“神威•太湖之光”计算机系统

OpenACC*用户手册

Version 1.0

国家并行计算机工程研究中心

2016 年 6 月

目录

1 简介

1.1概述

欢迎使用 OpenACC*语言及 SWACC 编译系统。本文主要介绍“神威•太湖之光”计算机系统中 SWACC 编译系统的使用以及其所支持的 OpenACC*语言的用法和说明。OpenACC*语言是在 OpenACC2.0 文本的基础上，针对国产申威 26010 众核处理器结构特点进行适当的精简和扩充而来的。OpenACC 是由 OpenACC 组织(www.openacc-standard.org)于 2011 年推出的众核加速编程语言，2013 年发布 2.0 文本。OpenACC 是以编译指示的方式提供众核编程所需的语言功能，其主要目的是降低众核编程的难度。

1.2范围

本文档所描述的编译指示、用户接口以及使用方式只针对加速编程，即将编译指示指定的程序代码段加载到从核加速核心上执行，程序的其他部分仍然在主核心上执行。本文档只介绍对加载到从核心代码的详细要求。对在主核心上编程的特征和限制，本文档不作说明。

本文档不讲述使用编译器或其他工具自动检测并将代码段加载到从核心运行的内容，也不介绍如何将加速代码分配到多个从核心上。

1.3术语

加速设备（Accelerator device）：CPU 的特殊协处理器，简称为 Device。CPU 可以将数据与计算核心下载到该特殊协处理器上执行，如 GPU、MIC、申威 26010 的运算核心阵列（简称从核阵列）等都可以称为加速设备。

主处理器（Host）：主处理器通常指通用处理器核心，如通用 CPU、申威 26010 的运算控制核心（简称主核）。在本文中特指连接有加速设备的主处理器。

主存（Host memory）：主处理器的内存空间。

设备内存（Device memory）：位于加速设备内的内存。加速设备访问设备内存通常比访问主存快，如 GPU 中的 shared memory、申威 26010 中的运算核心局存(LDM，Local Data Memory)。

主线程（Host thread）：在主处理器上运行的程序实体，简称为主进程、或主线程、或进程。

加速线程（Accelerator thread）：运行在加速设备处理器核心上的程序实体，简称为线程。

主程序代码（Host code）：运行在主处理器上的程序代码，简称主程序或主程序代码。在本文中所指的程序代码，除指定要在加速设备上执行的加速程序代码外，均为主程序代码。主程序代码控制加速程序代码的加载运行。

加速程序代码（Accelerator code）：运行在加速设备上的程序代码，也称为加速程序/代码。

加速任务（Kernel）：在加速设备上运行的代码和数据称为加速任务。在不同的加速设

备上，kernel 可以是简单的循环或复杂的子例程。

结构化代码块（Structured block）：结构化代码块是有单一入口和单一出口的复合语句；对于 C 程序来说，即为用一对大括号包含的代码段。

加速计算区（Accelerator compute region）：由加速计算区编译指示定义的一个代码区域。加速计算区是一个在加速设备上执行的结构化代码块。加速计算区通常要求在入口处向设备内存申请空间，并将数据从主存拷贝到设备内存；在出口处将数据从设备内存拷贝回主存并释放设备内存空间。加速计算区不可以包含其它加速计算区，但可以包含加速循环。在 OpenACC*语言中，parallel 指示标注的代码区域即为加速计算区，也是需要在加速设备上运行的加速程序代码(Accelerator code)，经过编译器的处理之后，加速计算区的程序代码将被封装成加速任务(Kernel)，最终加载到加速设备上运行。

加速数据区（Accelerator data region）：由加速数据区编译指示定义的一个区域。加速数据区通常要求在入口处向设备内存申请空间，并将数据从主存拷贝到设备内存；在出口处将数据从设备内存拷贝回主存并释放设备内存空间。加速数据区可以包含其它加速数据区和加速计算区。

加速循环（Accelerator loop）：由循环映射编译指示定义的一个循环体，位于加速计算区内。加速循环通常会以任务分担的方式被分配到多个加速线程上执行。

私有变量（Private variable）：指存放在线程私有主存空间内，仅本加速线程可以访问的变量。变量在私有化后，每个线程都会有与主线程变量相对应的副本，在并行运行时不会发生写冲突的情况。

本地变量（Local variable）：指存放在加速设备的设备内存中，仅本加速线程可以访问的变量，属于私有变量的一种特例。

1.4执行模型

OpenACC*程序的执行模型是在 host（主处理器）的指导下，host 和 device（加速设备）协作的加速执行模型，如图 1-1 所示。程序首先在 host 上启动运行，以一个主线程串行执行，或者通过使用 OpenMP 或 MPI 等编程接口以多个主线程并行执行，计算密集区域则在主线程的控制下作为 kernel（加速任务）被加载到加速设备上执行。kernel 的执行过程包括：在设备内存上分配所需数据空间；加载 kernel 代码（包含 kernel 参数）至 device；kernel 将所需数据从主存传输至设备内存；等待数据传输完成；device 进行计算并将结果传送回主存；释放设备上的数据空间等。大多数情况下，host 可以加载一系列 kernels，并在加速设备上逐个执行。

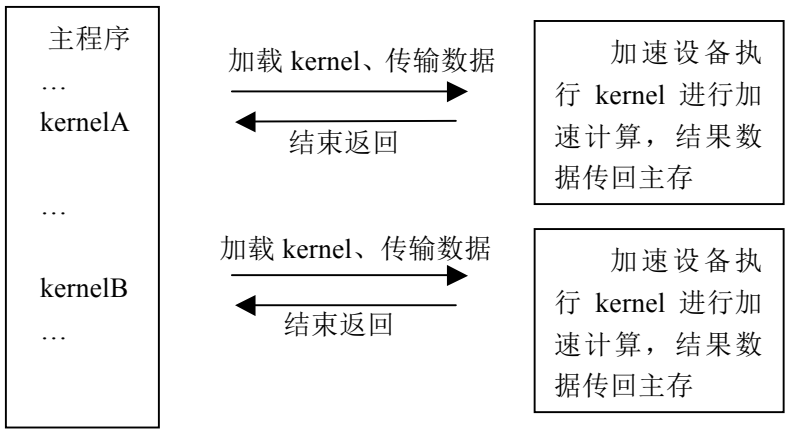

图 1-1 执行模型示意图

OpenACC*支持三级并行机制：gang、worker、vector。gang 是粗粒度并行，在加速设备上可以启动一定数量的 gang。worker 是细粒度并行，每个 gang 内包含有一定数量的 worker。vector 并行是在 worker 内通过 SIMD 或向量操作的指令级并行。gang、worker、vector 三级并行是包含关系，对于一层循环并行可以不用指定，对于嵌套循环并行则要遵循三者的嵌套关系，即 gang 级并行的循环内只允许出现 worker 或 vector 级并行的循环，worker 级并行的循环内只允许出现 vector 级并行的循环。

在申威 26010 中，一个运算控制核心（简称主核）仅控制一个运算核心阵列(加速设备，简称从核阵列)的运行，每个运算核心阵列内有 64 个运算核心（简称从核），每个运算核心可以运行一个加速线程。默认情况下，64 个加速线程被组织成 64 个 gang、每个 gang 内一个 worker、worker 内可以 vector 并行的逻辑视图。gang 的数量与 worker 大小的乘积即为实际运行的加速线程的数量，但是不可以超出 64。通常情况下，尽量用满 64 个加速线程可以获得较好的运行性能。

1.5存储模型

仅在 Host 上运行的程序和在 Host＋Device 上运行的程序，二者最大的区别在于存储层次和存储模型的不同。当前大多数 Host＋Device 平台，比如 CPU+GPU 平台，支持的存储模型为主存和设备内存分离的模型，device 不能直接访问主存，加速计算所需的数据必需通过 Host 进行传输。申威 26010 采用了片上融合异构众核体系结构，device 可直接访问主存空间，并在 device 内提供加速线程私有的高速缓冲(LDM, Local Data Memory)，加速计算需要存放到 LDM 的数据由 device 控制传输。本系统支持的存储模型如图 1-2 所示。

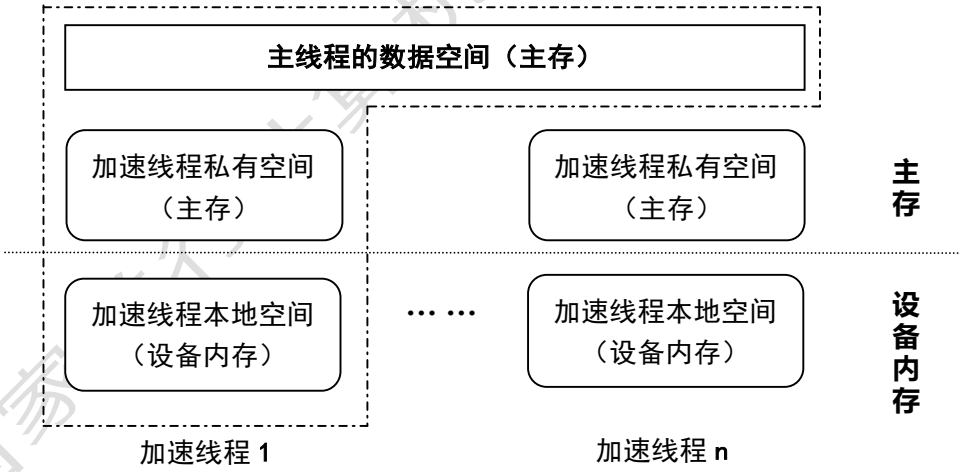

图 1-2 存储模型示意图

图 1-2 中虚线框内为一个加速线程可见的存储空间，包括三个部分：

1) **主线程数据空间**：位于主存，主线程的内存空间对其创建的加速线程直接可见，加速线程可以直接访问相应的数据；对于主线程创建的多个加速线程而言，这部分空间是共享的；程序中在加速区外定义的变量，均位于该空间内。
2) **加速线程私有空间**：位于主存，每个加速线程有独立的私有空间，程序中使用 private、firstprivate 子句修饰的变量将存放于该空间内。
3) **加速线程本地空间**：位于设备内存，每个加速线程有独立的本地空间(LDM)，本地空间的访问性能是三种空间中最高的。程序中使用 local、copy 等数据子句修饰的变量将由编译系统控制全部或局部存放于 LDM 内。主存与本地空间的数据交互由

加速线程控制。

使用 OpenACC*编写程序，数据的管理和传输是隐式的，由编译器根据编译指示的信息，自动生成空间管理、数据传输等控制代码，然而充分了解系统的存储模型和特点，对编写出高性能的众核应用十分重要，需要了解的一些加速编程的注意事项包括：

1) 一个高效的加速代码需要有较高的计算密度，需要多高的计算密度取决于主存和设备内存之间的访存带宽；
2) 设备空间的容量有限，编写程序时需要充分考虑设备空间的容量和加速代码的内存需求。

1.6兼容性

由于在申威 26010 上程序的执行方式和存储模型与 GPU、MIC 等平台存在一定的差异，主要体现在存储层次、数据传输控制方式上（参看 1.4 和 1.5），而 OpenACC2.0 文本中约定的存储模型主要针对 GPU 这类加速器的结构特点设定。因此本系统在支持 OpenACC2.0 方面有以下一些特点：

1) SWACC 编译系统支持 OpenACC2.0 文本全集，包括其中定义的编译指示、运行时库接口和环境变量，即按照 OpenACC2.0 文本规范，编写的基于 GPU 或其他平台的程序，可以直接使用 SWACC 编译系统进行编译并运行，保证其主要语言功能的含义和程序的正确性。
2) 存储模型的映射：OpenACC2.0 文本中主要针对存储模型是主存和设备内存分离的模型，加速设备不能访问主存；而申威 26010 支持的片上融合异构众核体系结构使得加速设备可以直接访问主存，按照 OpenACC2.0 文本的说明，在加速设备和主进程共享主存的平台上，OpenACC2.0 文本中在设备内存上申请空间和数据传输等操作在实现时不需要实际执行。在申威 26010 中设备可用的内存主要包括两部分的：主存和设备内存，在实际实现时将这两部分作为整体对应到 OpenACC2.0 文本中所定义的“设备内存”，由编译系统决定具体变量的存储位置。
3) 执行模型的映射：OpenACC2.0 文本定义的执行模型与申威 26010 的执行模型是基本一致的，因此其中的加速计算区等相关功能可直接对应到申威 26010 的执行部件。

在支持 OpenACC2.0 文本基础上，为了更好的利用 OpenACC2.0 未能在语言功能中体现、但对提升程序性能十分关键的申威 26010 的特征和用法，本系统对 OpenACC2.0 文本的功能进行了适当且必要的扩充、对一些语言功能的使用方式进行了扩展，本文档后续的内容不按照 OpenACC2.0 文本的格式进行说明，将主要介绍在申威 26010 上如何开发出高效的 OpenACC*加速应用所需要了解和掌握的语言功能和注意事项。

1.7文档概述

阅读本手册前，希望读者对 C、Fortran、OpenMP 语言比较熟练，对 UNIX/LINUX 操作系统进程和并行机制有一定的了解。本手册重点叙述 SWACC 编译系统和 OpenACC*语言的使用方法，本文档后续的主要内容包括：

2 SWACC 编译系统

3 OpenACC*编译指示

4 运行时库接口

5 使用及编程指南
6 版权说明

1.8引用文档

1) American National Standard Programming Language C, ANSI X3.15-1989(ANSI C).
2) ISO/IEC9889:1999, Information Technology – Programming Languages – C(C99).
3) ISO/IEC14882:1998, Information Technology – Programming Languages – C++.
4) ISO/IEC 1539:1980, Information Technology - Programming Languages – Fortran (Fortran77)
5) ISO/IEC 1539:1991, Information Technology - Programming Languages – Fortran (Fortran90).
6) ISO/IEC 1539-1:1997, Information Technology - Programming Languages - Fortran (Fortran95).
7) The OpenACC Application Programming Interface, version 2.0, 2013.06.

