(19)中华人民共和国国家知识产权局

(12)发明专利申请

(10)申请公布号 CN 103440229 A
(43)申请公布日 2013.12.11

(21)申请号 201310349628.8

(22)申请日 2013.08.12

(71)申请人 浪潮电子信息产业股份有限公司
地址 250014 山东省济南市高新区舜雅路1036号

(72)发明人 吴庆

(51)Int.Cl.
G06F 17/16(2006.01)

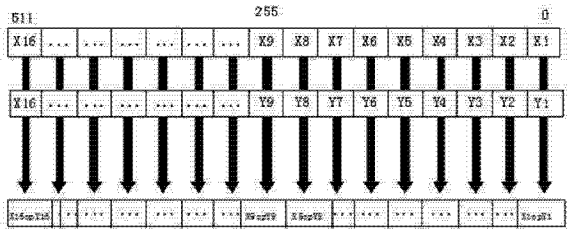
权利要求书4页 说明书13页 附图2页

(54)发明名称

一种基于MIC架构处理器的向量化优化方法

(57)摘要

本发明提供了一种基于MIC架构处理器的向量化优化方法，涉及算法数据依赖关系分析、算法向量化调整优化、向量化编译三个主要步骤，具体内容包括：算法的数据依赖分析、算法的向量化优化调整、编译器自动向量化技术、用户介入的向量化优化方法等。本发明提供的方法适用于MIC架构处理器平台的软件优化，指导软件开发人员以较短的开发周期，较低的开发成本，快速高效地对现有软件，尤其是核心算法进行向量优化改造，实现软件对向量处理器计算资源利用的最大化，最大限度地缩短软件运行时间，显著提高硬件资源利用率，提高软件的计算效率和软件整体性能。



CN 103440229 A

1. 一种基于MIC架构处理器的向量化优化方法，其特征在于，内容包括：1）对目标循环进行向量化可行性分析；2）向量化优化；3）编译器自动向量化；4）基于向量化编译指示的向量化；5）算法向量化改造；6）向量化正确性验证；通过重复以上迭代调优过程，以实现循环向量化率最大化，其中：

1）对目标循环进行向量化可行性分析，是对需要向量化的循环进行数据依赖分析，排除循环迭代依赖，所谓循环迭代依赖，是指循环间存在前后依赖，导致循环不具有完全独立性，不能并行处理，从而使该循环不能被向量化；

2）向量化优化，是在数据依赖关系的基础上，采用多种方法手段，使循环被向量化处理；

3）编译器自动向量化，是编译器会自动分析循环间的数据依赖关系，并自主决定是否进行循环向量化；

4）基于向量化编译指示的向量化，是基于数据依赖分析结果，通过在相应循环外添加编译指示语句，指导编译器对该循环进行向量化编译；

5）算法向量化改造，是基于数据依赖分析结果，对算法／循环进行优化改造，包括数据结构调整、循环拆分、循环合并、循环嵌套顺序调整；

6）向量化正确性验证，是验证向量化的正确性，体现在输出结果正确，误差在可以接受的范围内，具体实施步骤、方法细则如下：

1）循环向量化可行性分析：

编译器自动向量化过程中，可能收到某个循环无法被向量化的编译信息报告，很多时候无法向量化的原因都是循环间存在着变量依赖关系，通过阅读源代码，理解算法，必要时进行测试，确认代码中各循环结构间的数据依赖关系，进行数据依赖关系分析，是为了下一步对循环进行优化调整、并且介入编译器向量化编译行为；

向量化处理的实质，就是将原本串行循环处理的计算任务，转换成若干循环同时处理的方式，因此，各次循环之间不能有前后的数据依赖关系，在实际操作中，编译器通过设置私有变量、添加原子阻塞操作，实现广义的向量化，这取决于向量化处理器硬件的设计及其所支持的指令集的设计，因此，循环可向量化的必要条件是：

a）循环之间不存在依赖关系，也就是说，所有的循环能同时执行且互不干扰；

b）必须是内层循环，在一个嵌套的循环中，向量器只能尝试向量化最内层的循环，查看向量器的输出信息能知道循环是否被向量化以及原因，如果影响性能的关键循环没有向量化，需要做一些算法调整，比如调整嵌套循环的顺序；

另外，除了具备以上必要条件外，还要注意以下几点：

（a）向量化处理的数据类型尽量一致，需要向量化处理的语句，其包含的变量尽可能做到长度一致，即数据类型尽可能一致，尽量避免在同一表达式中同时出现单精度和双精度变量；

（b）向量化语句中尽可能避免函数调用：表达式中调用函数，会增加语句的复杂度，干扰编译器实施自动向量化，即使是标准的数学函数，也要尽量避免，如果一定要使用，也要尽可能使函数精度与变量精度一致；

2）向量化优化，采用以下方法，实现循环的向量化处理；

a）编译器自动向量化

编译器自动向量化依赖于编译器自身的能力来消除内存引用二义性，Intel C/C++ 编译器默认向量化编译选项为 -vec，即默认情况下向量化是打开的，若关闭向量化，在编译选项中添加 -no-vec，编译器在对源代码进行编译时，会输出编译信息 / 报告，某些编译器有多个编译信息输出级别，其编译信息输出级别可使用 -vec-report *level* 控制，通过编译器输出的编译信息报告，以了解编译的详细细节，包括某个循环是否被向量化，向量化失败的原因，这些信息能为向量优化提供依据和指导 ；

b）基于向量化编译指示的向量化

向量化编译指示，能更好地指导编译器进行数据依赖分析，从而更好地向量化代码，包括

__declspec(align(n)) 声明能够使编译器克服硬件对齐的限制，restrict 修饰词和自动向量化提示解决了由于作用范围、数据依赖和二义性等产生的问题，SIMD 编译指示使得最内层的循环被强制向量化，使用向量化编译指令是有风险的，使用的前提条件是程序员必须确保不存在数据依赖 ；其中 SIMD 编译指示有五个可供选择的子句来指导编译器执行何种向量化，恰当地使用这些子句，会让编译器获得足够的信息来产生正确的向量化代码，其中 ：

（1）vectorlength(num1, num2, …, numN)

