GCC编译器中编译指导的自动向量化实现

徐 颖 李春江 董钰山 周思齐
(国防科学技术大学计算机学院计算机研究所 长沙 410073)

摘 要 基于编译指导的自动向量化已经成为编译器开发SIMD体系结构性能潜力的必然选择。OpenMP 4.0规范新增了SIMD编译指导语句,在开发中的GCC 4.9版本已经开始着手支持OpenMP4.0规范。详细分析了SIMD编译指导在GCC 4.9中的实现情况,重点分析了SIMD编译指导在编译器自动向量化阶段的影响,这为改进GCC的现有实现和提高向量化能力提供了有价值的参考。

关键词 GCC,SIMD,编译指导,自动向量化
中图法分类号 TP314 **文献标识码** A

Implementation of Auto-vectorization Based on Directives in GCC

XU Ying LI Chun-jiang DONG Yu-shan ZHOU Si-qi
(Institute of Computer, School of Computer Science, National University of Defense Technology, Changsha 410073, China)

Abstract Auto-vectorization based on directives has become an inevitable choice for compilers to exploit performance on SIMD architecture. The latest OpenMP 4.0 specification has added some SIMD directives, and the GCC 4.9 in development began supporting OpenMP 4.0. We analyzed the implementation of SIMD directives in GCC 4.9 in detail, and focused on how the SIMD directives affect loop vectorization. Our work provides valued references to improve the existing auto-vectorization.

Keywords GCC, SIMD, Directives, Auto-vectorization

1 引言

几乎所有主流处理器芯片都实现了SIMD扩展,需要在编程和编译器实现两个方面付诸努力来开发SIMD部件提供的短向量处理能力。通过程序员在程序中手工加入指导语句,指导编译器实现自动向量化是近年来非常重要的研究方向。OpenCL[1]在程序语言中设计了一种用来表示向量的基本数据类型,用户可以指定对这种向量数据的操作,这些操作会被编译器识别并转换为利用处理器SIMD指令的向量化代码。信息工程大学姚远等人[2]在2011年提出在代码中插入向量化编译指示语句,来指导自动编译器的自动向量化。2012年Intel公司的研究人员基于C和C++语言设计了SIMD扩展[3],使得编译器在转换过程中会将整个函数向量化,编译生成的SIMD指令序列能更充分地发挥处理器中SIMD单元的性能。

2012年,巴塞罗那超算中心的Michael Klemm等人通过在OpenMP并行程序中扩展SIMD编译指导优化了编译器的自动向量化[4],提出了向量化串行循环的SIMD结构,以及将循环向量化后分到绑定线程组上执行的复合SIMD结构中。国防科技大学黄娟娟等人[5]在2013年针对OpenMP中的PARALLEL_FOR结构设计并实现了aligned从句,根据从句属性设置程序的数据对齐属性,通过指导自动向量化代价模型的评估来实现自动向量化优化。

2013年7月发布的OpenMP 4.0规范设计了SIMD编译指导[6],目标是实现循环的多迭代并行执行的同时使用SIMD指令;以及对SIMD循环中调用的函数创建多版本,以便在simd分道(simd lanes,执行SIMD指令操作的部件)之间调用。OpenMP 4.0发布后不久,GNU发布GCC 4.9的开发版[7]已经开始着手实现OpenMP 4.0中新增的SIMD编译指导。本文分析了GCC 4.9中SIMD编译指导及其从句的具体实现,这对优化编译器的自动向量化实现、设计实现更多的编译指导有非常重要的意义。

2 GCC的编译架构

GNU编译器集合(GCC)是当前广泛使用的开源编译器,支持多种语言及多个平台。GCC4.0开始引入Tree-SSA优化框架,支持面向SIMD的自动向量化[8]。图1为GCC打开-O3优化选项后的编译过程。前端解析源程序,并将其转换成GENERIC表示;之后编译器以函数为单元生成所有函数的GIMPLE树和SSA树;然后进入优化遍阶段,直至目标

本文受国家自然科学基金项目:多核多线程处理器SIMD扩展的编程模型及编译优化关键技术研究(61170046),863计划项目:面向国产飞腾处理器的并行程序综合优化系统(2012AA010903)资助。