2 SWACC 编译系统

SWACC 编译系统是支持 OpenACC*语言的基础。使用 OpenACC*语言编写的程序需要使用 SWACC 编译系统进行编译，同时 SWACC 编译系统支持使用 OpenACC*与 MPI 等编程接口编写的混合程序的编译。

2.1支持的基础语言标准

SWACC 编译系统支持的基础语言标准包括：

1) C 语言；
2) C++语言(运算核心不支持 C++，即加速代码不支持 C++)；
3) Fortran 语言。

2.2编译命令及选项

SWACC 编译系统提供三种编译命令：*swacc*、*swaCC*、*swafort*，分别对应 C、C++、Fortran 程序的编译。以编译 C 程序为例，其编译命令的格式为：

swacc [选项] 文件名

其中，选项和文件名为输入参数，中括号表示为可选的参数，比如 swacc hello.c。

SWACC 编译系统支持的编译选项包括自有的编译选项和通用编译选项。通用编译选项为 cc、gcc 和 icc 中提供的常规编译选项，包括：-c -g -Olevel -o -Idir -Ldir -Dmacro[=defn]等。SWACC 编译系统主要的自有编译选项如表 2-1 所示：

表 2-1 SWACC 编译选项

编译选项	说明
-|--(h|help)	显示帮助

-SCFlags	指定的选项将被传递到编译 device 程序的串行编译器，若同时传递多个选项需使用逗号进行分割，中间不能有空格，例如：swafort -SCFlags -extend_source,-O3 hello.c
-HCFlags	指定的选项将被传递到 host 程序的基础编译器，若同时传递多个选项需使用逗号进行分割，中间不能有空格。
-LFlags	指定的选项将被传递到链接器，若同时传递多个选项需使用逗号进行分割，中间不能有空格。
-priv	**辅助选项**：对加速区进行私有化变量分析，给出变量的读写属性，遇到无法识别的函数调用则分析终止
-priv:ignore_call	**辅助选项**：对加速区进行私有化变量分析，给出变量的读写属性，忽略加速区内可能存在的函数调用的影响
-arrayAnalyse	**辅助选项**：数组访问模式分析，可以给出可 copy 的数组的建议
-ldmAnalyse	**辅助选项**：设备内存使用情况分析，编译之后在运行时会反馈各加速计算区内对设备内存的使用情况及相关建议
-preinline	**辅助选项**：对含有#inline 指示的函数调用语句中的函数进行内联，不对其他的加速指示进行处理，主要用于辅助函数内联
-preinline:all	**辅助选项**：在-preinline 的基础上，对同一个函数内的所有同名的被调用函数进行内联。
-v|-version	显示编译器和运行时库的版本号
-Minfo	显示编译器关于程序的分析信息
-keep	保留编译的中间文件
-dumpcommand [dumpfile]	将对中间文件的编译命令输出到 dumpfile 中

[注：通用编译选项可使用命令 man gcc 等查看相关功能]

2.3程序及编译运行示例

本节通过一个简单的示例程序介绍如何在神威太湖之光计算机系统主机环境下使用 SWACC 编译系统编译及运行 OpenACC*程序。

```
1 #include <stdio.h>
2
3 main()
4 {
5       int A[1000];
6       int i;
7       #pragma acc parallel loop copyout(A)
8       for(i = 0; i < 1000; i ++)
9               A[i] = i;
10 }
```

图 2-1 简单的 OpenACC*示例

图 2-1 中，第 7 行为 OpenACC*的组合加速编译指示 parallel loop，用于指定紧随其后的加速计算区，并且加速计算区内仅有一个加速循环，第 8-9 行为对应的加速循环体。其中 copyout(A)子句表明 A 数组将在设备内存中申请必要的空间，并在计算结束后将设备内存中 A 数组的数值拷贝回主存中对应的地址空间。

将图 2-1 中的程序保存为 test.c，使用下面编译命令进行编译：

swacc test.c

SWACC 编译系统会将加速计算区的代码封装为加速任务，对于加速循环中的循环和相应的数据会以最优的方式映射到加速设备上运行，默认生成可执行文件 a.out。

提交运行命令如下：

bsub -I -q q_test -N 1 -np 64 -priv_size 4 ./a.out

其中 bsub 为系统提交作业的命令，-q q_test 为指定提交作业所需的队列名，-N 1 指定启动一个主进程运行，-np 64 指定需要 64 个加速线程进行加速（作业提交运行的具体使用方式参考相关手册）。

使用 OpenACC*进行应用移植开发的步骤和注意事项，请参考第 5 章的内容，对于初次接触 OpenACC*语言的用户建议优先阅读。

2.4编译系统支持的常用功能

2.4.1 变量私有化分析功能

在本文所述的 OpenACC*加速编程模型中，主存空间默认是主进程和加速线程共享的，因此准确确定私有化变量直接关系到程序运行的正确性。用户可以在编译时添加编译选项 -priv ，SWACC 编译系统将对标注 acc parallel loop 的核心代码段中需要私有化的变量、只读标量和只读数组进行自动分析。若存在无法分析的变量，则需要用户分析并确定相关属性以确保程序执行结果的正确性。图 2-2 是使用编译选项-priv 对某程序中标注 acc parallel loop 的核心段进行私有化分析的结果：

```
编译指示位置信息：/ldata/zlb/for_swacc_tune/gkufs/algorith.f : 94
    需要私有化的变量列表：jcbijk,FDVX,FDV0,FDV1,FDVNijk,cfijk,cf2dt,i,j,k,cpr,RONijk,Uijk,Vijk,Wijk,Tijk,Pijk,vijk2,cpt,crt,viu,vjv,
vkw,FDVM,FVQ;建议您将其加入到private或者local子句
    书写格式：local(jcbijk,FDVX,FDV0,FDV1,FDVNijk,cfijk,cf2dt,i,j,k,cpr,RONijk,Uijk,Vijk,Wijk,Tijk,Pijk,vijk2,cpt,crt,viu,vjv,vkw,FD
VM,FVQ)
    只读标量列表：im,jm,km,DThalf,DTsource,iv,jv,kv,PI,RCON,vi_sj,vj_sj,vk_sj; 建议您将其加入到copyin子句和annotate的readonly子句
    书写格式：copyin(im,jm,km,DThalf,DTsource,iv,jv,kv,PI,RCON,vi_sj,vj_sj,vk_sj) annotate(readonly=(im,jm,km,DThalf,DTsource,iv,jv,
kv,PI,RCON,vi_sj,vj_sj,vk_sj))
    只读数组列表：CF,jcb,S
    分析不清楚的变量列表:FDV
```

图 2-2 编译选项-priv 自动分析示例

如图 2-2 所示，SWACC 编译系统列出了相关子句的书写格式及调优建议，用户可以直接将其添加到 acc parallel loop 的编译指示中。

2.4.2 数组分布性分析功能

用户在编译时可以添加编译选项-arrayAnalyse，SWACC 编译系统将自动对标注 accparallel loop 的核心代码段中可以拷贝到加速设备的数组变量及需要私有化的数组变量进行分析，图 2-3 是对标注 acc parallel loop 的核心段进行数组分布性分析的结果：

```
--------------------------------------------------------------------------------
编译指示位置信息：/ldata/zlb/for_swacc_tune/gkufs/algorith.f : 95
    可以copy的变量列表：FDV,CF,jcb,S
--------------------------------------------------------------------------------
```

图 2-3 编译选项-arrayAnalyse 自动分析示例

用户可以结合私有化分析结果，对可 copy 的变量进行进一步的细分，如果数组位于只读数组列表，那么将其加入到 copyin 子句，否则就加入到 copy 子句。

2.4.3 设备内存空间优化分析功能

用户在编译时如果添加编译选项-ldmAnalyse 进行编译，那么程序在运行时将反馈给用户当前设备内存的详细使用情况并自动分析出一种较优的循环及数据分块值（关于循环分块，可参考 3.5.5 循环分块子句 tile 的说明），图 2-4 是对标注 acc parallel loop 的核心段进行设备内存优化分析的结果示例：

```
================================================================================
LDM 空间优化分析结果：
acc region 信息：
  文件位置信息：/home/export/base/pfsacc/zlb/gkufs/algorith.f : 94
  编译器参数使用空间大小：24 字节
  region上的数据拷贝变量列表：

         im          jm          km      DThalf    DTsource          iv          jv          kv
         PI        RCON       vi_sj       vj_sj       vk_sj      jcbijk        FDVX        FDV0
       FDV1     FDVNijk       cfijk       cf2dt           i           j           k         cpr
     RONijk        Uijk        Vijk        Wijk        Tijk        Pijk       vijk2         cpt
        crt         viu         vjv         vkw        FDVM         FVQ
  region上数据拷贝变量所使用的LDM空间大小：152 字节
  acc do 信息：
    文件位置信息：/home/export/base/pfsacc/zlb/gkufs/algorith.f : 95
    block指示信息：block(k:1,j:1)
    acc do上分布数组列表：

             FDV(64,1,1,12)          CF(64,1,1)        S(19,64,1,1)         jcb(64,1,1)
    acc do上分布数组所使用的LDM空间大小：8448 字节
总计：已使用的LDM空间大小为：8624 字节
我们建议使用的block指示信息如下：
  文件位置信息：/home/export/base/pfsacc/zlb/gkufs/algorith.f : 95
    block指示信息：block(k:1,j:2)
================================================================================
```

图 2-4 ldm 空间优化分析结果示例

3 编译指示

本章介绍 OpenACC*编译指示的语法和使用方式，在 C 和 C++中，用#pragma 机制来描述 OpenACC*编译指示。在 Fortran 中，用带有独特前导符的注释来描述 OpenACC*编译指示。对于不支持或者关闭了 OpenACC*的编译器而言，OpenACC*编译指示将被忽略。

3.1编译指示格式

在 C/C++中，使用#pragma 机制描述 OpenACC*指示，其语法格式为：

#pragma acc 编译指示名 [子句[[,] 子句]...] 换行

每个编译指示均以#pragma acc 开头。编译指示的其它部分都遵守 C/C++中 pragma 的使用规范。#的前后都可以使用空白字符；编译指示的多个子句之间使用空白字符或逗号分隔。*在 C/C++程序中编译指示区分大小写*。OpenACC*编译指示作用于紧接着的语句、结构块和循环。C/C++语言中使用\作为编译指示的续行符。

在 Fortran 语言自由格式的程序中，以**!$acc** 前导符作为 OpenACC*编译指示的起始标记，其语法格式为：

!$acc 编译指示名 [子语[[,]子语]...]

第一个注释字符!可以出现在一行的任意列，但它前面只能是空白字符(空格和跳格)。前

导符!$acc 必须以一个整体出现，中间不能有空白字符。每行的长度、空白字符、续行符规则同样适用于编译指示行。编译指示起始行的前导符(!$acc)后必须有一个空白字符。待续行中导语部分的最后一个非空白字符必须是续行符(&)，续行符后面仍然可以写注释；接续行中必需以前导符开始(前面允许有空白字符)，前导符后面的第一个非空白字符可以是续行符。

在 Fortran 固定格式的程序中，OpenACC*编译指示可以采用下列形式中的一种：

!$acc 编译指示名 [子语[[,]子语]...]

c$acc 编译指示名 [子语[[,]子语]...]

***$acc 编译指示名 [子语[[,]子语]...]**

前导符（!$acc，c$acc，*$acc）必需写在 1-5 列，固定格式的每行长度、空白字符、续行、列的规则同样适用于编译指示行。编译指示起始行的第 6 列必须是空格或者 0，接续行第 6 列不能是空格或 0。

在 Fortran 程序中编译指示不区分大小写。后续章节中 Fortran 相关的用法及示例均以自由格式进行说明。

每个编译指示中只能出现一个编译指示名，多个子句出现的顺序无关紧要，除非特别限定，同一个子句可以重复出现多次。

3.2条件编译

预定义的宏_OPENACC 的值为 yyyymm, 其中 yyyy 是编译器所支持 OpenACC*版本的发布年份，mm 是月份。当且使用 SWACC 编译系统编译 OpenACC*程序时，_OPENACC 宏被定义，其值为 201306。

3.3编译指示一览

OpenACC*的编译指示及子句如图 3-1：

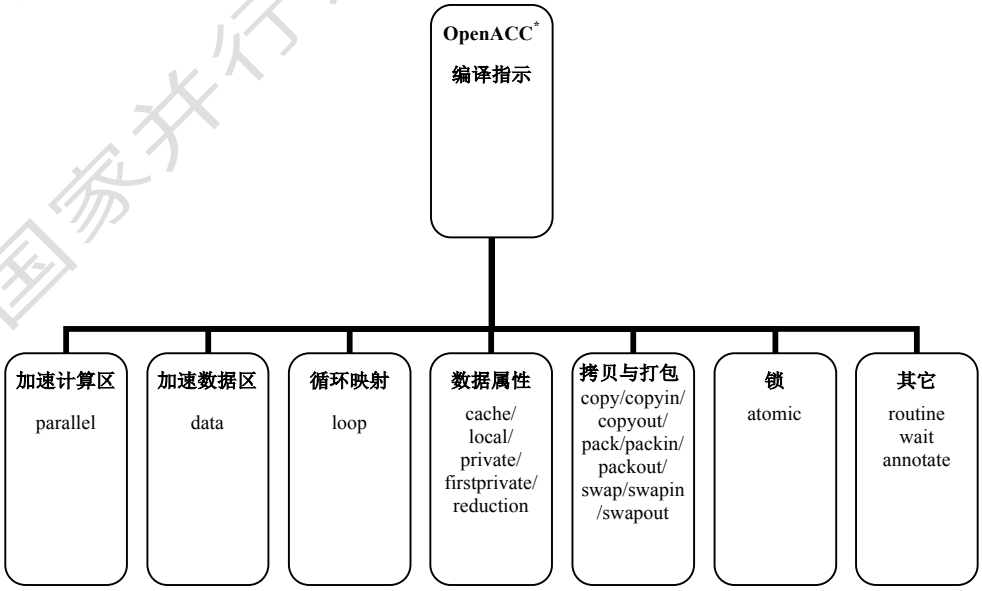

图 3-1 OpenACC*编译指示概览图

图 3-1 是 OpenACC*加速编程语言的主要编译指示概览图，OpenACC*编译指示和子句的功能如下表 3-1 和表 3-2：

表 3-1 编译指示

编译指示	说明	子句
parallel	用在一个代码段之前，表示这段代码将作为加速任务加载到加速设备上执行。	if private firstprivate reduction cache local copyin/copy/copyout packin/pack/packout swapin/swap/swapout wait async num_gangs num_workers annotate
data	指定数据的属性，其作用范围根据在程序中的位置有两种： 1) 在 parallel 外使用作用于多个 parallel； 2) 在内部使用作用于相应代码块。	if copyin/copy/copyout present index
loop	用于循环前面，标记加速循环。	private reduction nowait tile gang worker vector collapse annotate
parallel loop	parallel 和 loop 指示的组合，用在一个循环之前，表示循环的代码将被多个加速线程并行执行。	parallel 和 loop 指示子句的组合
atomic	用于指定一块内存区域被自动更新。	无
routine	用于说明将在加速计算区中被调用的函数	无
wait	等待在此语句之前运行的异步任务结束。	async

表 3-2 编译指示子句

编译指示子句	说明
private	将一个或若干个变量指定为加速线程私有变量
firstprivate	将一个或若干个变量指定为加速线程私有变量，并且变量赋初值，初值为主进程中的值
reduction	指定一个或多个变量是私有的，并且在并行处理结束后这些变量要执行指定的运算
nowait	loop 编译指示之后的 nowait，忽略任务执行中暗含的等待

if	如果 parallel 指示存在 if 子句，则加速区中的代码当且仅当<逻辑表达式>值为 TRUE 时并行执行，其它情况下，加速区被串行执行； 如果 data 指示存在 if 子句，则当且仅当<逻辑表达式>值为 TRUE 时发生数据的拷贝，其他情况下，不发生数据拷贝
cache	暗示变量非常重要，需尽可能放到最快的内存中，如局存、软 cache，使用最有效的存取方式，具体由编译器优化时决定
local	Local 变量列表中的变量、数组等，必须在设备内存中申请空间，改变主存中同名变量的值不会影响到加速设备，加速设备上对这些变量的值做的改变也不影响 host
copy/copyin/copyout	可指定把标量、数组、数组的全部或一部分在主存和局存之间拷贝，该子句可跟在 parallel、data 多个编译指示后，语义相同，但限制可能有所不同
pack/packin/packout	把多个变量打包为一个大变量，以减少拷贝数据到局存的操作次数。
swap/swapin/swapout	对数组进行转置，以消除不连续的访问。
num_gangs/num_workers	指定加速线程的 gang 和 worker 的数目
gang/worker	将循环映射到 gang 或 worker 上执行
present	指定相关数据已在设备内存中
tile	将循环按指定的块大小分割成两重循环
collapse	指定有多少层紧嵌套的循环跟 loop 指示相关联，这些循环的所有迭代将被统一调度
index	对当前数据区进行编号，主要用于设置数据打包、转置的操作点
annotate	对循环映射、数组大小、数组属性(如只读、访问特点)等进行补充说明
async/wait	用于操作的异步执行和等待。