指导向量优化单元可以从指定的若干向量长度 (VL)num1, num2, … , numN 中选择来向量化循环，对于选定的 VL，向量循环的每一次执行的计算工作相当于原来标量循环 VL 次执行的计算工作，多个向量长度子句会合并成一个集合 ；

（2）private(expr1, expr2, …, exprN)

指导向量优化单元使得这些左值 (L-value) 表达式 expr1, exper2, ..., exprN 对于每一次循环都是私有的，多个 private 子句会合并成一个集合，左值表达式的初值将被广播到所有的私有子句，除非编译器能够判定初值未在循环体内使用 ；左值表达式的终值也会被从最后一次执行的循环体复制出来 , 除非编译器能够判定终值未在循环后被使用 ；

（3）linear(var1:step1, var2:step2, …, varN:stepN)

指导编译器每一次标量循环的执行时，var1 的值增加 step1, var2 的值增加 step2, 依此类推，相应地，每次向量循环的执行，使得这些变量的值分别增加 VL*step1,VL*step2, …, VL*stepN，多个 linear 子句会被合并成一个集合，如果 var 被赋予两个或多个 step 值，会产生一个编译错误 ；

（4）reduction(oper:var1,var2,…,varN)

指导编译器对于变量 var1, var2, …, varN 执行向量化规约操作 oper，一个 SIMD 编译指示有多个归约子句，执行相同或者不同的操作，如果一个变量 var 与两个或多个不同的归约操作 oper 有关 , 会产生一个编译错误 ；

（5）[no]assert

指导编译器当向量化失败时是否报错，缺省是不报错的，一个 SIMD 编译指令不应该存在多个该子句，否则会产生一个编译错误，为了向量化一个包含潜在依赖关系的循环，用户经过数据依赖分析后，确认其不存在循环间数据依赖，可加上 #pragma ivdep 提示编译器忽略存在的数据依赖关系 ；

当向量化以上代码段中的循环时，编译器会认为该循环存在循环间依赖，即第 j 次循

环依赖第 j+k 次循环的结果，这种依赖关系称为交叉迭代依赖，而如果确定 k>16，超过 MIC VPU 的向量处理宽度，循环间就不存在实际意义上的数据依赖，加上 #pragma ivdep 指示编译器忽略数据的依赖关系并尝试进行向量化，甚至使用 #pragma simd 强制向量化该循环，如果 L<k<16，那么使用 #pragma simd vectorlength(L) 强制向量化该循环，并且能保证结果的正确性；

3）算法向量化改造

如果采用以上两种向量化方式，还有部分循环无法实现向量化，在数据依赖分析的基础上，对算法进行深度优化改进，向量化改造方法有：

a）调整嵌套循环的顺序；

b）自动向量化只能对嵌套中的最内层的循环进行向量化，然而内层循环向量化效果未必最好，通过调整嵌套循环的顺序达到更好的向量化效果，如调整之后的向量化能实现更好的连续访问；

c）拆分循环

在某些情况下，除了最内层的循环比较耗时外，其它不在最内层循环的代码也比较耗时，而这部分代码是无法自动向量化的，为此，采取拆分循环的方法实现更多的自动向量化，通过对循环的拆分，能使更多的代码自动向量化，获取更好的向量化性能；

d）手写 SIMD 指令向量化

第一代 Intel MIC 产品为 KNC(Knights Corner)，Knights Corner Instructions 是 KNC 支持的 SIMD 指令的总称，是类似于 SSE、AVX 的指令集，通过使用 Knights Corner 指令，能细粒度地控制向量化运算；

Knights Corner Instructions 分类：

（1）Knights Corner 指令 Knights Corner instruction 是指具体的 SIMD 指令，是汇编指令集中关于 SIMD 的子集；

（2）内建 Knights Corner（Intrinsics of Knights Corner）是对 Knights Corner 指令的封装，几乎涉及到所有指令，认为这些函数和数据类型是 C/C++ 的内建类型；

Knights Corner 类库 Knights Corner Class Libraries 是为了方便使用 Knights Corner 指令而做的封装，让程序员尽量简单地使用 SIMD 指令，介于引语方式和 SIMD 代码之间；其支持整型和浮点型数据；

4）向量化正确性验证

编译源代码，然后运行程序，检查程序的输出结果，验证向量化的正确性，向量化可能会带来精度损失，必要时通过编译器的 -fp-model 选项，调整向量化的精度；

5）迭代调优

重复以上过程，以实现循环向量化率最大化，从而使 MIC 处理器 VPU 的计算性能尽可能发挥出来；

6）性能测试及分析，包括：

（1）测试环境

（2）性能测试结果

（3）性能测试结果分析

利用该方法对矩阵乘法应用案例进行向量化改造后，显著地提升了该 MIC 模块的运行

效率，获得了较高的性能加速比。

2. 根据权利要求 1 所述的方法，其特征在于，编译过程中所使用的编译器，是支持 MIC 架构及其指令集的任意编译器，该编译器生成的目标代码，直接或通过后续编译处理后，能够运行于 MIC 架构处理器。

一种基于 MIC 架构处理器的向量化优化方法

技术领域

[0001] 本发明涉及计算机高性能计算领域、科学计算领域，具体涉及一种基于 MIC 架构处理器的向量化优化方法。

背景技术

[0002] 自从 1996 年 Intel 在奔腾处理器上集成了 MMX 后，越来越多的通用处理器上集成了 SIMD(Single Instruction Multiple Data，单指令多数据流）硬件扩展，这种集成了 SIMD 处理架构的处理器，称为向量处理器。向量处理器的应用也越来越广泛，从最初的多媒体应用扩展到各个应用领域，尤其在高性能计算领域，海量数据、大规模并行处理需求，对处理器的计算能力提出严峻挑战，向量化的并行处理，能有效提高并行处理效率和计算密度，提高硬件资源利用率，进而降低计算成本。

[0003] 新的 MIC 架构协处理器具备当前最宽的向量宽度，它构建在至强处理器的并行架构之上，通过集成众多低功耗内核，每一个处理器核具备一个 512 位的 SIMD 处理单元和很多新的向量运算指令，MIC 架构处理器创造了在一个芯片上的超级计算机，超过每秒一万亿次的计算能力。