徐 颖(1992-),女,硕士生,主要研究方向为编译及优化技术,E-mail:xytvxq@gmail.com;李春江(1974-),男,博士,研究员,硕士生导师,CCF高级会员,主要研究方向为计算机体系结构、编译及优化技术;董钰山(1990-),男,硕士生,主要研究方向为编译及优化技术;周思齐(1993-),女,主要研究方向为编译及优化技术。

代码生成。Tree-SSA 优化框架即为从 GENERIC 表示到目标代码生成的过程。

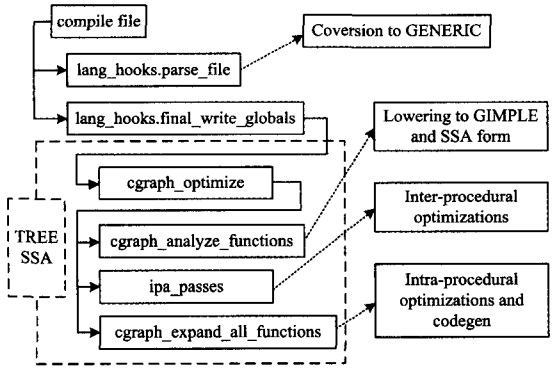

图 1　GCC 的编译过程

在具体实现时，Tree-SSA 框架通过“遍”完成所有优化工作。优化工作包括向量化、传统标量优化、死代码删除、向前传播等。编译遍的流程如图 2 所示[9]。

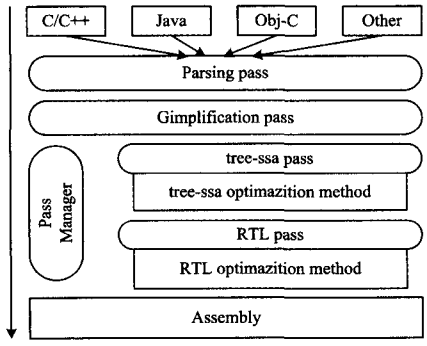

图 2　遍的编译流程

3　GOMP 的实现

OpenMP 是被广泛接受的面向共享主存多处理机系统进行并行程序设计的应用编程接口（API）标准。自 1997 年问世以来，OpenMP 被越来越多的编译器支持[10]。最新的 OpenMP 4.0 规范已于 2013 年 7 月发布。从 GCC 4.2 起，GCC 就支持 OpenMP。GOMP（GNU OpenMP）是 OpenMP 在 GCC 中的实现。它包含 4 个主要模块：前端、中间表示、代码生成和运行时库。

在 GCC 的 Tree-SSA 优化框架中，遍 pass_lower_omp 和遍 pass_expand_omp 用来处理 OpenMP 程序的编译实现。

3.1　pass_lower_omp

OpenMP 下降遍（pass_lower_omp）用来将 OpenMP 结构下降为 GIMPLE 中间表示，分为扫描阶段（scan_omp）和下降（lower_omp）阶段。

扫描阶段会递归遍历 OpenMP 程序语法树中的每个声明，对其中的表达式执行处理。若扫描到数据属性从句，则对并行区变量进行本地定义，对共享参数定义新结构以便之后的传递，并将变量间的映射关系存储到数据结构中。

下降阶段对不同的 OpenMP 语句分开处理。根据变量的映射值执行语句替换，并建立变量与其替换值的映射关系；插入参数传递语句及用来标识编译指导区域的 OMP_RETURN 语句。下降过程还会创建两个变量：.omp_data_o 和 .omp_data_i。前者保存所有发送给子线程的共享变量的值和地址；后者为子线程的输入参数，记录前者的地址。

3.2　pass_expand_omp

OpenMP 扩展遍以基本块为单位工作，根据已创建好的控制流图（CFG）构造一棵 OpenMP 区域树（omp region）。再按照该区域树间的层次关系递归调用各语法的相关处理语句，把每个单入单出的 OpenMP 并行区列入一个新的函数，并将所有其他的指导命令扩展为对 libgomp 库函数的调用或对应的 GIMPLE 扩展。

