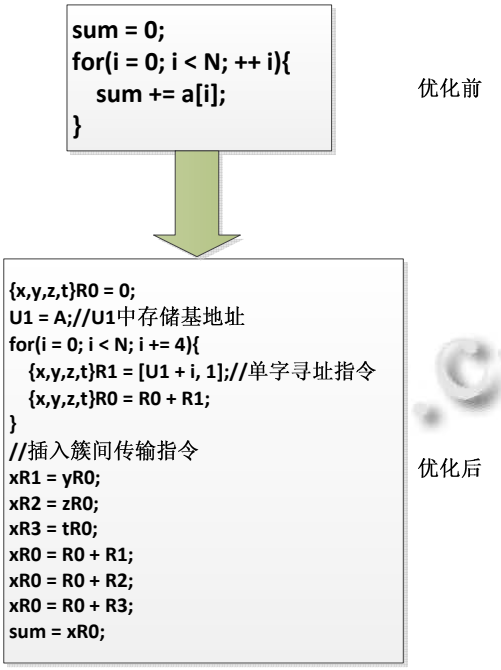

图 9　SIMD 后续规约

3　实验

为了验证优化方案的效果, 我们选取了 3 个典型的简单循环计算作为实验用例(图 10). 这些例子在 DSP 领域中都有着非常广泛的应用. 其中(a)向量加运算和(b)向量乘运算能够合成双字进行向量化优化, (c)和(d)只能进行单字的向量化优化.

```
for( i = 0 ; i < N ; ++i )    // N=2048
    A[i] = B[i] + C[i];
```

(a) 向量加运算

```
for( i = 0 ; i < N ; ++i )    // N=2048
    A[i] += B[i] * C[i];
```

(b) 卷积运算

```
for ( i = 0 ; i < N/2 ; ++i ) {    // N=2048
    A[2 * i] += B[i];
    A[2 * i + 1] *= C[i];
}
```

(c) 只能合成单字的向量运算

```
for ( ; i<N/4 ; ++i ) {       // N=4096
    A[4*i] += B[i];
    A[4*i + 1] *= C[i];
    A[4*i + 2] -= B[i];
    A[4*i + 3] += B[i];
}
```

(d) 只能合成单字的向量运算

图 10　实验用例

表 1　优化前和优化后的时钟周期数(cycles)

测试用例	关闭向量化	启用向量化	加速比
向量加运算	18477	3123	5.92
卷积运算	18477	2607	7.09
单字循环 1	15392	3677	4.19
单字循环 2	27642	6745	4.10

表 1 列出了这 4 个用例的测试结果. 可以看出, 使用基于 WHIRL 树的 SIMD 算法, 对能够合成双字的循环能够获得平均 6 的加速比, 对只能够合成单字的循环, 也能获得平均4.15的加速比. 文献[15]提出了一种基于动态循环展开因子的启发式 SIMD 优化算法 SLUS, 在 BWDSP100 上取得了 200～300%的效率提升. 相比 SLUS, 我们能够获得更好的效率提升.

4　结语

本文在编译基础设施 OPEN64 的基础上, 研究并实现了基于中间语言 WHIRL 的 SIMD 向量化算法. 实验表明, 算法应用在多簇 VLIW DSP, 能够取得很好的加速效果.

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(英文影印版).北京:机械工业出版社,2006:337–395.

2 CETC 38, BWDSP 100 Software User Manual.

3 High Performance Computing Tools Group. Overview of the Open64 Compiler Infrastructure. University of Houston Computer Science Department. November 12, 2002.

4 SIMD. en.wikipedia.org/wiki/SIMD.

5 张为华,臧斌宇.SIMD 编译优化技术研究概述.中国计算机学会通讯,2007,3(2):27–36.

6 Talla D, John LK, Burger D. Bottlenecks in multimedia

processing with SIMD style extensions and architectural enhancements. IEEE Trans. Comput, 2003, 52(8): 1015–1031.

7 Kim S, Han H. Efficient simd code generation for irregular kernels. ACM SIGPLAN Notices, 2012, 47(8): 55–64.

8 Manniesing R, Karkowski I, Corporaal H. Automatic SIMD parallelization of embedded applications based on pattern recognition. Euro-Par 2000 Parallel Processing. Springer Berlin Heidelberg. 2000. 349–356.

9 Pokam G, Simonnet J, Bodin F. A retargetable preprocessor for multimedia instructions. Proc. of the 9th Workshop on Compilers for Parallel Computers. 2001. 291–301.

10 Wang L, Zhang C, Huang YZ. An optimization approach for SIMD alignment in mathematical functions. Advances in Computer, Communication, Control and Automation. Springer Berlin Heidelberg. 2012. 37–43.

11 Ren G, Wu P, Padua D. Optimizing data permutations for SIMD devices. Proc. of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM Press. 2006. 118–131.

12 Chakrabarti G, Chow F. Structure layout optimizations in the Open64 compiler: design, implementation and measurements. Gautam Chakrabarti, Open64 Workshop at CGO. 2008.

13 Ng JL, Krishnaiyer R, Ostanevich A Y. Data dependence testing for loop fusion with code replication, array contraction, and loop interchange. U.S. Patent 8,677,338[P]. 2014-3-18.

14 Meenakshi K, Vemuri R, Govindarajan S, Ouaiss I. An automated temporal partitioning and loop fission approach for FPGA based reconfigurable synthesis of DSP applications. Proc. of the 36th annual ACM/IEEE Design Automation Conference. ACM. 1999. 616–622.

15 Yang Y, Gu N, Ren K, et al. An approach to enhance loop performance for multicluster VLIW DSP processor. 2014 27th International Conference on Architecture of Computing Systems (ARCS). VDE. 2014. 1–8.