OpenACC 应用编程接口

2.0 版，2013 年 6 月

小小河 译

法律条款

1. 对本文档的任何使用都被视为完全理解并接受本文档所列举的所有法律条款。
2. 本文档的所有权利归作者所有，作者保留所有权利。
3. 未经作者书面同意，禁止任何形式的商业使用。商业使用形式包括但不限于存储、出版、复制、传播、展示、引用、编辑。
4. 本文档允许以学术研究、技术交流为目的使用。复制、传播过程中不得对本文档作任何增减编辑，引用时需注明出处。
5. 实施任何侵权形为的法人或自然人都必须向作者支付赔偿金，赔偿金计算方法为：

   赔偿金 = 涉案人次×涉案时长（天）×涉案文档页数×受众人次×100 元人民币，

   涉案人次、涉案时长、涉案文档页数、受众人次小于 1 时，按 1 计算。
6. 对举报侵权行为、提供有价值证据的自然人或法人，作者承诺奖励案件实际赔偿金的 50%.
7. 涉及本文档的法律纠纷由作者所在地法院裁决。
8. 本文档所列举法律条款的最终解释权归作者所有。

译者序

2007 年以来，以 nVidia GPU 为代表的加速器并行计算风起云涌，带有加速器的超级计算机在 TOP500 中的份额逐年增加，支持加速器的主流应用软件也呈爆炸式增长，研究加速器计算的技术人员数以百万计，世界范围内的大学、研究机构竞相开设相关课程。

目前有三家厂商提供加速器产品：nVidia GPU、AMD GPU、Intel 至强 Phi 协处理器。三种加速器使用的编程语言分别为 CUDA C/CUDA Fortran、OpenCL 和 MIC 导语。加速器计算有四个技术困难：一是 CUDA/OpenCL 等低级语言编程难度大，且需要深入了解加速器的硬件结构。而大部分的用户不是专业编程人员，学习一门新的编程技术将耗费大量时间、人力、财力。二是加速器的计算模型与 CPU 差别很大，移植旧程序需要几乎完全重写。大量的旧程序在性能优化上已经千锤百炼，稳定性上也久经考验，完全重写是不可完成的任务。三是低级编程语言开发的程序与硬件结构密切相关，硬件升级时必须升级软件，否则将损失性能。而硬件每隔两三年就升级一次，频繁的软件升级将给用户带来巨大负担。四是投资方向难以选择。三种加速器均有自己独特的编程语言，且互不兼容。用户在投资建设硬件平台、选择软件开发语言时就会陷入困境，不知三种设备中哪个会在将来胜出。如果所选加速器将来落败，将会带来巨大损失；而犹豫不决又将错过技术变革的历史机遇。

OpenACC 可以克服这四个困难。OpenACC 应用编程接口的机制是，编译器根据作者的意图自动产生低级语言代码。程序员只需要在原程序中添加少量导语，无须学习新的编程语言和加速器硬件知识，可以迅速掌握。只在原始程序上添加少量导语，不破坏原代码，开发速度快，既可并行执行又可恢复串行执行。在硬件更新时，重新编译一次代码即可，不必手工修改代码。OpenACC 标准制定时就考虑了目前及将来的多种加速器产品，同一份代码可以在多种加速器设备上编译、运行，零代价切换硬件平台。

本文档将 directive 翻译为*导语*，将 clause 翻译为*子语*，将 construct 翻译为*构件*。directive 的作用是给编译器一些指导，指出哪些代码需要并行化、需要怎么并行化，编译器根据程序员的指导信息生成最佳的并行化代码。有人将 directive 翻译为*编译制导语句*、*编译指导语句*、*指令语*、*指令*等，意思都对，但*编译制导语句*、*编译指导语句*太长，使用不便，*指令语*、*指令*中的*指令*一词太普通，易混，且只有强制的含义，没有指导的含义，不太恰当。*导语*一词长度、意思都比较合适。clause 是导语的修饰部分，更详细地表明导语的意图。有人将 clause 翻译为*子句*，而子句一词含有小句子的意思，实际上 OpenACC 中的 clause 都只有一个词，很短，不能称为一个句子。*子语*一词既说明了它与导语的关系，又有一个相同的*语*字，读起来顺口。construct 翻译为*构件*，取自建筑名词，实际功能也很相像。

本人水平有限，错误疏漏在所难免，欢迎批评指正，感激不尽。原始英文版请到 http://www.openacc.org/ 下载。本文档的最新版请到 www.bytes.me/openacc/ 下载。联系译者请发送电子邮件至 openacc@foxmail.com.

OpenACC 技术交流 QQ 群：284876008，欢迎加入，共同进步。

2013 年 8 月 26 日

第1章　引言

本文档讲述的编译器导语、库例程和环境变量共同定义了 OpenACC 应用编程接口(OpenACC API ,此后简称 OpenACC 接口)。 在用 C/C++和 Fortran 编写的程序中 ,OpenACC 接口将代码从主机 CPU 卸载到一个加速器设备。对各种操作系统、各种类型的主机 CPU 和各种类型的加速器，这里介绍的编程模型都是可移植的。导语扩展了 ISO/ANSI 标准 C/C++和 Fortran 语言，它允许程序员将标准 C/C++或 Fortran 程序迁移到加速器上。