注：本文档中所描述的除 parallel 之外的指示（非子句），均需要在 parallel 或 parallel loop 内使用，除非有特殊说明。

3.4加速计算区指示 parallel

加速计算区编译指示定义了一个加速计算区，加速计算区内的代码将加载到加速设备上运行。

1) 语法

C/C++

```
#pragma acc parallel [子句列表]
        structured-block
```

C/C++

Fortran

```
!$ACC PARALLEL [子句列表]
        structured-block
[!$ACC END PARALLEL]
```

Fortran

子句：

async [(整数表达式)]
wait [(整数表达式列表)]
num_gangs(整数表达式)
num_workers (整数表达式)
if (条件语句)；
reduction (操作符: 变量列表)；
copy (变量列表)；
copyin (变量列表)；
copyout (变量列表)；
local (变量列表)；
pack(变量列表)；
packin (变量列表)；
packout(变量列表)；
swap(变量列表)；
swapin(变量列表)；
swapout(变量列表)；
private (变量列表)；
firstprivate (变量列表)；
cache (变量列表)；
annotate (暗示子句)。

2) 说明

当主进程遇到并行结构并且下列情况为真时，它就创建若干个 gang 来执行这个加速器并行区域，而每个 gang 又包含若干个 worker：

a) 没有 if 子句的出现；
b) if 中条件表达式值非 0(C/C++)或为.true.(Fortran)。

创建的这些 gang 一旦创建，gang 的数量和每个 gang 包含的 worker 数量在整个并行区域内都保持不变。接着，每个 gang 中的所有 worker 都开始执行构件中结构块的代码。gang 的数量由 num_gangs 指定，每个 gang 包含的 worker 数量由 num_workers 指定。num_gangs 和 num_workers 的数值必须是（作业提交时指定的）总线程数的公约数，且它们的乘积必须小于等于总线程数。如不指定，或指定的参数不符合上述要求，则由实现时决定实际运行的参数。

主进程创建好加速线程组后，不参与加速区域的执行，默认将等待加速线程组的执行结束。如果 if 表达式的值是 0，那么这个加速区由主进程串行执行。

加速线程组一旦创建，运行时系统不会改变线程组中的线程数，并在加速区执行期间保持不变。加速区动态范围中的语句由每个加速线程执行，并且每个加速线程可以执行与其他加速线程不同的语句轨迹。加速区的结尾隐含有一个栅栏。

如果有 async 子句，主进程在完成加速线程组的创建后，不等待加速线程组执行的结束，而直接执行后续代码，需要通过 wait 机制在后续代码中等待该加速任务的完成。

加速区存在以下限制：

a) 加速计算区指示 parallel 不支持嵌套，嵌套 parallell 的处理由实现时决定；
b) 不能有进出加速区的其他分支，否则可能会导致不确定的错误；
c) 程序的正确性不能依赖于 parallel 编译指示的子句出现的顺序；
d) if 子句最多只能出现一次。
e) 在加速区中不能访问私有属性的数据，比如 Fortran 程序中 module 中定义的

private 变量。

3) 示例

```
… …
#pragma acc parallel
{
      printf(“hello world\n”);
}
… …
```

图 3-2 编译指示 parallel 的程序示例

图 3-2 中定义了一个加速区，加速区中的代码将被所有加速线程执行。

3.4.1 if 子句

1) 语法

```
if(条件语句)
```

2) 说明

if 子语是 parallel 指示的可选项；没有 if 子语的时候，编译器为本区域生成加速器上执行的代码。

有 if 子语的时候，编译器为本区域生成两份代码，一份可以在加速器上执行，另一份可以在主机执行。对 C 和 C++语言，if 子语中的条件值为零时，对 Fortran 语言，if 子语中的条件值为.false.时，主机端代码将被执行。if 子语中条件的值在 C 和 C++语言中为非零时，或在 Fortran 语言中为.true.时，加速器端代码将被执行。

(3) 示例

```
main()
{
   int count=0;
   #pragma acc parallel if(count == 0)
      {
            count ++;
            printf(“count1=%d\n”,count);
      }
}
```

图 3-3 if 子句 C 程序示例

```
program main
      integer count
      !$ACC PARALLEL if(count .eq. 0)
      count = count + 1
      print *,”count=”,count
      !$ACC END PARALLEL
 end
```

图 3-4 if 子句 Fortran 程序示例

3.4.2 私有化子句 private/firstprivate

参见 3.11.1。

3.4.3 规约子句 reduction

参见 3.11.2。

3.4.4 本地化子句 local

1) 语法

```
local(变量列表)
```

2) 说明

变量列表中的变量、数组等需要在加速设备内存中申请空间，改变主存中同名变量的值不会影响到加速设备，加速设备上对这些变量值的改变也不影响主存中的变量，local 变量的初始值是不确定的，local 与 private 的唯一区别在于存储位置的不同。

变量列表中不支持子数组形式。

3) 示例

```
#pragma acc parallel loop copy(A) local(tmat)
for(i=0;i<N;i++)
{
      for (m = 0; m < 5; m++) {
            tmat[m] = …;
      }
      A[i]=A[i]+1.0/tmat[1];
      ….
}
```

图 3-5 local 子句示例

在图 3-5 中，因 tmat 是在加速循环中计算赋值且不在循环外使用，所以可声明为 local，将其存储位置置于设备内存上，以提高效率。

3.4.5 缓存子句 cache

1) 语法

```
cache(变量列表)
```

2) 说明

cache 子句指明需尽可能将变量列表中的变量放到最快的内存中（如加速设备局存），使用最有效的存取方式进行数据交互，具体实现由编译系统实现时决定。变量列表中指定的变量以逗号分隔；变量之后可通过冒号(:)指定其缓存大小(单位为 1 个 cache 行，系统默认为 2 个 cache 行)，例如 p:8，表示 p 占用 8 个 cache 行（其中一个 cache 行为 256 字节）。cache 变量的访问一致性由用户程序保证，通常用于只读变量或私有变量。

3) 使用限制

a） 对于需要修改内容的 cache 变量，不能有通过别名进行的访问；

b） cache 作用范围内不能对 cache 变量取地址；

c） cache 只作用于数组；

d） 对于 C 程序，栈数组不能使用 cache（以后能不能支持？）；

e） 用户程序保证 cache 变量的起始地址与长度都按 256 对界。例外的情况是，

用户程序可以不用考虑先私有化再使用 cache 的变量对界要求；

f） 一般是在无法使用 copy 机制的时候使用 cache。

4) 示例

```
#pragma accparallel loop copyin(rowstr,a) copyout(w) cache(p:8,colidx)
for (j = 1;j <= lastrow-firstrow+1;j++)
{
    sum = 0.0;
    for (k = rowstr[j]; k < rowstr[j+1]; k++)
    {
        sum = sum + a[k]*p[colidx[k]];
    }
    w[j] = sum;
}
```

图 3-6 cache 子句示例

在图 3-6 中，因对数组 colidx 的访问是动态的，对数组 p 的访问是通过间接下标进行的，无法使用规则的 copy 语句，使用 cache 指示后，编译器将视情况选择尽可能快的内存来存放这两个数组。

3.4.6 数据拷贝子句 copy/copyin/copyout

参见 3.11.3。

3.4.7 数据打包子句 pack/packin/packout

对应数据传输而言，零散变量的传输效率比批量的数据拷贝效率要差，因此把零散变量打包为一个大的变量可减少数据拷贝的开销，数据打包子句与数据拷贝子句类似。

1) 语法

```
pack(变量列表[,at data index])
packin(变量列表[,at data index])
packout(变量列表[,at data index])
```

子句含义：

a) pack 子句：先打包，然后操作步骤同 copy 子句，之后解包；

b) packin 子句：先打包，然后操作步骤同 copyin 子句；

c) packout 子句：首先操作步骤同 copyout 子句，之后解包。

2) 说明

参数含义如下：

a) 变量列表以逗号分隔，支持标量、数组等。at data index 用于指定打包操作在程序中的位置，称之为打包点（其中 index 是常数，与 data 指示的 index(num) 的 num 对应），OpenACC*支持打包点的设置，如果不指定打包点，编译器默认在 data/parallel 的前后分别插入打包解包语句。如果用户指定打包点，编译器就在指定的 data 前后分别进行打包和解包；指定打包点时需要注意，打包点的作用是将数据打包的操作放到打包点处进行操作，因此要保证在打包点和当前位置之间相关的数据不会发生改变，否则会导致结果错。

b) 默认情况下，pack/packin/packout 会根据循环的划分方式将对应的数据进行分割。如果用户使用暗示语句指出数据需要全部拷入，packin 会将对应的数据整个拷贝到加速设备内存上，一般用于只读数据

3) 示例

```
int A[128];
int x,y,z;

#pragma acc parallel loop packin(x,y,z) copyin(A)
for (i = 0; i < Natoms; i++)
{
      t=A[x+y+z];
}
```

图 3-7 数据打包子句例子 1

在图 3-7 中的程序将标量 x、y、z 打包为一个大的变量，然后通过一次数据拷贝操作到每个加速线程的设备内存中，打包的语句插在加速段之前。

```
int A[N],B[N] ,C[N];
int x,y,z;
#pragma acc data index(1)
{
    for(;;)
       for(;;)
          #pragma acc    parallel loop packin(A,B,at data 1) copyout(C)
          for (i = 0; i < Natoms; i++)
          {
              C[i] = A[i] + B[i];
          }
}
```

图 3-8 数据打包子句例子 2

在图 3-8 中，程序通过 at data 1 指定了数组 A、B 打包操作的数据点（或称为打包点），其中的数字 1 与 data 指示中的 index(1)相对应，即数组 A、B 打包的操作实际上是在 data 指示处进行。由于数组 A、B 在循环内是只读的，因此可以将打包操作提前到循环之外，这样可以减少打包/解包操作的次数，避免不必要的开销。由于数组 C 在循环内被写覆盖，所以只需要 copyout，而不需要和数组 A、B 一起打包。

3.4.8 数组转置子句 swap/swapin/swapout

对访问方式不连续的数组的拷贝，会导致编译器生成带跨步的数据拷贝操作，性能相对较差，用户可以使用转置编译指示来标识这种数组。编译器将数组转置后，可通过连续的数据拷贝操作进行拷贝，以提高数据拷贝的性能。

1) 语法

```
swap(数组名((dimension order:,,,….),…..[at data index])
swapin(数组名((dimension order:,,,….),…..[at data index])
swapout(数组名((dimension order:,,,….),…..[at data index])
```

子句含义：

a) swap 子句：先将数组进行转置操作，然后操作步骤同 copy 子句一致，之后再转置回原数组；

b) swapin 子句：先将数组进行转置操作，然后操作步骤同 copyin 子句一致；

c) swapout 子句：先将程序中的数组访问替换成对转置后的数组访问，然后操作步骤同 copyout 子句，再转置回原数组。

2) 说明

默认情况 下，swap/swapin/swapout 会根据循环的划分方式将对应的数据进行分割，如果用户用暗示语句指出数据需要全部拷入，swapin 会将对应的数据整个拷贝到加速设备内存上，一般用于只读数据。

OpenACC*支持转置点的设置，如果不指定转置点，编译器默认在 data/parallel 标识的代码区域的开始或结束位置插入转置或反转置语句。如果用户指定转置点，编译器就在指定的 data 标识的代码区域开始或结束位置进行转置或反转置。指定转置点时需要注意，转置点的作用是将数据转置的操作放到转置点处进行操作，因此要保证在转置点和当前位置之间相关的数据不会发生改变，否则会导致结果错。

注意：转置功能需要在主存中申请存放转置后数据的空间，如内存空间无法满足需求请谨慎使用。

维度次序（dimension order），标识数组维度的编号，从低维到高维递增：

a) Fortran 程序的数组维度按从左到右编号：1，2，3，4，…其中 1 表示最低维；描述数组维度顺序也是从左到右排列，即从低维到高维，如三维数组的维度顺序是(1,2,3)；

b) C 程序的数组维度按从右到左编号：1，2，3，4，…描述数组维度次序时按照从右到左排列,即从低维到高维，如三位数组的维度顺序是(3,2,1)。

3) 示例

```
extern float FDV[32][64][128];
extern float FDV1k,FDV1[64],FDV0T[64];

int test_swap(int im,int jm, int km)
{
        int i, j, k;
        #pragma acc parallel loop swapin(FDV(dimension order:2,1,3))
        for（j = 0; j< jm; j++）
              for(i = 0;i< im;i++)
                    for(k = 0;k < km;k ++)
                    {
                          FDV1k=FDV[k][j][i];
                          FDV1[k]=FDV1k;
                          FDV0T[k]=FDV1k;
                    }
        return 0;
}
```

图 3-9 数组转置子句示例

在图 3-9 的例子中，变化最快的内层循环 k 对应数组 FDV 的最高维，也就是对 FDV 的访问是不连续的，这种访问方式在多核系统中，其 cache 命中率也是很低的，效率不高。其中 swapin 的作用是对 FDV 数组进行转置，FDV 数组的维度次序是(3,2,1)，转置后新数组的维度顺序是(2,1,3)，如 FDV_new[64][128][32]。同时程序中对 FDV 的访问(FDV[k][j][i])将会被替换成对转置后数组 FDV_new 的访问(FDV_new[j][i][k])。转置后对 FDV_new 的访问是连续的，可以较大的改善程序性能。

3.4.9 num_gangs 子句

1) 语法

```
num_gangs_(整数表达式)
```

2) 说明

parallel 构件允许使用 num_gangs 子语。整数表达式的值规定并行执行该区域的 gang 的数量。如果没有本子语，编译器将使用自定义的默认值。

3) 示例

```
int n=32;
#pragma acc parallel num_gangs(n)
{
  printf("Hello.");
}
```

图 3-10 num_gangs 子句示例

图 3-10 的例子中设定了并行执行该区域的 gang 的数量是 32。

3.4.10 num_workers 子句

1) 语法

```
num_works_(整数表达式)
```

2) 说明

parallel 构件允许使用 num_workers 子语。整数表达式的值规定了执行该区域的每个 gang 中所包含 worker 的数量。如果没有本子语，编译器将使用自定义的默认值；默认值可能是 1。

3) 示例

```
#pragma acc parallel num_workers(16)
{
    printf("Hello.");
}
```

图 3-11 num_workers 子句示例

图 3-11 的例子中设定了并行执行该区域的每个 gang 中 worker 的数量是 16。

3.4.11 等待子句 wait

1) 语法

```
wait [(整数表达式列表)]
```

2) 说明

wait 子句可以作为 parallel、data 等指示的子句使用，其作用与 wait 指示相同，用于等待之前的异步操作完成。当编译指示中出现 wait 子句时，必须先等待 wait 子句关联的异步操作完成才执行相应指示所对应的操作。wait 子句的参数必须与之前某个 async 子句的参数一致，wait 子句的参数可以为多个整数表达式，使用逗号间隔；当 wait 子句无参数时，等待之前所有的异步操作完成。

3) 示例

```
#define NUM 64
```

```
int i,a[NUM],b[NUM];
#pragma acc parallel loop async(0) copyout(a)
for(i=0;i<NUM;i++)
{
    a[i]=1;
}
#pragma acc parallel loop wait(0) copyout(c) copyin(a)
for(i=0;i<NUM;i++)
{
    c[i]=a[i]+2;
}
```