[0004] 随着 MIC 架构处理器的推广，其强大的 SIMD 扩展技术将被广泛应用，不仅为高性能计算提供了新的解决问题、提升性能的途径，也带来了一个新的问题——如何快速、高效地实现可靠的向量化并行处理，从而充分释放 MIC 处理器的计算潜力？这是摆在软件工程师面前的现实挑战。

发明内容

[0005] 本发明的目的是提供一种基于 MIC 架构处理器的向量化优化方法。

[0006] 本发明的目的是按以下方式实现的，内容包括：1）对目标循环进行向量化可行性分析；2）向量化优化；3）编译器自动向量化；4）基于向量化编译指示的向量化；5）算法向量化改造；6）向量化正确性验证；通过重复以上迭代调优过程，以实现循环向量化率最大化，其中：

1）对目标循环进行向量化可行性分析，是对需要向量化的循环进行数据依赖分析，排除循环迭代依赖，所谓循环迭代依赖，是指循环间存在前后依赖，导致循环不具有完全独立性，不能并行处理，从而使该循环不能被向量化；

2）向量化优化，是在数据依赖关系的基础上，采用多种方法手段，使循环被向量化处理；

3）编译器自动向量化，是编译器会自动分析循环间的数据依赖关系，并自主决定是否进行循环向量化；

4）基于向量化编译指示的向量化，是基于数据依赖分析结果，通过在相应循环外添加编译指示语句，指导编译器对该循环进行向量化编译；

5）算法向量化改造，是基于数据依赖分析结果，对算法／循环进行优化改造，包括数据

结构调整、循环拆分、循环合并、循环嵌套顺序调整；

6）向量化正确性验证，是验证向量化的正确性，体现在输出结果正确，误差在可以接受的范围内，具体实施步骤、方法细则如下：

1）循环向量化可行性分析：

编译器自动向量化过程中，可能收到某个循环无法被向量化的编译信息报告，很多时候无法向量化的原因都是循环间存在着变量依赖关系，通过阅读源代码，理解算法，必要时进行测试，确认代码中各循环结构间的数据依赖关系，进行数据依赖关系分析，是为了下一步对循环进行优化调整、并且介入编译器向量化编译行为；

向量化处理的实质，就是将原本串行循环处理的计算任务，转换成若干循环同时处理的方式，因此，各次循环之间不能有前后的数据依赖关系，在实际操作中，编译器通过设置私有变量、添加原子阻塞操作，实现广义的向量化，这取决于向量化处理器硬件的设计及其所支持的指令集的设计，因此，循环可向量化的必要条件是：

a）循环之间不存在依赖关系，也就是说，所有的循环能同时执行且互不干扰；

b）必须是内层循环，在一个嵌套的循环中，向量器只能尝试向量化最内层的循环，查看向量器的输出信息能知道循环是否被向量化以及原因，如果影响性能的关键循环没有向量化，需要做一些算法调整，比如调整嵌套循环的顺序；

另外，除了具备以上必要条件外，还要注意以下几点：

（a）向量化处理的数据类型尽量一致，需要向量化处理的语句，其包含的变量尽可能做到长度一致，即数据类型尽可能一致，尽量避免在同一表达式中同时出现单精度和双精度变量；

（b）向量化语句中尽可能避免函数调用：表达式中调用函数，会增加语句的复杂度，干扰编译器实施自动向量化，即使是标准的数学函数，也要尽量避免，如果一定要使用，也要尽可能使函数精度与变量精度一致；

2）向量化优化，采用以下方法，实现循环的向量化处理；

a）编译器自动向量化

编译器自动向量化依赖于编译器自身的能力来消除内存引用二义性，Intel C/C++ 编译器默认向量化编译选项为 -vec，即默认情况下向量化是打开的，若关闭向量化，在编译选项中添加 -no-vec，编译器在对源代码进行编译时，会输出编译信息 / 报告，某些编译器有多个编译信息输出级别，其编译信息输出级别可使用 -vec-report *level* 控制，通过编译器输出的编译信息报告，以了解编译的详细细节，包括某个循环是否被向量化，向量化失败的原因，这些信息能为向量优化提供依据和指导；

b）基于向量化编译指示的向量化

向量化编译指示，能更好地指导编译器进行数据依赖分析，从而更好地向量化代码，包括

__declspec(align(n)) 声明能够使编译器克服硬件对齐的限制，restrict 修饰词和自动向量化提示解决了由于作用范围、数据依赖和二义性等产生的问题，SIMD 编译指示使得最内层的循环被强制向量化，使用向量化编译指令是有风险的，使用的前提条件是程序员必须确保不存在数据依赖；其中 SIMD 编译指示有五个可供选择的子句来指导编译器执行何种向量化，恰当地使用这些子句，会让编译器获得足够的信息来产生正确的向量化代

码，其中：

(1) vectorlength(num1, num2, …, numN)

指导向量优化单元可以从指定的若干向量长度 (VL)num1, num2, … , numN 中选择来向量化循环，对于选定的 VL，向量循环的每一次执行的计算工作相当于原来标量循环 VL 次执行的计算工作，多个向量长度子句会合并成一个集合；

(2) private(expr1, expr2, …, exprN)

指导向量优化单元使得这些左值(L-value)表达式expr1, exper2, ..., exprN 对于每一次循环都是私有的，多个 private 子句会合并成一个集合，左值表达式的初值将被广播到所有的私有子句，除非编译器能够判定初值未在循环体内使用；左值表达式的终值也会被从最后一次执行的循环体复制出来，除非编译器能够判定终值未在循环后被使用；

(3) linear(var1:step1, var2:step2, …, varN:stepN)

指导编译器每一次标量循环的执行时，var1 的值增加 step1, var2 的值增加 step2, 依此类推，相应地，每次向量循环的执行，使得这些变量的值分别增加 VL*step1,VL*step2, …, VL*stepN,多个 linear 子句会被合并成一个集合，如果 var 被赋予两个或多个 step 值，会产生一个编译错误；

(4) reduction(oper:var1,var2,…,varN)

指导编译器对于变量var1, var2, …, varN 执行向量化规约操作oper，一个SIMD 编译指示有多个归约子句，执行相同或者不同的操作，如果一个变量 var 与两个或多个不同的归约操作 oper 有关，会产生一个编译错误；