4　GCC 中基于循环的自动向量化

面向 SIMD 指令集的编译器自动向量化已成为产品化编译器实现的重要内容之一。当前编译器内实现自动向量化的方式主要有两种：基于循环的自动向量化和基于基本块的自动向量化。

目前，针对典型的 SIMD 体系结构，GCC 的自动向量化能力与商用编译器还有较大的差距[11,12]。因此，结合对 OpenMP 规范中 SIMD 编译指导的支持，在 GCC 中实现编译指导的自动向量化有非常重要的现实意义。

这里，首先简述下 GCC 中现有的关于循环的自动向量化实现。在循环向量化阶段，主要完成如下工作：

1）循环分析：测试循环能否被向量化。具体工作是检查循环的 CFG 特性，以及分析循环中的数据引用、对齐等多种情况，如果不满足其中任何一个条件，就不能执行向量化。循环分析阶段的信息被记录在 3 种数据结构中，即循环级别的 loop_vect_info、语句级别的 stmt_vec_info 以及访存级别的 data_reference。

2）循环变换：若循环可被向量化，那么自上而下扫描循环的每一条语句，为需要执行向量化的标量语句创建其对应的向量化语句，并插入到该标量语句之前。所有向量语句插入后，删除标量语句。store 操作在该阶段被显式删除，而其他标量语句会在死代码消除遍（pass_dse）中被删除。

5　GCC 中 SIMD 编译指导的实现

5.1　OpenMP 4.0 规范中 simd 编译指导

OpenMP 4.0 规范中新增内容之一是 SIMD 结构，涉及的 3 条编译指导语句及其从句定义如下。

1. simd 结构如图 3 所示。

```
#pragma omp simd [clause[[,] clause] …] new-line
    for-loops
```

图 3　simd 结构语法

其中从句为下列之一：

safelen(length)

linear(list[:linear-step])

aligned(list[:alignment])

private(list)

lastprivate(list)

reduction(reduction-identifier:list)

collapse(n)

simd 指导若作用到循环上，表明该循环的多个迭代可用 SIMD 指令并发执行。当前 simd 编译指导只能应用在 for 循环上。

2. declare simd 结构如图 4 所示。

```
#pragma omp declare simd[clause[[,] clause] …] new-line
[#pragma omp declare simd[clause[[,] clause] …] new-line]
[…]
function definition or declaration
```

图 4　declare simd 结构语法

从句为下列之一：

simdlen(length)

linear(argument-list[:constant-linear-step])

aligned(argument-list[:alignment])

uniform(argument-list)

inbranch

notinbranch

declare simd 结构应用在 SIMD 循环调用的函数或子例程上，用来创建函数或子例程 SIMD 版本，即可使用 SIMD 指令同时处理这些版本的参数。由于该编译指导是声明性的，因此多条 declare simd 编译指导可同时作用在同一函数或子例程上。

3. loop SIMD 结构如图 5 所示。

```
#pragma omp for simd [clause[[,] clause] …] new-line
    for-loops
```

图 5　loop SIMD 结构语法

loop SIMD 结构会先将循环的迭代分配到并行区的隐式任务上，分到各任务上的迭代块会根据从句转换成对应的 SIMD 循环。

OpenMP 关于 SIMD 的编译指导在当前 GCC 4.9 中的实现，主要涉及以下几个方面：

- 编译指导在前端的识别；
- GOMP 中的下降和扩展；
- GOMP 中，新增了专门用来处理 declare simd 函数的新遍 pass_omp_simd_clone；
- 在自动向量化阶段根据指导命令实现自动向量化。

下面分别进行介绍。

5.2　simd 编译指导的识别

新增 simd 编译指导后，在 c-pragma.h 文件内的数据结构中增添了 simd 编译指导的标识。如在 *typedef enum* pragma_kind 中增加编译指导的宏“PRAGMA_OMP_SIMD”；在 *typedef enum* pragma_omp_clause 中增加从句标识，如：PRAGMA_OMP_CLAUSE_COLLAPSE、PRAGMA_OMP_CLAUSE_LINEAR、PRAGMA_OMP_CLAUSE_SIMDLEN。