图 3-12　wait 子句示例

3.4.12 异步子句 async

参见 3.11.4。

3.4.13 暗示子句 annotate

参见 3.11.5。

3.5 循环映射指示 loop

编译指示 loop 作用于紧跟其后的循环，主要用于描述执行该循环所采用的并行方式等属性。

1)　语法

C/C++

```
#pragma acc loop [子句列表]
    for 循环
```

C/C++

Fortran

```
!$ACC LOOP [子句列表]
    do 循环
[!$ACC END LOOP]
```

Fortran

其中的子句是下列之一：

collapse(正整数常数)；

gang；

worker；

vector；

tile (整数表达式列表) ；

private(变量列表) ；

reduction (操作符: 变量列表)；

nowait；

annotate (暗示子句)。

2) 说明

OpenACC*要求加速循环必须是规范的，对循环语句有如下限制(假设循环变量为i)：

a) 初值表达式：只支持形式 i=expr；

b) 条件表达式：只支持 i<expr、i<=expr、i>expr、i>=expr，其余均非法；

c) 步进表达式：只支持形式为 i+=expr、i-=expr、i++、i--、i=i+expr、i=i-expr；

d) 循环中不能有中断该循环的语句，如 break，跳出循环的 goto 等。

e) 循环变量的值除了在步进表达式中被修改之外，在循环执行过程中不能修改循环变量的值，否则程序运行可能会出错。在循环执行结束后循环变量中的值是不确定的。

loop 指示需要在 parallel 的动态范围内使用，否则将不产生实际作用。需要并行执行的 loop 指示必须出现在 parallel 的语法范围内，即出现在被 parallel 加速计算区调用的函数内的 loop 指示只能是串行执行。在 parallel 内，最多只允许出现一个与 parallel 嵌套的 loop 指示，不支持 parallel 内嵌套多个并列的 loop 指示，loop 指示间的嵌套不受此限，否则有可能导致错误的结果。

对 Fortran 的要求类似，并行执行的 loop 如果没有指定 nowait 子句，则在循环结束之后会默认执行加速设备内加速线程同步。

当 parallel 中的 loop 指示均没有指定 gang、worker 子句时，默认情况下最外层的 loop 指示作用的循环将在多个 gang 间并行执行，gang 的数目等于加速线程的数目，每个 gang 中只有一个 worker。如果不需要多级并行，则可以不指定 gang、worker 子句。

loop 指示的执行方式有四种，按照从高到低的层次分别为 gang、worker、串行、vector，当多个 loop 指示嵌套时，需要注意执行方式的嵌套层次。

当 loop 指示出现在 parallel 加速计算区所调用的函数中时，实际有效的子句仅为 vector，其他子句将被忽略。

3) 示例

```
#pragma acc parallel copyin(a) copyout(r)
{
    #pragma acc loop
    for( i = 0; i < n; ++i )
    {
        s = sin(a[i]);
        c = cos(a[i]);
        r[i] = s*s + c*c;
    }
}
```

图 3-13 编译指示 loop 的程序示例

3.5.1 gang 子句

1) 语法

```
gang
```

2) 说明

在一个 parallel 加速区中， gang 子语要求将关联的循环中的迭代步分摊到 parallel 指示创建的多个 gang 上，从而实现并行执行。循环的所有迭代步必须没有相关性，但在 reduction 子语中指定的变量除外。带有 gang 子句的 loop 指示内不允许出现带有 gang 子句的 loop 指示。

3) 示例

通常 gang 子句和 worker 子句配合使用来实现多级循环划分，示例见图 3-14。

3.5.2 worker 子句

1) 语法

```
worker
```

2) 说明

在一个加速器 parallel 区域内， worker 子语指明，关联循环的迭代步将被分散给单个 gang 的多个 worker 并行执行。循环的所有迭代步必须是数据独立的，但 reduction 子句中指定的变量除外。带有 worker 子句的 loop 指示中不允许出现 gang、worker 子句的 loop 指示。

3) 示例

```
#include<stdio.h>
#define N 100
int A[N][N];

main()
{
    int i,j;
    #pragma acc parallel num_gangs(8) num_workers(2)
    {
        #pragma acc loop gang
        for( i = 0; i < N; i++ )
            #pragma acc loop worker
            for( j = 0; j < N; j++)
            {
                A[i][j]= i+j;
            }
    }
    for( i = 0; i < N; i++ )
        for( j = 0; j < N; j++ )
        {
            if(A[i][j]!=i+j)
                {
                    printf(“test fail!”);
                    exit(0);
                }
        }
    printf(“test ok!”);
}
```

图 3-14 worker 子句示例

例中表示将 j 层 for 循环的迭代步分摊到每个 gang 里面所包含的两个 worker 上面，从而实现并行执行。

3.5.3 向量化 vector 子句

1) 语法

```
vector
```

2) 说明

提示编译器对指定的循环进行 SIMD 向量化并行。一般用于串行执行的循环。该功能暂时未支持。

3) 示例

```
#include<stdio.h>
#define N 100
int A[N][N][N];

main()
{
    int i,j,k;
    #pragma acc parallel num_gangs(8) num_workers(2)
    {
        #pragma acc loop gang
        for( i = 0; i < N; ++i )
           #pragma acc loop worker
           for( j = 0; j < N; ++j)
              #pragma acc loop vector
                for (k = 0; k < N; ++k)
                {
                   A[i][j][k]= i+j+k;
                }
     }
     for( i = 0; i < N; i++ )
          for( j = 0; j < N; j++ )
             for(k = 0; k < N; k++ )
             {
                if(A[i][j][k]!=i+j+k)
                {
                   printf(“test fail!”);
                   exit(0);
                }
             }
      printf(“test ok!”);
}
```

图 3-15 vector 子句示例

例中表示对最内层的串行执行的 for 循环进行 SIMD 向量化并行。

3.5.4 循环合并子句 collapse

1) 语法

```
collapse(正整数常数)
```

2) 说明

collapse 子句用于指定有多少层紧嵌套的循环与 loop 指示关联，collapse 的参数必须是一个常量正整数，如果没有 collapse 子句，loop 指示只作用于紧跟的循环。与 collapse 关联的所有循环的步长必须是可计算的，并且在循环执行过程中保持不变。

3) 示例

```
#include<stdio.h>
#define N 100
int A[N][N];
main()
```

```
{
  int i,j;
  #pragma acc parallel
  {
       #pragma acc loop collapse(2)
       for( i = 0; i < N; i++ )
          for( j = 0; j < N; j++)
          {
              A[i][j]= i+j;
          }
   }
   for( i = 0; i < N; i++ )
      for( j = 0; j < N; j++ )
      {
        if(A[i][j]!=i+j)
        {
          printf(“test fail!”);
          exit(0);
        }
      }
    printf(“test ok!”);
}
```

图 3-16 collapse 子句示例

例中表示将 parallel 并行区中的 i、j 两层循环合并。

3.5.5 循环分块子句 tile

1) 语法

```
tile(块大小)
tile(块大小列表)
tile(循环迭代变量 i:块大小 1, 循环迭代变量 j:块大小 2, …)
```

2) 说明

块大小为正整数常数，块大小列表由多个块大小组成，中间以逗号分隔，tile 子句有三种具体的用法。

用法 1：tile(块大小)，仅作用于当前 loop 指示，表示 loop 指示后紧跟的循环将以该块大小进行分割。

用法 2：tile(块大小列表)，当列表中有 n 个参数时，则要求 loop 指示后有 n 个紧嵌套的循环，列表中第一个参数作用于最内层循环，最后一个参数作用于最外层循环，依次类推。

用法 3：tile(循环迭代变量 i:块大小 1, 循环迭代变量 j:块大小 2, …)，作用于 loop 指示内直接嵌套的所有循环，与列表中循环变量相同的循环使用对应的块大小进行分割，比如循环迭代变量是 i 的循环，均以块大小 1 进行分割，以此类推。用法 3 是对用法 1 的补充，当存在多个相同循环变量的循环时，使用方式 3 较为简便。需要注意的是，当同时以不同的方式对同一个循环设置块大小，且块大小不一致时，由实现时决定。

3) 示例

```
#include<stdio.h>
int A[64][4096];
int B[4096];
main()
```

```
#include<stdio.h>
int A[64][4096];
int B[4096];
main()
```

```
{
    int i, j;
    int sum=0;
    for(i=0;i<4096;i++)
      B[i]=1;
    #pragma acc parallel copy(A,B)
    #pragma acc loop
    for(i = 0; i < 32; i ++)
    {
        #pragma acc loop tile(1024)
        for(j = 0; j < 32; j ++)
        {
            sum += B[j] ;
        }
        #pragma acc loop tile(1024)
        for(j = 0; j < 32; j ++)
        {
            A[i][j] += sum;
        }
    }
    for(i=0;i<32;i++)
      for(j=0;j<32;j++)
      {
        if(A[i][j]!=32*(i+1))
        {
          printf(“Test fail!”);
          exit(0);
        }
      }
    printf(“Test OK!”);
}
```

```
{
    int i, j;
    int sum=0;
    for(i=0;i<4096;i++)
      B[i]=1;
    #pragma acc parallel copy(A,B)
    #pragma acc loop tile(i:1, j:1024)
    for(i = 0; i < 32; i ++)
    {
        for(j = 0; j < 32; j ++)
        {
            sum += B[j] ;
        }
        for(j = 0; j < 32; j ++)
        {
            A[i][j] += sum;
        }
    }
    for(i=0;i<32;i++)
      for(j=0;j<32;j++)
      {
        if(A[i][j]!=32*(i+1))
        {
          printf(“Test fail!”);
          exit(0);
        }
      }
      printf(“Test OK!”);
}
```

图 3-17 tile 子句示例

tile 子句使用示例如上图所示，左侧代码为使用方式 1 对 i 和 j 循环分块的用法，右侧代码是使用方式 3 实现同样效果的代码。

3.5.6 私有化子句 private

参见 3.11.1.1。

3.5.7 规约子句 reduction

参见 3.11.2。

3.5.8 暗示子句 annotate

参见 3.11.5。

3.6加速数据区指示 data

加速数据区编译指示 data 用于定义加速计算的数据区（简称加速数据区），可以用来描述变量在加速设备内的属性。

1) 语法

C/C++

```
#pragma acc data  子句[ 子句列表]
      structured-block
```

C/C++

Fortran

```
!$ACC DATA  子句[ 子句列表]
      structured-block
!$ACC END DATA
```

Fortran

其中，子句列表是可选项，可根据实际需要设置子句列表；子句列表中的子句以空格分隔。

子句可为下列之一：

if(条件表达式)；
copy(变量列表)；
copyin(变量列表)；
copyout(变量列表)；
local(变量列表)；
present(变量列表)；
index(非负整数常数)。

2) 说明

data 指示可以出现在加速计算区 parallel 之内和之外。

当 data 指示出现在加速计算区 parallel 内时，实际有效的子句为 if、copy、copyin、copyout、local、present，用于描述主存与设备内存间的数据拷贝操作。

当 data 指示出现在加速计算区 parallel 内时，data 指示可以嵌套使用。data 上数据拷贝的有效范围是当前 data 的作用域， 超出该作用域后，设备内存上的相关拷贝数据不保证继续有效。

当 data 指示出现在 parallel 指示之外时，实际有效的子句为 index，其它子句被忽略；index 子句的含义为标记当前数据区，并对当前位置编号为 index 中的数值，以便设置数据操作点（包括数据打包和数据转置）。

3.6.1 if 子句

1) 语法

```
if (条件表达式)
```

2) 说明

if 子句是可选的子句。

如果没有出现 if 子句，编译器将根据编译指示内容对编译指示作用区域内的代码进行变换，例如在代码区域前面申请局存空间并插入从主存拷入数据、在代码区域结束处拷出数据到主存的代码。

如果出现了 if 子句，编译器将根据 if 子句的条件表达式的值来确定是否对编译指示作用区域内的代码进行变换。当 if 子句的条件表达式的值为非 0（C 与 C++）或.true.（fortran）时，编译器将根据编译指示内容对编译指示作用区域内的代码进行变换；当 if 子句的条件表达式的值为 0（C 与 C++）或.false.（fortran）时，编译器将不对编译指示作用区域内的代码进行变换。

3) 示例

if 子句的例子 1：图 2-4 所示例子 test_data_if_001.c 测试在 data 编译指示上使用 if 子句。

```
function fun(i)
      integer i, fun
      fun = i + 1
      if(fun == 129) then
            fun =1
      endif
end

program main
      implicit none
      integer, external :: fun
      integer flag
      integer A(128,128), b(128,128),m,i,j,x

      do i=1,128
      do j=1,128
            A(j,i) = i + j
      enddo
      enddo

      !$ACC PARALLEL LOOP LoCaL(m,flag,j) copyout(B) tile(2)
      do i=1,128
            m = fun(i)
            flag=mod(m-1,2)
            !$ACC DATA COPYIN(A(*, m)) if(flag)
            do j=1,128
                  B(j, i) = A(j, m)
            enddo
            !$ACC END DATA
      enddo
      !$ACC END PARALLEL LOOP
      do i=1,128
            do j=1,128
                  if(A(j, i) /= i + j) then
                        print *, "TEST FAIL.A"
                        stop
                  endif
            enddo
      enddo
      do i=1,128
```

```
            m = fun(i)
            flag = mod((m-1),2)
            do j=1,128
                  if(flag == 1) then
                        if(B(j, i) /= A(j, m)) then
                              print *, "TEST FAIL.B"
                              stop
                        endif
                  else
                        x = m-1;
                        if(x == 0) then
                              x=128
                        endif
                        if(B(j, i) /= A(j, x)) then
                              print *, "TEST FAIL.B"
                              stop
                        endif
                  endif
            enddo
      enddo

      print *, "TEST OK"

end
```

图 3-18 编译指示 data 上 if 子句的程序示例

3.6.2 数据拷贝子句 copy/copyin/copyout

参见 3.11.3。

3.6.3 数据重用子句 present

1) 语法

```
present (变量列表)
```

2) 说明

present 可以作为 data 的子句使用，用于指定数据已经在之前的操作中被拷贝到设备内存，后面的一段代码将重用之前拷入的数据。变量列表内的变量可以是标量、数组或子数组。

present 主要用在被加速计算区所调用的函数(简称加速函数)中，当数据在加速函数执行期间已经存在于设备内存中，则可以在加速函数内使用 present 重用相应的数据。

present 可重用的数据包括 parallel、data 上指定的数据。使用 present 时，必须保证 present 所指明的数据已经存在于设备内存中。数据的一致性由用户程序保证。

3) 示例

```
#include <stdio.h>
#include <stdlib.h>

int A[128][128];
int B[128][128];
```

```
int func(int i)
{
	int j;
	#pragma acc data present(A[i][*]) copyout(B[i][*])
	for(j = 0; j < 128; j ++)
		B[i][j] = A[i][j];
}

main()
{
	int i,j;
	#pragma acc parallel loop copyout(A)
	for(i = 0;i < 128; i ++)
	{
		for(j = 0; j < 128; j ++)
		    A[i][j] = i + j;
		func(i);
	}


	for(i = 0; i < 128; i ++)
	for(j = 0; j < 128; j ++)
	{
		if(B[i][j] != i + j)
		{
			printf("TEST ERR.\n");
			exit(-1);
		}
	}
	printf("TEST OK!\n");
	return 0;
}
```