(5) [no]assert

指导编译器当向量化失败时是否报错，缺省是不报错的，一个 SIMD 编译指令不应该存在多个该子句，否则会产生一个编译错误，为了向量化一个包含潜在依赖关系的循环，用户经过数据依赖分析后，确认其不存在循环间数据依赖，可加上#pragma ivdep提示编译器忽略存在的数据依赖关系；

当向量化以上代码段中的循环时，编译器会认为该循环存在循环间依赖，即第 j 次循环依赖第 j+k 次循环的结果，这种依赖关系称为交叉迭代依赖，而如果确定 k>16，超过 MIC VPU 的向量处理宽度，循环间就不存在实际意义上的数据依赖，加上 #pragma ivdep 指示编译器忽略数据的依赖关系并尝试进行向量化，甚至使用 #pragma simd 强制向量化该循环，如果 L<k<16，那么使用 #pragma simd vectorlength(L) 强制向量化该循环，并且能保证结果的正确性；

3) 算法向量化改造

如果采用以上两种向量化方式，还有部分循环无法实现向量化，在数据依赖分析的基础上，对算法进行深度优化改进，向量化改造方法有：

a) 调整嵌套循环的顺序

b) 自动向量化只能对嵌套中的最内层的循环进行向量化，然而内层循环向量化效果未必最好，通过调整嵌套循环的顺序达到更好的向量化效果，如调整之后的向量化能实现更好的连续访问；

c) 拆分循环

在某些情况下，除了最内层的循环比较耗时外，其它不在最内层循环的代码也比较耗

时，而这部分代码是无法自动向量化的，为此，采取拆分循环的方法实现更多的自动向量化，通过对循环的拆分，能使更多的代码自动向量化，获取更好的向量化性能；

d) 手写 SIMD 指令向量化

第一代 Intel MIC 产品为 KNC(Knights Corner)，Knights Corner Instructions 是 KNC 支持的 SIMD 指令的总称，是类似于 SSE、AVX 的指令集，通过使用 Knights Corner 指令，能细粒度地控制向量化运算；

Knights Corner Instructions 分类：

(1) Knights Corner 指令 Knights Corner instruction 是指具体的 SIMD 指令，是汇编指令集中关于 SIMD 的子集；

(2)内建 Knights Corner (Intrinsics of Knights Corner)是对 Knights Corner 指令的封装，几乎涉及到所有指令，认为这些函数和数据类型是 C/C++ 的内建类型；

Knights Corner 类库 Knights Corner Class Libraries 是为了方便使用 Knights Corner 指令而做的封装，让程序员尽量简单地使用 SIMD 指令，介于引语方式和 SIMD 代码之间；其支持整型和浮点型数据；

4) 向量化正确性验证

编译源代码，然后运行程序，检查程序的输出结果，验证向量化的正确性，向量化可能会带来精度损失，必要时通过编译器的 -fp-model 选项，调整向量化的精度；

5) 迭代调优

重复以上过程，以实现循环向量化率最大化，从而使 MIC 处理器 VPU 的计算性能尽可能发挥出来；

6) 性能测试及分析，包括：

(1) 测试环境

平台	InspurNF5280M3
CPU	IntelXeonCPUE56753.07GHz，双路 8 核
Memory	DDR31333MHz128GB
MIC	KNC，60 核，1.0GHz，GDDR58GBmemory5.5GT/s
OS	RedHatEnterpriseLinuxServerrelease6.1,64bit
编译器	icc
测试用例	4096*4096 矩阵乘法

(2) 性能测试结果

程序版本	版本说明	时间 (s)
P_baseline	CPU 单线程基准版	312.83
P_OMP	CPU 多线程 + 自动向量化版	170.83
P_MIC_base	MIC 多线程 + 自动向量化版	174
P_baseline_vec	CPU 单线程 + 算法向量化改造 + 向量化指示版	25
P_OMP_vec	CPU 多线程 + 算法向量化改造 + 向量化指示版	4.53
P_MIC_vec	MIC 多线程 + 算法向量化改造 + 向量化指示版	3.43
P_MIC_simd	MIC 多线程 + 算法向量化改造 + 向量化指示 +simd 指令版	2

(3) 性能测试结果分析

利用该方法对矩阵乘法应用案例进行向量化改造后，显著地提升了该模块的运行效率，获得了较高的性能加速比。

[0007] 编译过程中所使用的编译器，是支持 MIC 架构及其指令集的任意编译器，该编译器生成的目标代码，直接或通过后续编译处理后，能够运行于 MIC 架构处理器。

[0008] 本发明的有益效果是：该方法广泛适用于MIC架构处理器并行处理的应用场合，指导软件开发人员以较短的开发周期，较低的开发成本，快速高效地对现有软件进行向量化优化改造，实现软件对系统资源利用最优化，显著提高硬件资源利用率和软件的计算效率，从而大大提升软件整体性能。

附图说明

[0009] 图1是单精度浮点数据向量化处理示意图；

图2是向量化的层次结构示意图。

具体实施方式

[0010] 参照说明书附图对本发明的方法作以下详细地说明。

[0011] 本发明提供了一种基于MIC架构处理器的向量化优化方法。其主要内容是提供一种利用MIC架构协处理器计算设备，最大化提高MIC硬件资源利用率，从而提升MIC处理器平台上软件运行效能的方法。该方法提出，基于MIC架构处理器的向量化优化流程如下：

1）对目标循环进行向量化可行性分析；

2）向量化优化

3）编译器自动向量化

4）基于向量化编译指示的向量化

5）算法向量化改造

6）向量化正确性验证。

[0012] 重复以上过程，迭代调优，以实现循环向量化率最大化。

[0013] 3、具体实施方式

本发明的目的在于提供一种基于MIC处理器的向量化优化方法。

[0014] 为了使本发明的目的、技术方案和优点更加清晰，下面结合附图和实施例，对本发明作以下详细说明。

[0015] 首先，简要介绍向量化处理原理及MIC处理器的向量处理单元VPU（Vector Process Unit）的架构。MIC处理器核的VPU支持512bit位宽的KCi向量指令，支持16*32bit或8*64bit等多种处理模式，即向量化宽度为8或16。512位相当于16个单精度浮点型数据的长度，单精度浮点数据向量化处理示意图如图1所示：