C 语言的前端分析程序在 c-parser.c 中。所有以 #pragma 开头的语句都在函数 c_parser_pragma() 中被解析，其中 c_parser_omp_construct() 解析所有的 OpenMP 编译指导。不同的 OpenMP 编译指导都有对应的函数分别解析。simd 的 3 种结构的解析情况如下：

- simd 结构：被前端识别后，调用解析函数 c_parser_omp_simd()。该函数中，调用函数 c_parser_omp_all_clauses()解析所有从句，之后又调用 c_parser_omp_for_loop()解析 for 循环。所有解析出来的信息放在 tree 类型的节点中。
- loop SIMD 结构：它在被解析时，由于先识别了 #pragma omp for，因此会进入函数 c_parser_omp_for()。在该函数中若发现编译指导后面还有 simd 指导命令，则调用 c_parser_omp_simd()。
- declare simd 结构：由于这是声明性的编译指导，因此是在解析函数声明和定义的函数 c_parser_declaration_or_fndef()中调用函数 c_finish_omp_declare_simd()来解析，解析出的信息放在全局变量 current_function_decl 中。该变量定义在 GCC 顶层文件 toplev.c 中，是 tree 类型的节点，表示当前正在被编译的函数。

5.3　simd 编译指导的下降和扩展

在 OpenMP 下降和扩展阶段，只涉及 simd 和 loop SIMD 两种结构，declare simd 结构的处理是在新增遍 pass_omp_simd_clone 中实现。simd 这两种结构的处理和其他 OpenMP 编译指导的处理基本相同，但增加了一些特殊的处理。

在下降遍 pass_omp_lower 中，函数 lower_rec_input_clauses()用来将从句下降为 gimple 代码，而它辅助函数 lower_rec_simd_input_clauses()负责 simd 编译指导的私有化。具体说来，函数里调整了最大向量化因子(max_vf)，并构造给定类型的数组，在后续遍 pass_omp_simd_clone 的执行过程中创建 simd array 时会用到。

在扩展遍 pass_omp_expand 中，函数 expand_omp_simd()专门给 simd 的非工作共享循环生成代码，它是函数 expand_omp_for()的子例程。它按照分支情况将 for 循环转换成具有多个分割块的伪代码。函数中还专门说明，如果没有-fno-tree-loop-vectorize 选项，则对当前循环的强制向量化的属性赋值为真：

loop->force_vect=true;

同时对当前函数 cfun 中存在必须要向量化的循环的属性赋值：

cfun->has_force_vect_loops=true;

cfun 为 *struct* function 类型的指针，定义在 function.c 中，表示当前被编译的函数，它的信息内容会在各编译阶段间传递使用。

5.4　pass_omp_simd_clone

该遍为 GCC 4.9 的新增遍，同样属于 GOMP 处理过程，运行在 OpenMP 下降遍和 OpenMP 扩展遍之后，仍然实现在 omp-low.c 中。它的目的是给添加了 declare simd 结构的函数(简称为 SIMD 函数)创建多个 SIMD 版本，这些版本被称为 simd clones。

该遍的入口函数为 expand_simd_clones()。使用到的数据结构主要有 struct cgraph_node 和 struct cgraph_simd_clone。前者是函数调用图 cgraph 的节点，每个函数的声明都有个被赋值的 cgraph_node，在图中和它连线的其他 cgraph_node 为该函数的调用函数或被调用函数；后者是保存 SIMD 函数特定信息的结构。

该入口函数主要进行两步操作：一是确认函数带有“omp declare simd”的编译指导，否则函数终止；二是使用一个 do-while 循环，遍历所有的 SIMD 函数，而处理每个 SIMD 函数的主要步骤如下：

1)先从当前函数中提取出 SIMD 函数的特定信息放在 struct cgraph_simd_clone 类型的变量 clone_info 中；

2)确定目标平台上有多少种 ISA(指令集)需要创建 simd clone。若无 ISA 需要创建 simd clone，则跳过该次循环并检测下一个 SIMD 函数；

3)若有 ISA 需要创建 simd clone，则以最初的 clone_info

作为基础，复制出不同 ISA 所需要的 simd clones，且当函数中存在 inbranch 从句时，会分别为 inbranch 和！inbranch 各创建一个 clone。所以如果有 n 个 ISA 需要 clone，那么最多会得到 $2*n$ 个函数副本；