图 3-19 编译指示 data 上 present 子句的程序示例 1

```
#include <stdio.h>
#include <stdlib.h>

int A[128][128];
int B[128][128];
int C[128][128];

void func2(int i)
{
	int j;
	#pragma acc data present(A[i][*]) copyout(C[i][*])
	for(j = 0; j < 128; j ++)
	{
		C[i][j] = A[i][j];
		A[i][j] = 0;
	}
}

void func1(int i)
{
```

```
    int j;
    #pragma acc data copyin(A[i][*]) copyout(B[i][*])
    {
        for(j = 0; j < 128; j ++)
        {
            B[i][j] = A[i][j];

        }
        func2(i);
    }
}

int main()
{
    int i,j;
    for(i = 0; i < 128; i ++)
    for(j = 0; j < 128; j ++)
    {
        A[i][j] = i + j;
        B[i][j] = 0;
        C[i][j] = 0;
    }
    #pragma acc parallel loop
    for(i = 0;i < 128; i ++)
        func1(i);


    for(i = 0; i < 128; i ++)
    for(j = 0; j < 128; j ++)
    {
        if(A[i][j] != i + j)
        {
            printf("TEST ERR.A\n");
            exit(-1);
        }
        if(B[i][j] != i + j)
        {
            printf("TEST ERR.B\n");
            exit(-1);
        }
        if(C[i][j] != i + j)
        {
            printf("TEST ERR.C\n");
            exit(-1);
        }
    }
    printf("TEST OK!\n");

    return 0;
}
```

图 3-20 编译指示 data 上 present 子句的程序示例 2

3.6.4 本地化子句 local

1) 语法

```
local(varlist)
```

2) 说明

varlist 是变量列表，变量之间以逗号分割，支持标量、数组、子数组。和数据拷贝子句中变量列表的含义相同。

变量列表中的变量、数组等需要在加速设备内存中申请空间，改变主存中同名变量的值不会影响到加速设备，加速设备上对这些变量值的改变也不影响主存中的变量，local 变量的初始值是不确定的。

与数据拷贝子句 copy/copyin/copyout 的唯一区间是，local 子句没有主存和设备内存之间的数据拷贝。

3.6.5 index 子句

3) 语法

```
index(num)
```

4) 说明

当 data 编译指示在 parallel 编译指示的作用域之外使用时，实际有效的子句为 index。此处 index 子句的含义是将当前 data 编译指示所在位置编号为 num，其中 num 的值必须是整数（>=0），便于进行程序中数据操作点的设置（包括设置数据打包点和数据转置点）。具体用法还可参考“3.4.7 数据打包子句”和“3.4.8 数组转置子句”。

5) 示例

```
#include <stdio.h>
#include <stdlib.h>

#define NN 128

int A[NN], B[NN], C[NN];

int main()
{
    int i;

    for(i = 0; i < NN; i++)
    {
        A[i] = 1;
        B[i] = 2;
    }

  #pragma acc data index(1)
  {
      #pragma acc parallel loop packin(A, B, at data 1) copyout(C)
      for(i = 0; i < NN; i++)
      {
          C[i] = A[i] + B[i];
      }

  }
```

```
    for(i = 0; i < NN; i++)
    {
        if(C[i] != 3)
        {
            printf("Test Error! C[%d] = %d\n", i, C[i]);
            exit(-1);
        }
    }
    printf("Test OK!\n");
    return 0;

}
```

图 3-21 编译指示 data 上 index 子句的程序示例

3.7组合指示 parallel loop

1) 语法

C/C++

```
#pragma acc parallel loop [子句列表]
    for 循环
```

C/C++

Fortran

```
!$ACC PARALLEL LOOP [子句列表]
    do 循环
!$ACC END PARALLEL LOOP
```

Fortran

2) 说明

组合编译指示是指在一个加速计算区内嵌套一个循环编译指示的简短用法，其含义与显式的指明一个包含循环编译指示的计算区相同，组合编译指示支持所有计算区和循环编译指示所支持的子句。该编译指示不能出现在另一个加速计算区和循环映射区里面。

3) 示例

```
#pragma acc parallel loop copyin(a) copyout(r)
for( i = 0; i < n; ++i )
{
    r[i] = a[i] + i;
}
```

图 3-22 组合编译指示示例

3.8原子操作指示 atomic

1) 语法

C/C++

```
#pragma acc atomic
    expression-stmt
```

expression-stmt 是下面形式的表达式之一：

x binop=expr；

x++；
++x；
x--；
--x。

其中：

x 是一个标量类型的左值表达式，expr 是一个标量类型的表达式，并且它不引用 x 指定的变量，binop 是下列符号之一: +,*,-,/,&,^,|,<<,>>，binop、binop=、++、--都不是重载运算符。

C/C++

Fortran

```
!$ACC ATOMIC
      expression-stmt
```

statement 是如下形式之一：

x=x operator expr；
x=expr operator x；
x=intrinsic_procedure_name(x,expr_list)；
x=intrinsic_procedure_name(expr_list,x)。

其中：

x 是一个固有类型的标量变量，expr 是一个不引用 x 的标量表达式，expr_list 为一个逗号分隔的，非空标量表达式链，不能引用 x。当 intrinsic_procedure_name 引用 IAND，IOR 或 IEOR 时，expr_list 中只能有一个表达式，instrisic_procedure_name 是 MAX,MIN,IAND,IOR,IEOR 之一，operator 是+,*,-,/,.AND.,.EQV.,.NEQV.之一，expr 里的操作的优先权与 operator 相等或者更高，x operator expr 与 x operator （expr）算术相等，expr operator x 与(expr) operator x 算术相等，intrinsic_procedure_name 必须引用固有过程名，而不能是其他程序实体。赋值必须是固有赋值。

Fortran

2) 说明

编译指示 atomic 指定一个存储位置被原子更新，而不会被多个线程同时写，atomic 所保护的存储位置通常是位于共享主存的变量。只有对 x 指定的变量 load 和 store 操作是原子的，expr 求值不是原子的。对 x 所指定的变量进行读或写的操作中不允许有任何任务调度点。为了防止竞争情况，所有可能被并行更新的位置都必须用 atomic 指示来保护。

3) 示例

```
#pragma acc parallel loop
for (i=0; i<n; i++)
{
    #pragma acc atomic
    x[index[i]] += work1(i);

    y[i] += work2(i);
}
```

图 3-23 atomic 指示示例

3.9等待指示 wait

1) 格式

C/C++

C/C++
```
#pragma acc wait [(整数表达式列表)] 子句
```
C/C++

Fortran
```
!$ACC WAIT [(整数表达式列表)] 子句
```
Fortran

其中子句是：

async [(整数表达式)]

2) 说明

wait 指示用于等待之前的异步操作结束，异步操作包括加速计算区的加载(parallel)和之前的 wait 等操作。

wait 指示如果有参数，则参数必须是之前某操作的 async 子句的参数，可以同时等待多个异步操作结束，用逗号分隔；如果 wait 指示没有参数，则等待之前所有的异步操作结束。

3) 示例

参见 3.11.4。

3.10函数指示 routine

1) 格式

C/C++
```
#pragma acc routine
```
需要在函数的定义之前使用。

C/C++

Fortran
```
!$ACC ROUTINE [子句列表]
```
需要在被修饰的 subroutine 或 function 的声明区域内使用。

Fortran

其中，子句列表是可选项，可根据实际需要设置子句列表；子句列表中的子句以空格分隔。

子句可为下列之一：

memstack(size=非负整数常数**[; name=**名字字符串**][; list=(**变量列表**)])**；

reuseldm(size=非负整数常数**)**；

2) 说明

routine 指示用于说明将在加速计算区中被调用的函数，编译器将自动为对应的函数创建可在加速设备上执行的版本。

routine 指示的子句目前只能用于 Fortran 中。

限制：

a) 在 C/C++中，routine 所修饰的函数中不支持函数内 static 变量的使用；

b) 在 Fortran 中，routine 所修饰的函数中不支持带有 save 属性的变量。

3) 示例

C/C++
```
#pragma acc routine
int add(int a, int b)
{
```

```
    return a+b;
}
```

C/C++

Fortran

```
function add (a,b)
!$acc routine
integer add, a, b
add = a + b
end function
```

Fortran

图 3-24 routine 指示示例

3.10.1 memstack 子句

1) 语法

memstack(size=非负整数常数**[; name=**名字字符串**] [; list=(**变量列表**)])**

2) 说明

加-b 选项运行众核程序时，从核函数栈放在从核的局存中。如果某个函数的栈的容量超出局存的大小，会造成程序运行出错。memstack 子句将本从核函数的栈数组放到主存，以减少栈空间对局存的占用，从而可以让程序在-b 选项下正确运行。

memstack 有两个参数 size 和 list，其中 size 参数是必需的。

size 指示栈数组占用主存空间的大小。

name 给栈数组占用的主存空间取的名字。

list 是放入主存的栈数组列表，列表中的变量顺序指示了栈数组在主存中的排列顺序。栈数组放在主存中访问速度较慢，为提高程序性能，一般需要在执行过程中使用 data 指示逐步拷贝部分数据。通常每个数据需要发起一次拷贝，但如果拷贝的多个数据在主存中是连续的，则可以一次拷贝包含多个数组的大块数据，提高数据拷贝的效率。list 参数的作用是按需排列栈数组的在主存的存放顺序，便于编译器优化数据拷贝。list 参数一般和 reuseldm 子句配合使用。

3) 示例

```
        subroutine sub1()
        implicit none
        !$acc routine  memstack(size=40960; name=private_region1; list=(A, C,
D, B, E))
        integer :: A(2048), B(2048),C(2048), D(2048), E(2048)
        integer :: i, s1

        !$acc data local(E)

        !$acc data copyin(A, C, D)
        do i = 1, 2048
                E(i) = A(i) + C(i)
                E(i) = E(i) + D(i)
        end do
        !$acc end data

        !$acc data copyin(B, D)
        do i = 1, 2048
                 s1    = E(i) + B(i)
```

```
            s1    = s1 + D(i)
        end do
        !$acc end data

        !$acc end data
        end subroutine


        program main
        implicit none

        !$acc parallel
        call sub1()
        !$acc end parallel
        end program
```

图 3-25 memstack 子句示例

3.10.2 reuseldm 子句

1) 语法

```
reuseldm([size=非负整数常数]，[ucaller])
```

2) 说明

reuseldm 子句有 size 和 ucaller 两种参数，在使用时必须且只能带一个参数。

若 reuseldm 子句带 size 参数时，指示函数中同一层次的多个 data 语句拷贝的数据复用同一块 LDM 空间。size 参数指示了 LDM 空间的大小。当需拷贝的数据总大小超过 LDM 的可用容量时，可以使用该子句实现 LDM 空间的重用。注意，重用只发生在同一层次的 data 语句之间，嵌套的 data 语句使用的 LDM 空间不是重用而是叠加。

若 reuseldm 子句带 ucaller 参数时，一般用在在多层函数调用中的被调函数中，指示被调函数（callee）复用调用函数（caller）申请的 LDM 空间。该函数在 caller 中被调用时可用的 LDM 地址，是被调函数可以使用的 LDM 空间的首地址。

3) 示例

```
        subroutine sub2()
        implicit none
          !$acc routine reuseldm(ucaller) memstack(size=4096)
          integer ::T(1024)
          integer :: i
          !$acc data copyout(T)
            do i = 1,1024
              T(i) = i
            end do
        end subroutine

        subroutine sub1()
          implicit none
          !$acc routine reuseldm(size=20480) memstack(size=49152;
name=private_region1; list=(A, C, D, B, F, E))
          integer :: A(2048), B(2048),C(2048), D(2048), E(2048), F(2048)
          integer :: i, s1
```

```
!$acc data local(E) copyin(D)      ! 1-1

!$acc data copyin(A, C)             ! 2-1
do i = 1, 2048
        E(i) = A(i) + C(i)
        E(i) = E(i) + D(i)
end do
!$acc end data

   call sub2()

!$acc data copyin(B, F)             ! 2-2
do i = 1, 2048
         s1    = E(i) + B(i)
         s1    = s1 + F(i)
         s1    = s1 + D(i)
end do
!$acc end data

!$acc end data
end subroutine


program main
implicit none

!$acc parallel
call sub1()
!$acc end parallel
end program
```

图 3-26 reuseldm 子句示例

示例中，2-1, 2-2 的 data 语句数据拷贝会复用同一块 LDM 空间，1-1 和 2-1 的 data 语句使用的 LDM 空间是叠加的。函数 sub2 复用 sub1 申请的 LDM 空间，空间的起始地址跟 2-2 的起始地址相同。

3.11编译指示子句

本章主要介绍 OpenACC*编译指示中的子句。OpenACC*加速编译指示的子句大部分是对加速代码中的数据属性进行说明和限定；大部分子句接受一个变量列表参数，变量列表中的变量用逗号分隔，同时这些变量必须是当前代码结构块可见的。在编译指示子句里的所有变量要求是当前代码结构块可见的，可按照需求重复出现，但同一变量不能在多个子句中同时使用，除非有明确说明。

本章主要介绍可以作为不同编译指示子句出现的子句，对于仅在特定编译指示中使用的子句在对应的编译指示章节介绍。

3.11.1私有化子句 private/firstprivate

3.11.1.1 private

1) 语法

```
private(变量列表)
```

2) 说明

private 子句的作用是声明变量列表中的变量为每个加速线程私有。

在 private 子句中说明一个变量的行为如下：在相关联的代码结构块的执行过程中分配一个有自动存储周期的新对象（在当前代码结构块中是可见的），新对象的大小和对界属性由变量类型决定，且新对象的初始值是不确定的（与下文 firstprivate 最大的区别），其结果也不更新至主进程的对应内存中。

private 子句限制如下：

- 出现在 reduction 子句的变量必须是加速线程私有的；
- 变量列表中不支持子数组形式。

C/C++

- 在 private 子句中说明的变量不能有不完全的类型或引用类型。

C/C++

Fortran

- 出现在 private 子句里的变量必须是可定义的，或者是可分配的数组。
- 出现在 private 子句里的可分配数组，必须在结构的出、入口有“非当前分配”的分配状态。
- 假定大小的数组不能出现在 private 子句里。
- 出现在语句名字表中、变量格式表达式中和用来定义函数的表达式中的变量，不能出现在 private 子句里。

Fortran

3) 示例

```
#include<stdio.h>
#define N 100
int A[N];
main()
{
    int i;
    #pragma acc parallel private(i)
    {
        for( i = 0; i < N; ++i )
        {
                A[i] = i;
        }
    }
    for( i = 0; i < N; i++ )
    {
       if(A[i]!=i)
       {
           printf(“test fail!”);
           exit(0);
       }
    }
    printf(“test ok!”);
}
```

图 3-27 private 子句示例

3.11.1.2 firstprivate

1) 语法

```
firstprivate(变量列表)
```

2) 说明

变量列表中的变量语义与private子句类似，不同的是firstprivate子句需初始化每个新的私有变量，在结构块中有一个隐含的初始化说明，并且其初始化值为该变量在主进程中原始对象的值。

C/C++

对非数组类型的变量，初始化就像赋值；对数组（可以是多维），初始化就像从源数组拷贝元素到新数组的对应元素。

C/C++

Fortran

初始化和赋值一样。

Fortran

firstprivate 子句限制与 private 子句的限制一致。

3) 示例

```
#include<stdio.h>
#define N 100
int A[N];
main()
{
    int i,a=1;
    #pragma acc parallel
    {
        #pragma acc loop firstprivate(a);
        for( i = 0; i < N; ++i )
        {
                A[i] = a+i;
        }
    }
    for( i = 0; i < N; i++ )
    {
       if(A[i]!=i+1)
       {
            printf(“test fail!”);
            exit(0);
       }
    }
    printf(“test ok!”);
}
```

图 3-28 private 子句示例

3.11.2 规约子句 reduction

1) 语法

```
reduction(operator:list)
```

其中，operator 指明操作类型，具体支持的操作在下面说明；list 为变量列表，变量之间以逗号分隔。

2) 说明

规约子句 reduction 可以在加速计算区及并行循环上使用，用来并行完成单个标量型变量上的规约计算。

编译器会在每个线程上为 list 表中的每个变量创建一个私有副本，并根据 operator

及变量的数据类型对该私有副本进行初始化；在指定了 reduction 子句的区域的结尾处，使用 operator 操作符对同一个变量的所有私有副本进行一次并行规约操作，并将规约计算的结果与变量的原始值合并，最终结果写回原始变量中。

operator 支持的操作及规约变量副本的初始值如表 3-1 所示。

表 3-3 operator 支持的操作及规约变量副本的初始值

C/C++		Fortran	
operator	初始值	operator	初始值
+	0	+	0
*	1	*	1
max	极小值	max	极小值
min	极大值	min	极大值
&	~0	iand	所有位为 1
|	0	ior	0
^	0	ieor	0
&&	1	.and.	.ture.
||	0	.or.	.false.
		.eqv.	.true.
		.neqv.	.false.