例如向量加操作C[0~15]=A[0~15]+B[0~15]（A、B、C均为float型数据），没有使用向量化时这个操作需要16次加运算，而向量化之后，把A、B、C放到向量寄存器中，进行一次向量加操作即可完成原来的16次加操作，因此，向量化可以大大提高计算速度。

[0016] 以下说明都基于intel的编译器和C语言进行阐述。

[0017] 一种基于MIC架构处理器的向量化优化实施步骤、方法细则如下：

6）循环向量化可行性分析。

[0018] 编译器自动向量化过程中，可能收到某个循环无法被向量化的编译信息报告。很多时候无法向量化的原因都是循环间存在着变量依赖关系。

[0019] 通过阅读源代码，理解算法，必要时进行测试，确认代码中各循环结构间的数据依赖关系。进行数据依赖关系分析，是为了下一步对循环进行优化调整、并且介入编译器向量

化编译行为。

[0020] 向量化处理的实质，就是将原本串行循环处理的计算任务，转换成若干循环同时处理的方式，因此，原理上，各次循环之间不能有前后的数据依赖关系。当然，在实际操作中，编译器可以通过设置私有变量、添加原子阻塞操作等方式，实现广义的向量化，这取决于向量化处理器硬件的设计及其所支持的指令集的设计。

[0021] 因此，循环可向量化的必要条件是：

a) 循环之间不存在依赖关系。

[0022] 也就是说，所有的循环能同时 执行且互不干扰。例如：

```
for (int i=0; i<1000; i++)
 {
s1: a[i] = b[i] * T + d[i] ;
s2: b[i] = (a[i] + b[i])/2;
s3: c = c + b[i];
 }
```

等价于下面的操作：

```
for (int i=0; i<1000; i++) a[i] = b[i] * T + d[i] ;
for (int i=0; i<1000; i++) b[i] = (a[i] + b[i])/2;
for (int i=0; i<1000; i++) c = c + b[i];
```

因此，这个循环是可以被向量化的。

[0023] 再看一个例子：

```
for (int i=1; i<1000; i++)
 {
s1: a[i] = a[i-1] * b[i];
 }
```

无论如何，这个循环是不能被向量化的，因为 a[i] 在每次迭代中都依赖前一次迭代的结果。我们称这是一个交叉迭代的数据依赖或者“flow dependence”，这样的循环不能被编译器向量化。

[0024] b) 必须是内层循环。

[0025] 在一个嵌套的循环中，向量器只能尝试向量化最内层的循环，查看向量器的输出信息可以知道循环是否被向量化以及原因，如果影响性能的关键循环没有向量化，你可能需要做一些算法调整，比如调整嵌套循环的顺序。

[0026] 另外，除了具备以上必要条件外，还要注意以下几点：

a) 向量化处理的数据类型尽量一致

需要向量化处理的语句，其包含的变量尽可能做到长度一致，即数据类型尽可能一致。如，尽量避免在同一表达式中同时出现单精度和双精度变量。

[0027] 7) 向量化语句中尽可能避免函数调用：表达式中调用函数，会增加语句的复杂度，干扰编译器实施自动向量化，即使是标准的数学函数，也要尽量避免，如果一定要使用，

也要尽可能使函数精度与变量精度一致，如：

```
int fun(float* a, float* b,int N)
{
…
for(int i=0; i<N; i++)
{
b[i] = sinf(a[i]);
}
…
}
```

8) 向量化优化

采用以下多种方法，实现循环的向量化处理。

[0028] a) 编译器自动向量化

编译器自动向量化依赖于编译器自身的能力来消除内存引用二义性。Intel C/C++ 编译器，默认向量化编译选项为 -vec，即默认情况下向量化是打开的，若关闭向量化可以在编译选项中添加 -no-vec。编译器在对源代码进行编译时，会输出编译信息 / 报告，某些编译器有多个编译信息输出级别，其编译信息输出级别可使用 -vec-report *level* 控制，见下表：

-vec-report[*level*]	含义
0	不显示诊断信息。
1	只显示已向量化的循环(默认值)。
2	显示已向量化和未向量化的循环。
3	显示已向量化和未向量化的循环以及数据依赖信息。
4	只显示未向量化的循环。
5	显示未向量化的循环以及数据依赖信息。

通过编译器输出的编译信息报告，可以了解编译的详细细节，比如某个循环是否被向量化，向量化失败的原因等，这些信息可以为向量优化提供依据和指导。

[0029] b) 基于向量化编译指示的向量化

向量化编译指示，可以更好地指导编译器进行数据依赖分析，从而更好地向量化代码。

[0030] 例如 __declspec(align(n)) 声明能够使编译器克服硬件对齐的限制。restrict 修饰词和自动向量化提示解决了由于作用范围、数据依赖和二义性等产生的问题。SIMD 编译指示使得最内层的循环被强制向量化。使用向量化编译指令是有风险的，使用的前提条件是程序员必须确保不存在数据依赖。

[0031] Intel 编译器向量化编译指示如下表：

功能	说明
变量修饰符	
__declspec(align(n))	指导编译器将变量按照n 字节对齐，变量的地址是address mod n=0。
__declspec(align(n,off))	指导编译器将变量按照n 字节再加上off 字节的偏移量进行对齐。变量的地址是address mod n=off。
restrict	允许消除别名假定中存在的二义性，从而能够更大程度地向量化。
__assume_aligned(a,n)	当编译器无法获得对齐信息时，则假定数组a 已按照n字节对齐。
自动向量化编译提示	
#pragma ivdep	告诉编译器忽略可能存在的向量依赖关系
#pragma vector {aligned|unaligned|always}	指定循环向量化的方式
#pragma novector	指定循环不被向量化
强制向量化编译指示	
#pragma simd	强制向量化最内层循环

其中 SIMD 编译指示有五个可供选择的子句来指导编译器执行何种向量化。恰当地使用这些子句，会让编译器获得足够的信息来产生正确的向量化代码。

[0032] （6）vectorlength(num1, num2, …, numN)