4）每新建一个 simd clone，就创建一个 cgraph_node，将新建好的 simd clone 赋给 cgraph_node 的 simdclone 域；

5）将所有新建的 cgraph_node 串起来。

创建完成的 simd clones 会在自动向量化阶段被使用。

在该遍的实现中，还创建了一个重要结构——simd array。在 SIMD 函数中的参数在最后都会被转换成向量。因此，每个标量参数都应该有一个对应的向量，即 simd array。它的大小是 simd lane 的个数，元素类型即为标量参数对应的类型。函数 create_tmp_simd_array()用于创建 simd array，函数 ipa_simd_modify_function_body()用于将函数的所有参数用 simd array 替换。

5.5 自动向量化阶段对 simd 编译指导的处理

在目前的 GCC 4.9 自动向量化阶段，simd 结构和 loop SIMD 结构的实现基本相同，因此在下面的介绍中将这两种结构作为同一种情况。

5.5.1 declare simd 结构在自动向量化中的实现

遍 pass_omp_simd_clone 就是专门针对声明为 declare simd 的函数创建所需要的一个或多个 simd clones。当生成 simd clones 后，在向量化阶段，若分析到循环中确实有调用 SIMD 函数的语句，则会寻找该函数最合适的 simd clone，若该 simd clone 能被向量化，就将其转换成对应的向量化代码。该过程在 vectorizable_simd_clone_call()中实现。

在实现过程中，遍历传递过来的所有参数，为每个参数寻找最合适的 simd clone，该 clone 的对齐量与参数自带对齐量有最小的差距，差距量用变量 this_badness 表示，图 6 为 this_badness 的定义。使用到的辅助数据结构为 *struct* simd_call_arg_info，用它来记录调用语句参数的 simd 信息。

```
this_badness += (exact_log2 (arginfo[i].align) -
    exact_log2 (n->simdclone->args[i].alignment));
```

图 6 this_badness 的定义

通过循环遍历寻找拥有最小 this_badness 的 simd clone，并找到该 *clone* 对应的 cgraph 节点 bestn。图 7 为通过 bestn 给一些变量赋值：

```
fndecl = bestn->decl;
nunits = bestn->simdclone->simdlen;
ncopies = LOOP_VINFO_VECT_FACTOR
      (loop_vinfo) / nunits;
```

图 7 给变量赋值

图 7 中，当前函数的有效声明 fndecl 被赋值为 bestn 的声明；nunits 为该函数可并发执行参数的个数；ncopies 为该标量语句转换为向量语句时向量语句的数目。这就完成了寻找最合适的 simd clone 并将其属性赋给当前函数的过程。

5.5.2 simd 结构与 loop SIMD 结构在自动向量化中的实现

这两种编译指导都只能应用于紧嵌套 for 循环，这些 for 循环被称为 SIMD 循环。

1. SIMD 循环的识别

为标识 SIMD 循环，分别在 *struct* loop 和 *struct* function 中新增域：

1）*tree* simduid：*struct* loop 新增域，SIMD 循环的唯一标识；

2）*unsigned int* has_simduid_loops：*struct* function 的新增域，若该函数包含的循环的 loop->simduid 不为 0，则该域置为非 0。

只有当 simduid 不为 NULL，has_simduid_loops 不为 0 时，才能确定该循环为 SIMD 循环。这两个域在 OpenMP 扩展遍的函数 expand_omp_simd()中被赋值（见图 8）。

```
tree simduid = find_omp_clause (
      gimple_omp_for_clauses (fd->for_stmt),
            OMP_CLAUSE__SIMDUID_);
.....
.....
if (simduid)
{
    loop->simduid =OMP_CLAUSE__SIMDUID__DECL
                  (simduid);
    cfun->has_simduid_loops = true;
}
```

图 8 simduid 及 has_simduid_loops 的赋值

如图 8 所示，has_simduid_loops 域为 true 的信息被放入全局变量 cfun 中。由 cfun 将 SIMD 循环的指定信息传递到自动向量化阶段。