规约变量的数据类型必须是对所指定的 operator 有效的，支持基本的数值类型包括 C/C++语言的整型、浮点型和复数类型以及 Fortran 语言的 integer、real、double precision、complex、logical 等数据类型。

需要注意的是，在规约计算中值结合的顺序是不确定的，因此，比较串行和并行执行或者比较不同的并行执行（即使线程数目相同），并不能保证得到的结果的每一位完全相同（例如浮点异常）。

3) 示例

图 3-29 所示例子 test_reduction_001.c 测试在 parallel 编译指示上使用 reduction 子句。

```
/*
*   文件名：test_reduction_001.c
*   目   的：测试在 parallel 编译指示上使用 reduction 子句
*/

#include <stdio.h>
#include <stdlib.h>

#define SIZE       1024

long A[SIZE];

main()
{
      int i;
      long A_total = 123, tmp_a;

      #pragma acc parallel private(tmp_a) copyout(A) reduction(+:A_total)
      {
          tmp_a=0;
          #pragma acc loop
```

```
        for(i = 0; i < SIZE; i ++)
        {
            A[i]=i;
            tmp_a += A[i];
         }
         A_total += tmp_a;
    }

    //check
    if(A_total != (SIZE/2*(SIZE-1)+123))
    {
        printf("TEST FAILED, A_total = %d.\n", A_total);
        exit(-1);
    }
    printf("TEST OK.\n");

    return 0;
}
```

图 3-29 reduction 子句的例子 1

图 3-30 所示例子 test_reduction_002.c 测试在 loop 编译指示上使用 reduction 子句。

```
/*
*   文件名：test_reduction_002.c
*   目   的：测试在 loop 编译指示上使用 reduction 子句
*/

#include <stdio.h>
#include <stdlib.h>

#define SIZE       1024

long B[SIZE];

main()
{
    int i;
    long B_total = 124;

    #pragma acc parallel copyout(B)
    {
        #pragma acc loop reduction(+:B_total)
        for(i = 0; i < SIZE; i ++)
        {
            B[i]=i;
            B_total += B[i];
         }
    }

    //check
    if(B_total != (SIZE/2*(SIZE-1)+124))
    {
        printf("TEST FAILED, B_total = %d.\n", B_total);
        exit(-1);
    }
    printf("TEST OK.\n");

    return 0;
```

```
}
```

图 3-30 Reduction 子句的例子 2

图 3-31 所示例子 test_reduction_003.c 测试在 parallel 和 loop 编译指示上同时使用 reduction 子句。

```
/*
*   文件名：test_reduction_001.c
*   目   的：测试在 parallel 和 loop 编译指示上同时使用 reduction 子句
*/

#include <stdio.h>
#include <stdlib.h>

#define SIZE       1024

long A[SIZE], B[SIZE];

main()
{
      int i;
      long A_total = 123, B_total = 124, tmp_a;

      #pragma acc parallel private(tmp_a) copyout(A,B) reduction(+:A_total)
      {
          tmp_a=0;
          #pragma acc loop reduction(+:B_total)
          for(i = 0; i < SIZE; i ++)
          {
                A[i]=i;
                B[i]=i;
                tmp_a += A[i];
                B_total += B[i];
           }
           A_total += tmp_a;
      }

      //check
      if(A_total != (SIZE/2*(SIZE-1)+123))
      {
            printf("TEST FAILED, A_total = %d.\n", A_total);
            exit(-1);
      }
      if(B_total != (SIZE/2*(SIZE-1)+124))
      {
            printf("TEST FAILED, B_total = %d.\n", B_total);
            exit(-1);
      }
      printf("TEST OK.\n");

      return 0;
}
```

图 3-31 Reduction 子句的例子 3

3.11.3 数据拷贝子句 copy/copyin/copyout

对众核加速编程模型而言，内存一般分为主存（host memory）和设备内存（device memory）两部分。主进程的数据空间在主存分配，加速线程的数据空间则既可以在主存上分配，也可以在设备内存上分配。对于加速线程而言，主存容量大，但访问速度慢；设备内存容量小，但速度快。要保证加速线程高效运行，必须尽量保证其操作数据在设备内存里，因此就需要先把主存里的数据分段拷贝到设备内存，完成计算后再拷贝回主存，为此OpenACC*提供了数据拷贝的编译指示。

1) 语法

```
copy (varlist)
copyin (varlist)
copyout(varlist)
其中，varlist 为变量列表，变量之间以逗号分隔。
```

子句含义：

a) copy 子句：varlist 中的变量，在计算前，首先将它们的值从主存拷贝到设备内存；计算结束后，将新的值从设备内存拷回到主存；通常 varlist 中的变量在加速代码段中是先读后修改的；

b) copyin 子句：varlist 中的变量，仅仅在计算前将它们的值从主存拷贝到设备内存，在相关的计算结束后，并不把加速设备内存上的内容拷贝回主存；通常 varlist 中的变量在加速代码段中是只读的；

c) copyout 子句：varlist 中的变量，仅仅在计算结束后将它们的值从设备内存拷回到主存；通常 varlist 中的变量在加速代码段中是写覆盖的；

d) 在加速设备中，由于局存有限，copy 指定的数组可能会分块传输。如果数组太大且不能分块，则不能使用 copy 系列子句指定，可考虑使用 cache 子句。

2) 说明

默认情况下，copy/copyin/copyout 会根据循环的划分方式将对应的数据进行分割；如果用户用暗示语句指出数据需要全部拷入，copyin 会将对应的数据整个拷贝到加速设备内存上，一般用于较小的只读数据。

varlist是变量列表，变量之间以逗号分割，支持标量、数组、子数组，其中对于子数组的描述规则如下：

C/C++

```
arr[kk][*]   //kk 对应的最低维数据
arr[kk][ lower:upper:max_length]   //kk 对应的最低维 lower-upper 的数据
```

C/C++

Fortran

```
arr(*,kk)            ! kk 对应的最低维数据
arr(lower:upper:max_length, kk)   ! kk 对应的最低维 lower-upper 的数据
```

Fortran

其中，lower 和 upper 表示数组在该维度上的下标区间，lower 不能小于数组下标的最小值，upper 不能超过数组下标的最大值。Lower 和 upper 可以是整数表达式，不支持其他的情况。max_length 只能是常数，用于描述数组某一维区间跨度的最大值，该参数是可选的。

当数组某一维的区间长度可以确定时，不需要给出 max_length 参数。

当数组某一维的区间长度不能确定时，即 upper-lower+1 不是常数，通常将动

态申请所需要的数据空间，而动态申请释放空间的开销相比较大，因此为减少程序的性能损失，用户可以用 max_length 参数给出该区间跨度的最大值，编译系统将依据 max_length 的值分配静态空间。max_length 参数最多只能出现在数组的某一维上。

注：当 copy 作为 data 的子句时，varlist 的描述允许使用子数组的形式；当 copy 作为 parallel 的子句时，varlist 仅支持标量和数组，不支持子数组的形式。由于每个加速线程有独立的设备内存区域，因此 varlist 中的标量只能拷入设备内存，而不能更新回主存，否则存在多个副本，可能导致结果不一致，编译器会根据这个原则自动忽略对标量的拷出操作。

3) 示例

```
int A[N],B[N],C[N];
#pragma acc paralllel loop copyin(A,B) copyout(C)
for (i = 0; i < N; i++)
{
        C[i] = A[i]+B[i];
}
```

图 3-32 数据拷贝子句例子 1

在图 3-32 中所示例子 1 是 copy 编译指示的典型用法，A、B 数组只读，所以只需拷进，C 数组只写，所以只需拷出。如果数组所需空间超过局存限制，循环迭代将以任务分担的方式由多个加速线程执行，相应的数组 A、B 和 C 也会根据循环的划分方式进行适当的分割。例如执行 i=0 迭代的线程只需要 A[0]、B[0]和 C[0]三个元素的数据。

```
int x[N];
#pragma acc parallel loop copyin(x)
for (i = 0; i < Natoms; i++)
{
        xi = x[ i + 0 ];
        yi = x[ i + 1];
        zi = x[ i + 2];
}
```

图 3-33 数据拷贝子句例子 2

在图 3-33 中，在一次循环迭代计算中，需要访问 x[i]、x[i+1]、x[i+2]三个元素，在这种情况下，编译器会根据循环的分块情况自动拷贝所需的相关数据到加速设备内存中。

```
!$ACC PARALLEL LOOP
do i =1,100
        ICCG = id2pos1(i)
        indwfd=id2pos2(i)
        !$ACC DATA COPY(pvkd(*, ICCG, indwfd))
        do k=1,100
            cg=pvkd(k, ICCG, indwfd)
        enddo
        !$ACC END DATA
enddo
!$ACC END PARALLEL LOOP
```

图 3-34 data copy 指示示例 1

在图 3-34 的示例中，data copy 指示出拷贝数组 pvkd 中最高维为 indwfd、次高维为 ICCG 对应的最低维数组数据到设备内存中，并在 DATA 区域结束时拷回主存。

```
subroutine func(A_slice, n)
integer A_slice(n)
  integer i1,i2,lower,length,upper,block
  block=1024
```

```
    do i1=1,n/block
        lower=i1*block
        length=min(block,n-(i1+1)*block)
        upper=length
        !$ACC DATA COPY(A_slice(lower:length))
        do i2=1,upper
            A_slice(i1*block+i2)=….
        enddo
        !$ACC END DATA
    enddo
end subroutine

program    main
    integer A(8192,8192,8192)
    integer i,j,k1,k2
    !$ACC PARALLEL LOOP
    do k1=1,100
        do j=1,100
            call func(A(1:8192, j,k), 8192)
        enddo
    enddo
    !$ACC END PARALLEL LOOP
end program
```

图 3-35 data copy 指示示例 2

在图 3-35 的示例中，data copy 指示添加在 parallel 内被调用的函数内，用于对函数内用到的大数组进行分块处理。

3.11.4 异步子句 async

1) 语法

```
async [(整数表达式)]
```

2) 说明

async 子句可以作为 parallel、wait 等指示的子句使用，其含义表明相应编译指示的操作将以异步方式执行，即只启动对应的操作，而不用等待操作的完成。通常需要与 wait 指示配合使用，使用 wait 指示等待相应操作的完成。

async 的参数是非负整数标量，或者是 OpenACC*预定义的参数，当参数是非负整数标量时，其含义是异步操作队列的编号，即当前的操作将放入对应编号的异步操作队列中执行，相同编号的不同操作将按先后顺序执行，不同编号队列的执行相互独立。当不指定参数时，使用默认的异步操作队列。

3) 示例

C/C++

```
    #define NUM 1024
    int a[NUM];
    int b[NUM];
    int c[NUM];
    main()
    {
        int i;
        #pragma acc parallel loop async(0) copyout(a)
        for(i=0;i<NUM;i++)
        {
```

```
            a[i]=1;
        }
        #pragma acc parallel loop async(1) copyout(b)
        for(i=0;i<NUM;i++)
        {
            b[i]=1;
        }
        #pragma acc parallel loop wait(0,1) copyout(c) copyin(a,b) async(2)
        for(i=0;i<NUM;i++)
        {
            c[i]=a[i]+b[i];
        }
        #pragma acc wait(2)
    }
```

C/C++

Fortran

```
    program AA
    integer NUM
    parameter(NUM=1024)
    integer   a(NUM)
    integer b(NUM)
    integer c(NUM)
    integer i;
    !$ACC parallel loop async(0) copyout(a)
    do i=1,NUM
        a(i)=1
    enddo
    !$ACC end parallel loop
    !$ACC parallel loop async(1) copyout(b)
    do i=1,NUM
        b(i)=1
    enddo
    !$ACC end parallel loop
    !$ACC parallel loop wait(0,1) copyout(c) copyin(a,b) async(2)
    do i=1,NUM
        c(i)=a(i)+b(i)
    enddo
    !$ACC end parallel loop
    !$ACC   wait(2)
    end program
```

Fortran

图 3-36 async 子句示例

3.11.5 暗示子句 annotate

暗示子句的语法格式如下：

```
annotate ([暗示信息][; 暗示信息])
```

其中暗示信息可以是下列之一：

tilemask(array1,array2,…)/ **tilemask**(dim1:array1,dim2:array2,…)；

entire(array1, array2, array3,….)

readonly(varlist)；

dimension(array1name(dim1,dim2,..),array2name(dim1,dim2,…),…)

co_compute
margin_compute
slice(array1name(access_mode_info), …)

暗示子句用于对 OpenACC*其他指示或子句进行辅助说明，暗示子句能否发生作用由编译器根据实际情况进行分析和判断决定。

3.11.5.1 tilemask

1) 语法

```
annotate (tilemask(array1,array2,…))
annotate (tilemask(循环迭代变量 i:数组 1, 循环迭代变量 j:数组 2,…))
```

2) 说明

屏蔽循环分割对指定数组的影响，使相关数组相关维的数据不被分割，提升数据传输效率。tilemask 暗示子句有两种用法：

用法 1：tilemask 暗示子句的参数是数组名列表，一般用于有循环分割的 loop 指示。在程序中，一个循环往往是对数组的某一维度进行迭代，数组的这一维是该循环的对应维度。tilemask 用于指出列表中的数组不受该循环分块的影响，对应维度的数据能够在该循环分割之前全部拷入设备存储器，从而能够在对应的循环外拷贝相关数据，减少 DMA 次数。

用法 2：tilemask(循环迭代变量 i:数组 1, 循环迭代变量 j:数组 2,…)，作用于 loop 指示内直接嵌套的所有循环，通过循环迭代变量的方式指定受影响的循环，并屏蔽对应循环的分割对相关数组的影响，这种用法通常与 tile(循环迭代变量 i:块大小 1,循环迭代变量 j:块大小 2, …)配合使用。

3) 示例

```
#include<stdio.h>
int A[64][128], B[64][128], C[64][128];

int main()
{
   int i, j;
   for(i=0;i<64;i++)
      for(j=0;j<128;j++)
      {
         A[i][j]=i+j;
         B[i][j]=i*j;
      }
      #pragma acc parallel copyin(A, B) copyout(C)
      #pragma acc loop
      for(i = 0; i < 64; i++)
      {
            #pragma acc loop tile(16) annotate(tilemask(A, B))
            for(j = 0; j < 128; j++)
            {
                  C[i][j] = A[i][j] + B[i][j];
            }
      }
   for(i=0;i<64;i++)
      for(j=0;j<128;j++)
      {
         if(C[i][j]!=(i+j+i*j))
         {
```

```
            printf(“Test Fail!”);
            exit(0);
        }
    }
     printf(“Test OK!”);
}
```