指导向量优化单元可以从指定的若干向量长度 (VL)num1, num2, … , numN 中选择来向量化循环。对于选定的 VL，向量循环的每一次执行的计算工作相当于原来标量循环 VL 次执行的计算工作。多个向量长度子句会合并成一个集合。

[0033] （7）private(expr1, expr2, …, exprN)

指导向量优化单元使得这些左值(L-value)表达式expr1, exper2, ..., exprN 对于每一次循环都是私有的。多个 private 子句会合并成一个集合。左值表达式的初值将被广播到所有的私有子句，除非编译器能够判定初值未在循环体内使用；左值表达式的终值也会被从最后一次执行的循环体复制出来，除非编译器能够判定终值未在循环后被使用。

[0034] （8）linear(var1:step1, var2:step2, …, varN:stepN)

指导编译器每一次标量循环的执行时，var1 的值增加 step1, var2 的值增加 step2, 依此类推。相应地，每次向量循环的执行，使得这些变量的值分别增加 VL*step1, VL*step2, …, VL*stepN。多个 linear 子句会被合并成一个集合。如果 var 被赋予两个或多个 step 值，会产生一个编译错误。

[0035] （9）reduction(oper:var1,var2,…,varN)

指导编译器对于变量 var1, var2, …, varN 执行向量化规约操作 oper。一个 SIMD 编译指示可以有多个归约子句，执行相同或者不同的操作。如果一个变量 var 与两个或多

个不同的归约操作 oper 有关，会产生一个编译错误。

[0036] (10)[no]assert

指导编译器当向量化失败时是否报错，缺省是不报错。一个 SIMD 编译指令不应该存在多个该子句，否则会产生一个编译错误。

[0037] 下面，列举一个典型实例说明基于向量化编译指示的向量化。为了向量化一个包含潜在依赖关系的循环，用户经过数据依赖分析后，确认其不存在循环间数据依赖，可加上 #pragma ivdep 提示编译器忽略存在的数据依赖关系，示例程序片段如下：

```
1  void foo(float* a, int k)
2  {
3      …
4  #pragma ivdep
5      for(int j=0; j<1000; j++)
6      {
7          a[j] = a[j] + a[j+k];
8      }
9      …
10 }
```

当向量化以上代码段中的循环时，编译器会认为该循环存在循环间依赖，即第 j 次循环依赖第 j+k 次循环的结果，这种依赖关系称为交叉迭代依赖。而如果我们确定 k>16，超过 MIC VPU 的向量处理宽度，循环间就不存在实际意义上的数据依赖，加上 #pragma ivdep 指示编译器忽略数据的依赖关系并尝试进行向量化，甚至可以使用 #pragma simd 强制向量化该循环。如果 L<k<16，那么可以使用 #pragma simd vectorlength(L) 强制向量化该循环，并且能保证结果的正确性。

[0038] 9) 算法向量化改造

如果采用以上两种向量化方式，还有部分循环无法实现向量化，可以在数据依赖分析的基础上，对算法进行深度优化改进。主要的向量化改造方法有：

a) 调整嵌套循环的顺序

自动向量化只能对嵌套中的最内层的循环进行向量化，然而内层循环向量化效果未必最好，我们可以通过调整嵌套循环的顺序达到更好的向量化效果，如调整之后的向量化可以实现更好的连续访问，如下面的代码所示，B 代码的向量化效果比 A 的好。

[0039] b)

A	B

for(j=0; j<N; j++)
#pragma ivdep
for(i=0; i<M; i++)
{
C[i][j] = A[i][j]+B[i][j];
}
	for(i=0; i<M; i++)
	#pragma ivdep
	for(j=0; j<N; j++)
	{
	C[i][j] = A[i][j]+B[i][j];
	}

c) 拆分循环

在某些情况下，除了最内层的循环比较耗时外，其它不在最内层循环的代码也比较耗时，而这部分代码是无法自动向量化的，为此，我们可以采取拆分循环的方法实现更多的自动向量化，下面通过一段伪代码说明其使用方法。

```
1  for(i=0; i<N; i++)
2  {
3      rand();
4      …
5      …
6  }
```

[0040] 假设上面的代码中循环无数据依赖，由于 rand 函数无法向量化，从而导致整个循环无法向量化，我们可以把一个循环拆成两个循环的方法达到第二个 for 循环（主要耗时的）实现向量化的目的，代码如下：

```
1  for(i=0; i<N; i++)
2  {
3      rand();
4  }
5  for(i=0; i<N; i++) //自动向量化
6  {
7      …
8      …
9  }
```

另一个例子：

```
1  float s;
2  for(i=0; i<N; i++)
3  {
4      …
5      s=…;
6      for(j=0; j<M; j++)//自动向量化
7      {
8          if(s>0)
9          {
10             …
11         }
12     }
13 }
```

假设上面的代码中两层循环均无数据依赖，除了内层循环 for(j=0; j<M; j++) 比较耗时，对于 s 的求解也很耗时，然而求解 s 的部分是无法自动向量化的。我们可以通过拆分外层的循环做到更好地自动向量化效果，修改后的伪代码如下：

```
1   float s[16];
2   for(i=0; i<N; i+=16)
3   {
4       T=min(N-1,16);
5       for(k=0; k<T; k++) //自动向量化
6       {
7           ...
8           s[k]=...;
9       }
10      for(k=0; k<T; k++)
11      {
12          for(j=0; j<M; j++) //自动向量化
13          {
14              if(s[k]>0)
15              {
16                  ...
17              }
18          }
19      }
20  }
```

} 通过对循环的拆分，我们可以使更多的代码自动向量化，可以获取更好的向量化性能。

[0041] d) 手写 SIMD 指令向量化

向量化的层次如下图所示，越往上的级别，使用的语言越低级，编程越复杂，但控制的灵活性越好，理论上性能也越高。相反的，越往下的级别，编程越容易，但性能可能不是最理想。

[0042] 第一代 Intel MIC 产品为 KNC(Knights Corner)，Knights Corner Instructions 是 KNC 支持的 SIMD 指令的总称。可以看作是类似于 SSE、AVX 等的指令集。通过使用 Knights Corner 指令，可以细粒度地控制向量化运算。