值得注意的是，图 8 中的 loop->simduid 和 cfun->has_simduid_loops 能被赋值的条件是树节点 simduid 不为空，即函数 find_omp_clause()能找到 OMP_CLAUSE_SIMDUID_这个从句，而该从句是在 OpenMP 下降遍的函数 lower_rec_input_clauses()中被创建的，创建该从句的条件是编译指导中必须带 private、firstprivate 或 reduction 从句。从现有的实现看，要在向量化阶段识别 SIMD 循环，编译指导必须带上以上 3 种从句之一。

在循环向量化入口函数 vectorize_loops()中，有如图 9 所示代码。

```
if (cfun->has_simduid_loops)
  adjust_simduid_builtins (simduid_to_vf_htab);
.....
.....
if (cfun->has_simduid_loops)
  note_simd_array_uses (&simd_array_to_simduid_htab);
.....
.....
```

图 9 识别 SIMD 循环

这说明自动向量化阶段的函数已经具有识别 SIMD 循环的能力。

2. SIMD 循环有关的数据结构

下面介绍在自动向量化阶段会被用到的与处理 SIMD 循环相关的数据结构：

1）*struct* simduid_to_vf：simduid 到向量化因子的映射；

2）*struct* simd_array_to_simduid：OMP simd array 到对应 simduid 的映射，simduid 转换成 OMP simd array 的索引；

3）IFN_GOMP_SIMD_LANE、IFN_GOMP_SIMD_VF、IFN_GOMP_SIMD_LAST_LANE，这是 3 个内部函数（inter-

（下转第 392 页）

Virtual File System for Linux Clusters[C]// the 4th Annual Linux Showcase and Conference. 2000:317-327
[41] Sun Microsystems, Inc.. Lustre™ 1.6 Operations Manual[EB/OL]. [2014-03-19]. http://wiki.lustre.org/images/alae/820-3681.pdf
[42] 邵佩英. 分布式数据库系统及其应用[M]. 北京:科学出版社,2005,7
[43] 李川. 分布式数据库查询策略优化的研究[D]. 西安:西安电子科技大学,2012
[44] 萨师煊,王珊. 数据库系统概论(第三版)[M]. 北京:高等教育出版社,2000
[45] Vaquero L, Rodero-Marino L, Caceres J, et al. A break in the clouds: towards a cloud definition[J]. SIGCOMM Computer Communication Review, 2009, 39(1): 50-55
[46] 罗军舟,金嘉晖,宋爱波. 云计算-体系架构与关键技术[J]. 通信学报,2011,32(7)

(上接第 367 页)

nal functions),内部函数和内嵌函数的区别在于前者没有链接,所以只能被 GCC 内部调用,而不能被用户手工调用。

3. SIMD 循环的向量化

SIMD 循环在向量化阶段的处理主要涉及如下 3 个函数:

1)入口函数 vectorize_loops()

该函数中对循环的向量化可能性进行分析前,先判断 cfun 所带循环数目。若数目不超过 1,函数将返回,不执行向量化。但在返回之前,若发现 cfun 有 SIMD 循环,则调用 adjust_simduid_builtins(),该函数为 IFN_GOMP_SIMD_VF, IFN_GOMP_SIMD_LANE 和 IFN_GOMP_SIMD_LAST_LANE,3 个内部调用函数分别创建一个整型树节点,替换 cfun 中调用该内部函数的语句。

而若 cfun 的循环个数超过 1,则调用 note_simd_array_uses(),构建从 simd 数组到 simduid 的映射。

在分析循环阶段,全部分析完后,执行代码的向量化转换。在循环已被向量化后,为了使得该循环能被顺利展开,新建 simduid_to_vf 类型的变量 simduid_to_vf_data,并给其成员变量赋值,如成员 simduid 的值来自已向量化循环的 simduid,成员 vf 的值来自已向量化循环的向量化因子。

在结束阶段,完成各数据结构的销毁后,仍调用 adjust_simduid_builtins()来转换内部调用函数语句,之后,对 simd 数组(simd array)遍历,对其中的每个元素取其 simduid,再到哈希表 simduid_to_vf_htab 中去匹配,找到对应元素并取出 vf 值赋给最终的向量化因子。

2)函数 vect_analyze_data_refs()