图 3-37 annotate 子句示例-tilemask

tilemask 暗示子句用法 1 的示例如图 3-37 所示，tilemask 指示用在内层 loop 指示上，参数为数组 A，B。虽然内层循环的分块大小为 16，但数组的 A，B 的最低维不受此分块影响， 最低维的数据将在内层循环之外全部拷入设备内存。而数组 C 的最低维则被相应的划为大小为 16 的块，每次拷入 1 块，分多次拷入。

```
#include<stdio.h>
int A[64][128], B[64][128], C[64][128];

int main()
{
    int i, j;
    for(i=0;i<64;i++)
      for(j=0;j<128;j++)
      {
          A[i][j]=i+j;
          B[i][j]=i*j;
      }
    #pragma acc parallel copyin(A, B) copyout(C)
    #pragma acc loop tile(i:1, j:16) annotate(tilemask(j:A, j:B))
    for(i = 0; i < 64; i++)
    {
        for(j = 0; j < 128; j++)
        {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
    for(i=0;i<64;i++)
      for(j=0;j<128;j++)
      {
        if(C[i][j]!=(i+j+i*j))
        {
            printf(“Test Fail!”);
            exit(0);
        }
      }
     printf(“Test OK!”);
}
```

图 3-38 annotate 子句示例-tilemask

tilemask 暗示子句用法 2 的示例如图 3-38 所示，其效果与图 3-37 的程序相同。

3.11.5.2 entire

1) 语法

```
annotate (entire(array1, array2, array3,….))
```

2) 说明

entile 暗示的参数是数组名列表，一般用于 parallel 指示的子句，指出列表中的数组要全部拷入每个线程对应的设备内存。

3) 示例

```
int A[64], B[128][64];

int main()
{
    int i, j;

    #pragma acc parallel copyin(A) copyout(B) annotate(entire(A))
    {
        #pragma acc loop
        for(i = 0; i < 128; i++)
        {
            for(j = 0; j < 64; j++)
                B[i][j] = A[j] + i;
        }
    }
}
```

图 3-39 entire 示例

在上图的示例中，数组 A 的全部数据在每个线程对应的设备存储器中都有一份完整的拷贝，数组 B 则是根据并行循环划分的方式，分块分布存储在每个线程的设备存储器中。

3.11.5.3 readonly

1) 语法

```
annotate (readonly(varlist))
```

2) 说明

readonly 的参数是变量名列表，一般用于 parallel 指示的子句，指出加速计算区范围内的只读变量信息。

3) 示例

```
int A[64][128];

int main()
{
    int i, x;

    x = rand()%64;
    #pragma acc parallel loop copyout(A) annotate(readonly(x))
    for(i = 0; i < 128; i++)
        A[x][i] = i;
}
```

图 3-40 readonly 示例

在上图的示例中，readonly 暗示用于辅助说明数组下标访问。在加速计算区内，数组 A 的最高维下标访问 x 为只读变量，编译器在设备存储器上为数组 A 申请空间时，最高维的大小为 1。否则，设备存储器上数组 A 的最高维大小为 64。

3.11.5.4 dimension

1) 语法

```
annotate (dimension(array1name(dim1,dim2,..),array2name(dim1,dim2,…),…))
```

2) 说明

当指针(或动态数组)在加速计算区中以数组的形式使用时，编译器往往无法判断出该指针(或动态数组)所指向数组的维度信息，进而无法在设备存储器中申请适当大小的空间并拷贝数据。dimension 暗示一般用于 parallel 指示的子句，指出指针(或动态数组)

所指向数组空间的维度信息，从而帮助编译器在加速计算区中申请设备空间并并拷贝数据。

dimension 暗示中所指出的数组维度信息，Fortran 数组的维度从低维到高维说明，C 数组的维度从高维到低维说明，中间用逗号隔开。

3) 示例

```
int A[64][128];
int **B;

int main()
{
    int i, j;
    int *p1, **p2;

    p1 = (int *)malloc(sizeof(A));
    p2 = (int **)malloc(64*sizeof(int *));
    for(i = 0; i < 64; i ++)
    {
        *(p2+i) =   (int *)(p1+i*128);
    }
    B = p2;

    for(i = 0; i < 64; i++)
        for(j = 0; j < 128; j++)
            A[i][j] = i+j;

    #pragma acc parallel copyin(A) copyout(B)
              annotate(dimension(B(64, 128)))
    {
    #pragma acc loop
    for(i = 0; i < 64; i++)
        for(j = 0; j < 128; j++)
            B[i][j] = A[i][j];
    }
}
```

图 3-41 dimension 示例 1

在上图的示例中，dimension 暗示描述了指针 B 所指向空间的维度信息。

```
program main
implicit none
integer :: i, j
integer , allocatable :: b(:, :)
integer :: a(128, 64)

allocate(b(128, 64))

do i = 1, 64
  do j = 1, 128
      a(j, i) = i+j
  end do
end do

!$ACC PARALLEL COPYIN(a) COPYOUT(b)
      ANNOTATE(DIMENSION(b(128, 64)))
!$ACC LOOP
do i = 1, 64
  do j = 1, 128
```

```
                b(j, i) = a(j, i)
            end do
        end do
        !$ACC END LOOP
        !$ACC END PARALLEL

        end program
```

图 3-42 dimension 示例 2

在上图示例中，dimension 暗示描述动态数组 b 的维度信息。

3.11.5.5 co_compute

1) 语法

```
annotate (co_compute)
```

2) 说明

co_compute 暗示一般用于 loop 指示的子句，指定主线程按照一个加速线程的模式来进行计算，主线程和加速线程一起参与并行循环迭代任务的分担。

3) 示例

```
int A[130];

int main()
{
    int i;

    #pragma acc parallel copyout(A)
    {
    #pragma acc loop annotate(co_compute)
        for(i = 0; i < 130; i++)
        {
            A[i] = i;
        }
    }
}
```

图 3-43 co_compute 示例

在上图的示例中，并行循环由主线程和加速线程共同分担。若有 64 个加速线程，则主线程承担 i=65, i=129 次迭代。

3.11.5.6 margin_compute

1) 语法

```
annotate (margin_compute)
```

2) 说明

margin_compute 暗示一般用于 loop 指示的子句。并行循环的迭代次数被拆分为两部分，一部分是能够被加速线程数整除的规整部分，另一部分是被整除后的余量部分。margin_compute 暗示指定规整部分由加速线程执行，余量部分由主线程执行。

3) 示例

```
int A[132];

int main()
{
    int i;

    #pragma acc parallel copyout(A)
```

```
        {
        #pragma acc loop annotate(margin_compute)
            for(i = 0; i < 132; i++)
            {
                A[i]    = i;
            }
        }
    }
```

图 3-44 margin_compute 示例

在上图的示例中，第 0~127 次迭代由加速线程执行，第 128~131 次迭代由主线程执行。

3.11.5.7 slice

1) 语法

```
annotate (slice(array1name(access_mode_info), …))
```

2) 说明

slice 暗示的参数列表是数组名及其访问模式的描述，一般用于 parallel 编译指示的子句，是对 parallel 上数据 copy 子句的补充。slice 暗示用于说明需要拷贝的数组在加速计算区中的访问模式。特别地，如果数组是分段的区间访问时，可以将数组每段的访问模式都列在参数列表中，并与 parallel 指示的 copy 子句配合使用，来拷贝数组一个片段或多个片段的数据，提升数据传输的效率和降低对高速局存的占用，其数组访问模式的格式描述如下：

C/C++

```
arr[kk][*]                  //kk 对应的最低维数据
arr[kk][ lower:upper]   //kk 对应的最低维 lower-upper 的数据
```

C/C++

Fortran

```
arr(*,kk)           ! kk 对应的最低维数据
arr(lower:upper, kk)    ! kk 对应的最低维 lower-upper 的数据
```

Fortran

可以看出 slice 的格式与 copy 子句中的子数组描述类似，但是二者的含义和用法有所区别。slice 是用于描述加速计算区内数组的访问模式，而没有数据拷贝的动作。*、kk、lower:upper 属于 access_mode_info 的三种描述方式，*表示该维所有数据均需要访问，kk、lower、upper 中可以是常数、变量或变量表达式，通常情况下变量或变量表达式应在程序中的对应数组访问中实际使用，变量表达式的格式支持 i、i+n、i-n，x，其中 n 为常数，x 为整数变量。

使用时需要注意的包括：

a) 当同一个数组出现多种访问模式时，**需要保证每种访问模式是静态可区分的**，例如，不能出现 A[i-1:i+1][*]，A[20:30][*]，因为无法区分程序中的 A[i][j]属于哪一种模式，这种情况可能会导致无法预知的错误。

b) 当多种访问模式之间有数据重叠时，**需要用户保证程序运行过程中的数据一致性**，这种情况下通常用于只读数据。

c) slice 用于指导编译系统按照访问模式将数据切分成小的数据片段，通常在希望压缩高速局存使用量时采用。需要注意的是当大块数据切分成多个数据片段时，需要多次相对小数据量的数据传输，其数据传输开销比整块数据传输要高。因此**在实际使用时需要在局存的空间占用量和数据传输开销之间权衡**，在局存

空间允许的情况下，尽量使用连续的整块数据传输。

3) 示例

```
int A[100][100][32], B[100][100][32];
int tmp1=0, tmp2=0;

int main()
{
#pragma acc parallel loop tile(1,1) copyout(B)
        copyin(A) annotate(slice(A[i-5:i+5][j][*], A[i][j-5:j+5][*]))
for(i=5;i<95;i++)
   for(j=5;j<95;j++)
      for(k=0;k<32;k++)
      {
       tmp1=tmp1 + A[i][j][k] +
            A[i][j-1][k]+ A[i][j+1][k] +
            A[i][j-2][k]+ A[i][j+2][k] +
            A[i][j-3][k]+ A[i][j+3][k] +
            A[i][j-4][k]+ A[i][j+4][k] +
            A[i][j-5][k]+ A[i][j+5][k] ;

       tmp2=tmp2 + A[i][j][k] +
            A[i-1][j][k]+ A[i+1][j][k] +
            A[i-2][j][k]+ A[i+2][j][k] +
            A[i-3][j][k]+ A[i+3][j][k] +
            A[i-4][j][k]+ A[i+4][j][k] +
            A[i-5][j][k]+ A[i+5][j][k] ;

            B[i][j][k]=tmp1+tmp2;
      }
}
```

图 3-45 slice 暗示示例

图 3-45 的例子中 A 数组的访问模式较为复杂，当 i，j 固定时(k 维不划分)，所需要访问的数据区间为 A[i-5:i+5][j-5:j+5][*]，则数据量为 11*11*32*sizeof(int)=15488B=15KB (32 是最低维的长度)，当设备内存紧张时，这么多数据可能无法全部存放，但是通过分析程序可以发现，程序中实际用到的数据并没有这么多，可以进一步分拆成 A[i-5:i+5][j][*], A[i][j-5:j+5][*]，如果按这两种模式来细分所需访问的数据，则只需要 11*1*32*sizeof(int) + 1*11*32*sizeof(int)=2816B，会大大减少高速局存占用量。

3.11.6 组合子句

1) 语法

```
private(varlist) copyin(varlist)
private(varlist) copyout(varlist)
private(varlist) copy(varlist)
private(varlist) cache(varlist)
```

2) 说明

该子句为 private 和 copy 子句的组合，首先对 varlist 中指定的变量先进行私有化处理（放在主存的私有空间），再进行 copyin、copyout、copy 或 cache 操作。

其中 private 子句可以换成 firstprivate 子句，在这种情况下，varlist 中指定的变量先

进行私有化处理，并用原始数据初始化每个新的私有化变量。

3) 示例

```
int A[8192];

int main()
{
    int i, k;

    #pragma acc parallel private(A) copy(A)
    {
    #pragma acc loop
    for(k = 0; k < 64; k++)
    {
    #pragma acc loop tile(1024)
        for(i = 0; i < 8192; i++)
        {
            A[i] = i*10;
        }
      #pragma acc loop tile(1024)
        for(i = 0; i < 8192; i++)
        {
            A[i] = A[i] + k;
        }
    }
    }
}
```

图 3-46 private 和 copy 组合子句示例

在上图的示例中，OpenACC*编译器对循环变量 k 进行并行划分。在内层循环中，每个加速线程都需要读写数组 A，然而数组 A 的大小无法全部拷入设备内存。因此，先把数组 A 私有化，再对内层循环进行分块。每个加速线程从自己的私有主存区每次拷贝数组 A 的一块到设备内存。

```
program main
implicit none
integer :: i, k, m
integer :: a(8192)
common a

do i = 1, 8192
  a(i) = i
end do

!$ACC PARALLEL LOOP FIRSTPRIVATE(a) CACHE(a:8)
do k = 1, 64
  do i = 1, 8192
      m = mod(2*i+k, 8192)
      a(m) = a(m)+k+i
  end do
end do
!$ACC END PARALLEL LOOP
end program
```

图 3-47 private 和 cache 组合子句示例

在上图的示例中，每个加速线程都要读写数组 A 的全部数据，而数组 A 不能全部拷入设备内存。程序中对数组 A 的访问是离散的，所以将 A 私有化，并采用 cache 的方式提高数组 A 的访问效率。

4 运行时库接口

OpenACC*语言环境支持 OpenACC2.0 中定义的所有运行时库接口，但是由于不恰当地使用运行时库接口可能会对程序的可移植性造成影响，因此不建议过多的使用运行时库接口。所有的运行时接口的功能均有相应的编译指示功能对应。

5 使用及编程指南

5.1 使用 OpenACC*进行应用移植和开发的步骤

使用 OpenACC*进行应用移植通常遵循如下步骤：

1) 调试通过多核程序（串行或消息并行）；
2) 找到可并行的核心循环，通常通过以下几个方面来确定：
   a) 核心代码必须为串行代码，由于加速设备内通常没有消息通信机制，在核心代码中不能包含有使用消息通信功能的代码；
   b) 可以使用类似 gprof 等性能工具确定热点函数并确定其中的核心代码；
   c) 支持在核心代码中的函数调用，但通常情况下被调用函数最好是纯函数，或者使用 routine 指示标注被调用的函数。
3) 在核心循环前加上“#pragma acc parallel loop”编译指示；
4) 确定核心代码中需要私有化的变量，并添加 private 子句；
   a) 在私有化变量分析方面，用户可以使用-priv 选项编译，调用 SWACC 编译系统私有化分析功能，SWACC 编译系统会进行私有化变量分析，并将显示分析结果（由于编译器分析存在局限性，需要对分析结果进行确认，以免出现遗漏或错误的情况）；
   b) 在数组访问分析方面，用户可以额外添加-arrayAnalyse 选项使用 SWACC 编译系统的分析功能进行分析。
5) 编译并运行，如果私有化变量添加正确，运行结果是正确的。

至此，程序移植初步完成，简单的说主要是三个步骤：1)找到核心循环；2)添加 parallel loop 指示，并确定 private 变量；3)编译运行，如果不正确检查第 2)步有无遗漏，直到程序可以正确的在加速核心上运行。

后续步骤是对程序的性能进行优化。程序在加速设备上的运行性能与下面一些因素相关：

(1) 是否充分利用加速设备上的高速设备内存；（关键数据尽量放在 LDM）
(2) 数据传输和计算比例；（计算访存比，通常与程序的算法和实现有关）
(3) 合理设置循环分块大小；（灵活使用 tile，将关键数据尽可能多的放在 LDM，循环分块会影响放入 LDM 中的数据量）
(4) 设法提高数据传输效率。（使用打包子句将分散的数据整合成大块的数据、使用转置子句将不连续的数据访问变成连续的数据访问）