[0043] Knights Corner Instructions 分类：

(3) Knights Corner 指令(Knights Corner instruction)是指具体的 SIMD 指令，是汇编指令集中关于 SIMD 的子集。

[0044] (4)内建 Knights Corner(Intrinsics of Knights Corner)是对 Knights Corner 指令的封装(几乎涉及到所有指令)，可以认为这些函数和数据类型是 C/C++ 的内建类型。

[0045] (5) Knights Corner 类库(Knights Corner Class Libraries)是为了方便使用 Knights Corner 指令而做的封装，可以让程序员尽量简单地使用 SIMD 指令，介于引语方式和 SIMD 代码之间。其支持整型和浮点型数据。

[0046] 下面通过单精度浮点向量加的例子说明三者的区别：

Knights Corner 指令	内建 Knights Corner	Knights Corner 类库
__m512 a, b, c; __asm{ vloadd v0, b vloadd v1, c vaddps v0, v1 vstored a, v0 }	#include <immintrin.h> ... __M512 a, b, c; a = _mm512_add_ps(b, c); ...	#include <micvec.h> ... F32vec16 a, b, c; a = b + c; ...

通过上面的例子可以看出使用类库的方式非常简单，能够以最类似于标量的方式(把数组看成变量)，进行向量化改造。而直接使用内建 Knights Corner 则更接近常规的思维方式，将两个数组通过向量化函数进行运算，当然，其代码要比使用类库方式复杂一些，但由于减少了封装和调用，因此性能也会略有提高。而内联汇编则是最难阅读的，由于最贴近底层，因而执行效率也最高，只是编程的成本也是最高的。在实际的 SIMD 指令编写中，我们一般采用内建 Knights Corner 的方式。

[0047] 下面我们通过一个向量加的示例说明 SIMD 指令的使用方法。

[0048]

```
#include <immintrin.h>
void foo(float *A, float *B, float *C, int N)
{
#ifdef __MIC__
        __M512 _A, _B, _C;
        for(int i=0; i<N; i+=16)
        {
                _A = _mm512_loadunpacklo_ps (_A, (void*)(&A[i]) );
                _A = _mm512_loadunpackhi_ps (_A, (void*)(&A[i +16]) );
                _B = _mm512_loadunpacklo_ps (_B, (void*)(&B[i]) );
                _B = _mm512_loadunpackhi_ps (_B, (void*)(&B[i +16]) );
                _C = _mm512_add_ps(_A, _B);
                _mm512_packstorelo_ps ((void*)(&C[i]) , _C );
                _mm512_packstorehi_ps ((void*)(&C[i +16]), _C );
        }
#endif
}
```

SIMD 指令与汇编指令类似，可读性较差，并且严重依赖于硬件，可移植性差。因此，SIMD 指令一般选择性使用，如代码量较少，计算却十分密集的地方。

[0049] 10) 向量化正确性验证。

[0050] 编译源代码，然后运行程序，检查程序的输出结果，验证向量化的正确性。

[0051] 向量化可能会带来精度损失，必要时可以通过编译器的 -fp-model 选项，调整向量化的精度。

[0052] 11) 迭代调优。

[0053] 重复以上过程，以实现循环向量化率最大化，从而使 MIC 处理器 VPU 的计算性能尽可能发挥出来。

[0054] 4、性能测试及分析

将该方法应用于一个典型的高性能运算案例——矩阵乘法。

[0055] 1）测试环境

平台	InspurNF5280M3
CPU	IntelXeonCPUE56753.07GHz，双路 8 核
Memory	DDR31333MHz128GB
MIC	KNC，60 核，1.0GHz，GDDR58GBmemory5.5GT/s
OS	RedHatEnterpriseLinuxServerrelease6.1，64bit
编译器	icc
测试用例	4096*4096 矩阵乘法

2）性能测试结果

程序版本	版本说明	时间（s）
P_baseline	CPU 单线程基准版	312.83
P_OMP	CPU 多线程 + 自动向量化版	170.83
P_MIC_base	MIC 多线程 + 自动向量化版	174
P_baseline_vec	CPU 单线程 + 算法向量化改造 + 向量化指示版	25
P_OMP_vec	CPU 多线程 + 算法向量化改造 + 向量化指示版	4.53
P_MIC_vec	MIC 多线程 + 算法向量化改造 + 向量化指示版	3.43
P_MIC_simd	MIC 多线程 + 算法向量化改造 + 向量化指示 +simd 指令版	2

3）性能测试结果分析

利用该方法对矩阵乘法应用案例进行向量化改造后，显著地提升了该模块的运行效率，获得了较高的性能加速比。

[0056] 5、总结

由本发明的技术方案可见，本发明提供了一种基于 MIC 架构处理器的向量化优化方法，该方法广泛适用于 MIC 架构处理器并行处理的应用场合，指导软件开发人员以较短的开发周期，较低的开发成本，快速高效地对现有软件进行向量化优化改造，实现软件对系统资源利用最优化，显著提高硬件资源利用率和软件的计算效率，从而大大提升软件整体性能。