若存在 SIMD 循环,则可确定存在支持 simd lane 的访问的可能性,然后还需要对数据引用进行相关性分析,确定 simd lane 是否能真的访问。若不能访问,则以向量化分析失败返回。

3)函数 vectorizable_call()

该函数用来向量化调用语句(即调用其他函数的语句)。分析当前语句 stmt 调用的函数发现其如果不能被向量化,则还会检查一种特殊情况,即若 stmt 调用的是 IFN_GOMP_SIMD_LANE 且 stmt 的循环是 SIMD 循环,则仍可向量化。在后面实现向量化的过程中,IFN_GOMP_SIMD_LANE 被处理成$\{0,1,2,\cdots,v_{f-1}\}$向量。

4)vect_estimate_min_profitable_iters()

该函数用于判断向量化收益,在向量收益小于标量收益时,不能进行向量化。此时,若有 SIMD 循环,仅输出信息以说明该 SIMD 循环不可向量化。

结束语 从以上分析可以看出,在当前编译器的实现中,simd 的 3 种结构都能被前端识别,且在 OpenMP 下降和扩展及自动向量化阶段都有处理。GCC 4.9 的新增遍 pass_omp_simd_clone 专门为声明为 declare simd 的函数创建向量化版本。在已有的如 function、loop 等数据结构中新增域来记录 simd 编译指导的相关信息,还会临时创建 simduid_to_vf 等数据结构辅助相关功能的实现。在编译各阶段间,全局变量 cfun 作为主要数据结构来传递 simd 编译指导的信息。但是,当前实现中仍存在较多不足,主要表现在以下两个方面:

1)SIMD 循环的识别受限。当前只能识别带 private、firstprivate 或 reduction 3 种从句的 SIMD 循环;

2)向量化阶段功能实现不完善。主要是指两种 simd 编译指导下的 SIMD 循环在向量化阶段,仅是将内部函数转换成整型树节点、构建一些映射、部分影响向量化条件等,并未真正实现直接指导向量化或转换 SIMD 循环。而且在循环转换成向量化代码的过程中,并未对 SIMD 函数或 SIMD 循环有任何特殊处理。

在下一步工作中,将针对其不完善的部分进行改进,实现用 SIMD 编译指导提升 GCC 自动向量化的能力。

参 考 文 献

[1] Khronos OpenCL Working Group. The OpenCL Speci? cation [R]. [2009]. http://www.khronos.org/registry/cl/
[2] 姚远,赵荣彩. 基于编译指示的向量化方法[J]. Computer Engineering, 2012, 38(12): 272—275
[3] Tian X, Saito H, Preis S V. Compiling C/C++SIMD Extensions for Function and Loop Vectorization on Multicore-SIMD Processors[C]// Multicore and GPU Programming Models, Languages and Compilers Workshop. 2012: 2349-2358
[4] Klemm M, et al. Extending OpenMP* with vector constructs for modern multicore SIMD architectures[C]// OpenMP in a Heterogeneous World, 2012: 59-72
[5] 黄娟娟,李春江,徐颖. GCC 中自动向量化代价模型剖析[C]// 第 17 届计算机工程与工艺年会暨第三届微处理器技术论坛论文集. 长沙:国防科技大学出版社,2013:259-268
[6] OpenMP Architecture Review Board: OpenMP Application Program Interface[M]. Version 4.0(July 2013)
[7] Free Software Foundation Inc. GCC 4.9 Release Series http://gcc.gnu.org/gcc-4.9/
[8] Novillo D. Design and Implementation of Tree SSA[C]// Proceedings of the 2004 GCC Summit. Ottawa, Canada, 2004
[9] 黄娟娟. 多线程多 SIMD 自动向量化技术研究[D]. 长沙:国防科学技术大学,2013
[10] 辛乃军,陈旭灿,等. 基于 GCC 的高性能 DSP Matrix 向量指令集扩展[J]. 计算机工程与科学,2012,34(1):58-63
[11] 徐颖. 编译器自动向量化效能评估与分析[D]. 长沙:国防科学技术大学,2012
[12] 李春江,黄娟娟,徐颖. 典型编译器自动向量化效果评估与分析[J]. 计算机科学,2013(4):41-46