在程序移植优化方面，需要分析核心循环内的数据访问情况，适当的添加 copy、local、pack 等指示，将核心数据导入设备内存中进行访问和计算，OpenACC*中程序移植过程中常

见用法与编译指示用法如表 5-1 所示。

表 5-1 OpenACC*常见用法与编译指示/子句对照表

序号	功能	编译指示子句用法	说明
1	在 LDM 中为临时变量(标量、小数组)申请空间	local(变量列表)，用法及示例参见 3.4.4。	在加速代码中需要被改写后再使用，并且其数值不需要更新回主存中的变量，包括数组和标量，可以使用 local 子句在 LDM 中创建相应的存储空间，local 中的变量没有初始值。
2	只读的标量拷贝到 LDM 中	copyin(变量列表)，用法及示例参见 3.4.6。	copyin 的标量将在 LDM 中申请空间，并将原始变量的值赋给 LDM 中对应的变量。
3	只读小数组整个拷贝到 LDM	copyin(变量列表) annotate(entire(变量列表))，用法及示例参见 3.4.13。	copyin 的数组默认会进行分块拷入，当有小数组需要整个拷入 LDM 时，采用 entire 暗示的方式通知编译器。此时对应的数组将在每个加速线程的 LDM 中申请空间，并将原始数组的数值拷贝到 LDM 中。
4	数组拷入拷出 LDM	copy/copyin/copyout，用法及示例参见 3.4.6。	按照循环的划分方式将数组分割，以数据片段的方式在 LDM 中申请空间，并在主存和 LDM 间进行数据传输。如果数组与被划分的循环非直接线性仿射则无法对数据进行分割。
5	最外层循环量较小，并行度不够	collapse 子句，用法及示例参见 3.5.4。	collapse 的作用是将紧嵌套的若干循环合并进行并行划分，适用于最外层循环量较小，而且紧嵌套若干层可并行的循环的情况。
6	LDM 空间没用满	增加 tile 子句，或者增大 tile 的分块大小，用法及示例参见 3.5.5。	循环的 tile 大小会直接影响到与之关联的数组的分块情况，增大 tile 的块大小可以将每次拷入拷出 LDM 的数据增大。
7	parallel copy 的多维数组太大，LDM 存放不下	对数组访问相关的内层循环前添加 loop 指示，并根据需要使用 tile 子句，用法及示例参见 3.5、3.5.5。	当仅对最外层循环划分时，某些多维数组的数据量过大，无法一次性放入 LDM，则可以在与数组访问相关的内层循环前添加 loop 指示，用于将对应的循环分块处理，相应的会对相关的数据也分块，则可以将大数据划分成小的数据块。
8	数组访问与并行循环无关，与内层循环相关，使用 parallel copy 无法	loop 指示、tilemask 暗示子句、tile 子句，用法及示例参见 3.5、3.11.5.1、	当数组访问与最外层并行循环无关、仅与内层循环相关时，使用 parallel copy 编译器无法确定数组

	拷入 LDM	3.5.5。	如何分块存储，整个数组通常无法整个放到 LDM 中。则可以在与数组访问相关的内层循环前添加 loop 指示，用于将对应的循环分块，并根据需要设置 tile 子句。另外，在某些情况下，又希望对某些数组屏蔽新增加的 loop tile 信息，可以使用 tilemask 子句，提升某些数组的数据传输效率。
9	数组访问的数据区间是在运行时确定的	data copy ，用法及示例参见 1)。	这种情况下需要动态的数据传输，可以使用 data copy 在需要数据的代码前描述所需的数据信息，并可以根据需要使用 if 子句控制数据的重用。
10	离散访问的数组无法放入 LDM	cache 子句，用法及示例参见 3.4.5。	对于离散访问的数组，编译器无法知道拷贝哪些数据到 LDM 中，因此 copy 子句不适用这种情况，可以使用 cache 子句解决这个问题，另外根据程序特点可以使用 readonly 暗示子句对 cache 进行优化，只读的 cache 数据不用刷新会主存。
11	加速函数内需要用到 parallel copy 的数据	data present，用法及示例参见 3.6.3。	当 parallel copy 的数据在 parallel 内所调用的函数中需要使用时，需要实现在 parallel 中拷贝到 LDM 中的数据在加速函数内的重用和映射，可以用 data present 描述重用的数据信息。
12	具有多个访问模式相同的标量或数组，想进一步提升数据传输效率	pack/packin/packout 子句，用法及示例参见 3.4.7。	数据打包子句的作用是将具有相同数据访问模式的变量打包成新的数据结构，可以实现一次数据传输拷贝多个数据对象的目的。另外，当加速段本身在循环内、且打包的数据在该循环内不改变时，可以使用 data index 设置数据点，将打包操作外提到循环之外。
13	优化跨步数组访问的性能	swap/swapin/swapout 子句，用法及示例参见 3.4.8。	当二维数组是按列访问时，其访问属于跨步访问，高维数组以此类推。这种情况下访存及数据传输效率较低，可以使用转置子句对原始数据重新布局改善数据传输效率。另外，当加速段本身在循环内、且被转置的数据在该循环内不改变时，可以使用 data index 设置数据

			点，将转置操作外提到循环之外。
14	加速代码中有函数调用	routine、data copy、data present 指示，用法及示例参见 3.10、3.6。	加速代码中有函数调用的情况有三种处理方式：1)在被调用函数的定义处使用 routine 指示，表明该函数的定义将生成加速版本，同时可以根据情况使用 data present 和 data copy 来控制函数内的数据重用和传输；2)将函数内联；3)将函数改造成纯函数，即函数所需的数据均通过参数传递。
15	临时数组太大，无法放入 LDM	组合子句，用法及示例参见 3.11.6。	当程序中需要私有化的临时数组很大，使用 local 无法将该数组存放到 LDM 时，可以使用组合子句，将该数组先 private，保证其私有属性，之后根据情况使用 copy 或者 cache 将其数据分段导入 LDM，保证程序性能。

5.2程序移植示例

使用矩阵乘程序来说明使用 OpenACC*进行程序移植和优化的过程，程序代码如图 5-1 所示。

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <sys/time.h>
 4
 5 #define M 1024
 6 #define N 1024
 7 #define K 1024
 8
 9 int A[M][N];
10 int B[N][K];
11 int C[M][K];
12
13 // 计时函数
14 double
15 timer()
16 {
17         double t;
18         struct timeval tv;
19         gettimeofday(&tv, NULL);
20         t = (double)tv.tv_sec*1.0+(double)tv.tv_usec*1e-6;
21         return t;
22 }
23
24 int
25 main()
26 {
27         int i,j,k;
28         double t1, t2;
```

```
29      // init A
30      for(i = 0; i < M; i ++)
31          for(j = 0; j < N; j ++)
32              A[i][j] = i;
33      // init B
34      for(i = 0; i < N; i ++)
35          for(j = 0; j < K; j ++)
36              B[i][j] = j;
37
38      t1 = timer();
39      // matrix multiply, C=A*B
40
41      for(i = 0; i < M; i ++)
42      {
43
44          for(k = 0; k < K; k ++)
45          {
46
47              for(j = 0; j < N; j ++)
48              {
49                  C[i][k] +=    A[i][j] * B[j][k];
50              }
51          }
52      }
53      t2 = timer();
54      printf(“Matrix-Multiply A[%d][%d] * B[%d][%d] , use time:%.2f\n”,
55                                  M, N, N, K, t2 – t1);
56}
```

图 5-1 矩阵乘串行程序代码

图中例子的核心段是 41~52 行的循环段，即进行矩阵乘运算的代码段。另外，为了比较移植和优化前后的程序性能，在核心代码段前后增加了计时代码，将图中的例子保存为 matrixMul.c。

1) 移植

(1) 首先使用 swacc 调试该串行程序，并记录串行版本的运行时间。

- swacc matrixMul.c –O3；
- 提交到运算节点以单进程方式执行，运行时间：**T_Serial = 4.10 秒**（注意：运行时间与课题数据规模、运行环境、编译环境等多个因素有关，仅供参考）。

(2) 找到核心循环，并在核心循环前添加#pragma acc parallel loop 指示，保存。

- 在 matrixMul.c 的第 40 行添加**#pragma acc parallel loop** 指示。

(3) 使用”swacc –priv”选项编译程序，根据私有变量自动分析结果或通过手工分析，添加 private 子句。所谓需要私有化的变量就是每个加速线程都可能会去改写，并且改写的内存区域有重叠的变量。具体到 matrixMul.c，需要私有化的变量有三个：i、j、k。

- 在 matrixMul.c 第 40 行的编译指示中添加私有化子句：#pragma acc parallel loop **private(i, j, k)**

(4) 保存，使用 swacc 编译，并使用 1 个主核加 64 个从核的规模运行，此时程序已经可以正确的在申威 26010 上运行。

2) 优化

优化的基本思想是将程序中的关键数据尽可能多的放到 LDM 中，并设法提高程序中数据的传输效率。

(1) **将标量、小数组放入 LDM**：对于这个程序只需要将 private 子句换成 local 子句，即将 i、j、k 三个标量放在每个加速线程的 LDM 中，其性质仍然是加速线程私有的。

- 第 40 行编译指示变为：#pragma acc parallel loop **local**(i, j, k)
- 编译运行，验证正确性和效果。

(2) **将关键数组拷贝到 LDM**：程序中访问的数组有三个，A、B、C，其中 A 和 B 是只读的，C 是先读后写的，对于只读的数据只需要拷入(copyin)，只写的数据只需要拷出(copyout)，否则既需要拷入也需要拷出(copy)。在程序中第 40 行添加数据拷贝子句。

- 第 40 行编译指示变为：#pragma acc parallel loop local(i, j, k) **copyin(A, B) copy(C)**
- 编译运行，验证正确性和效果，使用同样规模运行时间约 **T_Copy = 9.02 秒**。

可以看到，程序的运行不仅没有变快，反而变慢了。注意一下编译时的屏幕输出或者使用-keep 选项查看变换后的_slave 中间文件，可以发现在加速代码中对 B 数组的访问仍然是直接访问主存数据，并没有使用 LDM 空间。

分析程序可以发现，B 数组的访问与并行循环 i 是无关的，也就是说加速线程每执行一次并行循环迭代 i 都需要访问 B 数组所有的数据，由于 LDM 容量有限，B 数组无法整个拷入加速线程的 LDM，因此对 B 数组的访问被处理成直接访问主存数据。

而 A、C 数组的访问是跟并行循环 i 线性相关的，也就是说跟随着并行循环 i 的划分，每个加速线程所需要读取的 A 数组的数据和需要改写的 C 数组的数据也是不同的，因此，对于 A、C 数组，可以仅将所需的数据放入对应加速线程的 LDM。

(3) **解决 B 数组拷贝到 LDM 的问题**：数组 B 之所以没有放入 LDM 中，是因为数组 B 相关的循环没有划分，导致数据太大没法放入 LDM，解决数据过大无法放入 LDM 的办法是对没有进行划分的循环使用 tile 进行分块，使得循环分块后对应的数据能够拷入 LDM，对于这个程序具体实现上有两种方式：

(a) 方式 1：对 k 循环 tile，并对数组 B 转置拷入(swapin)

B 数组访问相关的循环变量是 k 和 j，首先可以选择对 k 循环 tile，其用意是将循环分块执行，同时可以使相关的数据按照同样的分块方式拷贝到 LDM 中。具体做法是：

- 在第 43 行，k 循环前添加**#pragma acc loop tile(2) annotate(tilemask(C))**
  - 最外层 i 循环的 loop 指示表明 i 循环将在多个加速线程间并行划分
  - 而内存的 k 循环上的 loop 指示表明 k 循环将在同一个加速线程内分块执行，此处 tile 指定分块大小为 2。
  - 由于数组 C 的访问也与 k 相关，对 k 循环分块之后，会影响到 C 数组的数据传输，添加 annotate(tilemask(C))的作用是，通知编译器 C 数组忽略 k 循环的分块信息，即 C 数组与 k 相关的维度（最低维）不分块，整个拷入 LDM，这样可以获得高效的数据传输。
- 修改第 40 行的编译指示，将 copyin(B)改成 swapin(B(dimension order:1, 2))
  - 第 40 行编译指示变成**#pragma acc loop local(i,j,k) copyin(A) copy(C) swapin(B(dimension order:1, 2))**
  - 之所以使用 swapin 代替 copyin 是因为与数组 B 的访问相关的下标时 j 和 k，分别对应的是 B 数组的高维和低维，而 j 循环在 k 循环之内，也就是说 B 数组的访问是不连续的。不连续的数据访问会导致数据传输效率较

低，而 swapin 的作用是对 B 数组的原始数据按照指定的维度顺序进行转置，使用转置后的数据代替原始 B 数组的访问，这样可以使不连续的数据访问变成连续的数据访问。

- 上述两个修改完成后，保存，编译运行，此时的矩阵乘时间为 **T_Tile_K = 0.45 秒**。

(b) 方式 2：对 j 循环 tile

方式 1 是对 k 循环 tile，这个程序还可以对 B 数组的高维分块，即对应的将 j 循环分块执行。

- 在第 46 行，j 循环前添加**#pragma acc loop tile(2) annotate(tilemask(A))**
  - 与方式 1 的含义相同，该编译指示表明将对 j 循环以 2 为块大小进行分块，相应的会影响到与 j 相关的数据拷贝，包括数组 A 和 B，同时通过 tilemask(A)的暗示子句通知编译器，A 数组忽略该分块信息，即 A 数组与 j 相关的维度不分块。
- 保存，编译运行，该版本矩阵乘的时间为 **T_Tile_J = 0.53 秒**。
- 方式 2 的运行时间之所以比方式 1 略长，是因为在方式 2 中数组 B 的访问方式仍然是先高维后低维，这对程序的性能有一定影响。
- 因此，可以尝试修改原始程序，调整 j、k 循环的顺序，之后再按照本节介绍的方法进行移植和优化。

至此，该程序基本上移植和优化完毕，优化后的核心段可以获得 9 倍左右的性能加速，后续还可以基于 SWACC 编译器生成的中间代码，进行二次开发，对核心段中的循环采用 SIMD 进行优化，二次开发的方法见后续章节，SIMD 的使用方式请参考相关用户手册。

5.3基于 OpenACC*的二次开发

在使用 SWACC 调试通过加速程序之后，如果需要在编译器生成的中间代码的基础上进一步分析和优化改造，则可以按下列步骤操作。

1) 中间文件的建立

以 test.c 程序为例，在该文件目录下输入指令：

```
swacc test.c -dumpcommand mk
```

其中-dumpcommand 为 2.2 节介绍的 SWACC 编译选项，其作用是保留中间文件，并将对中间文件的编译命令写到 mk 文件中。

命令行执行后，将在当前目录下生成 host 程序文件和 device 程序文件，同时编译 host 和 device 程序文件的命令将被写入指定的用户自定义文件(例子中是 mk)中。host 和 device 程序文件的命名规则如下（以 c 程序为例，Fortran 类似）：

a) host 程序文件：源文件名_host.c，如 test_host.c；
b) device 程序文件：源文件名_slave_行号.c，其中行号为加速区指示所在的行号，如有多个 parallel loop 指示，则会分别生成不同的 device 程序文件，如 test_slave_7.c。

2) 中间文件的分析改造

用户可根据需要对中间文件（host 程序文件、device 程序文件和编译命令文件）进行修改。

3) 中间文件的重新编译

执行之前保留的中间文件编译过程，执行 sh 编译命令文件名（例子中为 sh mk）即可。

6 版权声明