[0057] 除说明书所述的技术特征外，均为本专业技术人员的已知技术。

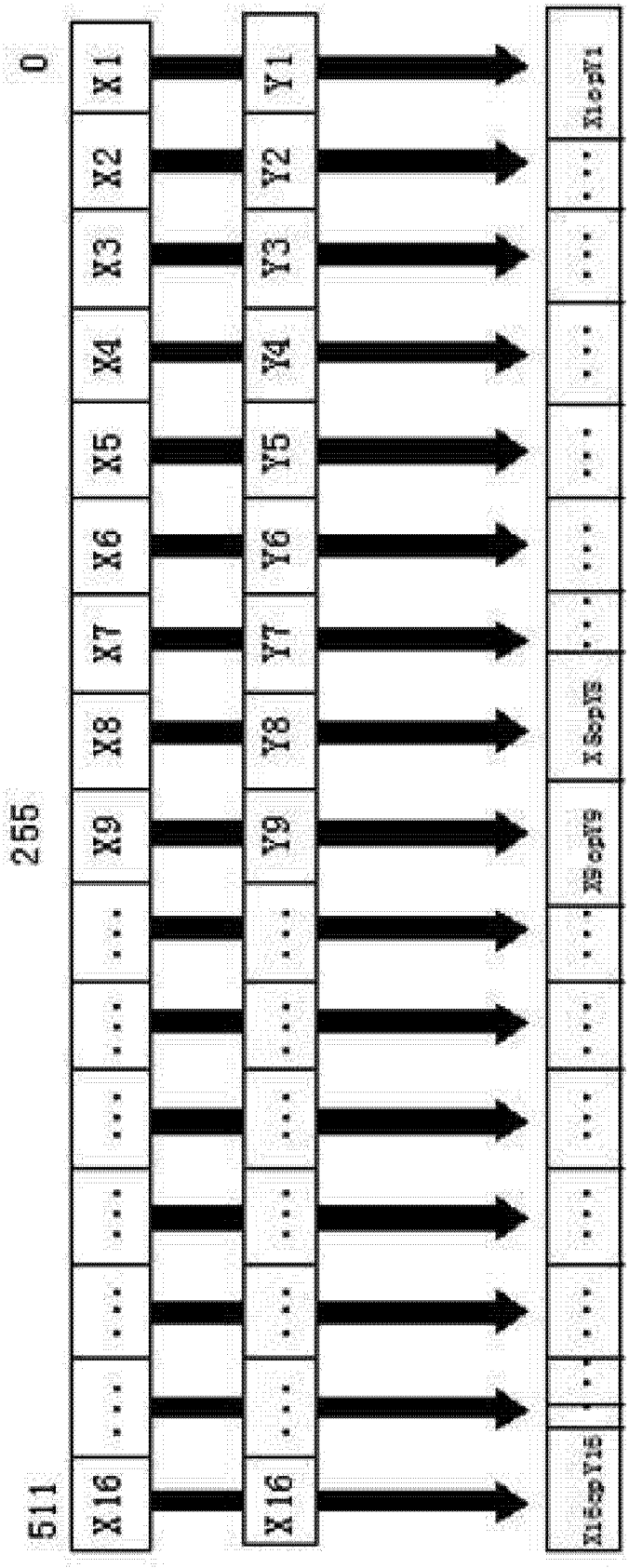


图 1

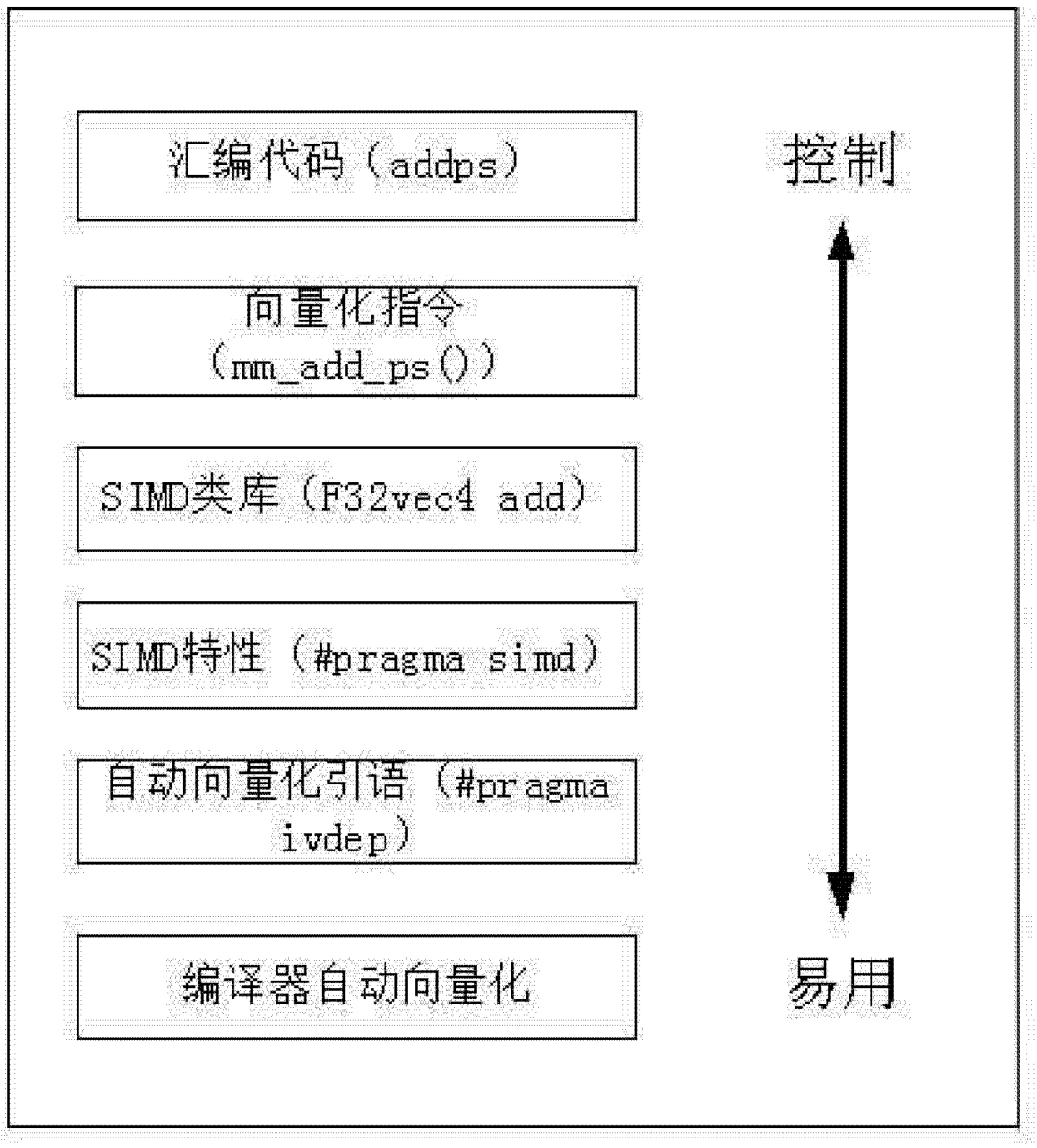

汇编代码（addps）
向量化指令
（mm_add_ps()）
SIMD类库（F32vec4 add）
SIMD特性（#pragma simd）
自动向量化引语（#pragma ivdep）
编译器自动向量化
控制
易用

图 2