利用本文定义的导语和编程模型，程序员创建有能力使用加速器的应用，但不用管理主机与加速器间的数据转移和程序转移，也不必费心加速器的启动与关闭. 当然了，这些细节隐含在编程模型之中，支持 OpenACC 接口的编译器和运行环境会管理它们。编程模型允许程序员告诉编译器更多的有用信息，包括指明在加速器本地存储数据、引导编译器将循环映射到加速器上，以及与性能相关的更多细节。

1.1　范围

这份 OpenACC 接口文档只涉及用户的加速器编程，由用户指定将哪些主机程序区域卸载到加速器设备上。程序的剩余部分将在主机上执行。本文不整体讲述主机编程环境的特性和限制，仅聚焦于那些卸载到加速器上的循环和代码区域的使用规范。

本文档不讲述使用编译器或其它工具自动侦测并将代码区域卸载到加速器的内容。也不讲述如何将循环或其它代码区域分配到与单个主机相连的多个加速器上。将来的编译器可能支持自动卸载、卸载到多个相同类型的加速器、卸载到多个不同类型的加速器，但这些可能发生的情况本文档概不涉及。

1.2　执行模型

OpenACC 编译器的执行模型是主机指导加速器设备（如 GPU）的运行。主机执行用户应用的大部分代码，并将计算密集型区域卸载到加速器上执行。设备上执行并行区域和内核区域，并行区域通常包含一个或多个工作分担(work-sharing)循环，内核区域通常包含一个或多个被作为设备上内核执行的循环。即使在加速器负责的区域，主机也必须精心安排程序的运行：在加速器设备上分配存储空间、初始化数据传输、将代码发送到加速器上、给并行区域传递参数、为设备端代码排队、等待完成、将结果传回主机、释放存储空间。大多数时候，主机可以将设备上的所有操作排成一队，一个接一个的顺序执行。

目前大多数加速器支持二到三层并行。大多数加速器支持粗粒度并：执行单元完全并行地执行。加速器可能有限支持粗粒度并行操作间的同步。许多加速器也支持细粒度并行：在单个执行单元上执行多个线程，这些线程可以快速地切换，从而可以忍受长时间的存储操作延时。最后，大多数加速器也支持每个执行单元内的单指令多数据(SIMD)操作或向量操作。该执行模型表明设备有多个层次的并行，因此程序员需要理解它们之间的区别。例如，一个完全并行的循环和一个可向量化但要求语句间同步的循环之间的区别。一个完全并行的循环

可以用粗粒度并行执行。有依赖关系的循环要么适当分割以允许粗粒度并行，要么在单个执行单元上以细粒度并行、向量并行或串行执行。

OpenACC 通过 gang、worker 和 vector 来显露这三个并行层次。gang 并行是粗粒度并行。加速器上将启动许多个 gang. worker 并行是细粒度并行。每一个 gang 都将有一个或多个 worker. vector 并行是一个 worker 内的 SIMD[1]操作或向量操作

当在设备上执行一个计算区域时，启动一个或多个 gang，每个 gang 都包含一个或多个 worker，这里的每个 worker 可能还有能力执行一个或多个向量通道（vector lanes）。多个 gang 以 gang 冗余模式（gang-redundant，GP 模式）开始执行，每个 gang 都有一个 orker 的一个向量通道冗余地执行相同的代码。当到达一个标记为 gang 层次工作分担的循环或嵌套循环时，程序开始以 gang 分裂模式（gang-partitioned，GP 模式）执行，这个循环或嵌套循环的所有迭代步分裂分配给各个 gang 以实现真正的并行执行，但每个活动的 gang 内仅有一个 worker 且一个 worker 内仅有一个向量通道。

当只有一个活动 worker 时，无论是在 gang 冗余模式中还是在 gang 分裂模式中，程序都处于 worker 单独模式（worker-single，WS 模式）。当只有一个活动向量通道时，程序就处于向量单独模式（vector-single，VS 模式）。当一个 gang 到达标记为 worker 层次工作分担的循环或嵌套循环时，这个 gang 就转换为 worker 分裂模式（worker-partitioned，WP 模式），从而激活这个 gang 中的所有 worker. 循环或嵌套循环的所有迭代步分裂分配给这个 gang 的各个 worker. 如果一个循环同时被标记为 gang 分裂和 worker 分裂，那么循环里的所有迭代步将分散到所有 gang 的所有 worker 上. 如果一个 worker 到达一个标记为向量层次并行的循环或嵌套循环，那么这个 worker 将转换为向量分裂模式（vector-partitioned，VP 模式）。与 worker 分裂模式类似，转换为 worker 分裂模式将激活这个 worker 的所有向量通道。使用向量操作或 SIMD 操作，这个循环或嵌套循环的所有迭代步将分散给所有向量通道。单个循环可以被标记为 gang 并行、worker 并行、vector 并行中的一种、二种或三种，相应地，所有迭代步会酌情分散到所有的 gang、worker、vector 之上.

主机程序以单线程开始执行，该线程由其程序计数器和栈确定。这个线程可以遥 OpenMP 编程接口之类的工具衍生出更多线程。加速器上，单个 gang 的单个 worker 的单个向量通道称为一个线程。在设备上执行时，程序会创建一个并行执行上下文，该上下文可能包含很多这样的线程。

用户不要试图在任何 gang 之间、worker 之间或 vector 之间使用屏障同步、关键区域或锁。执行模型允许编译器将一些 gang 执行完后再开始执行其它 gang. 这意味着在 gang 之间实施同步操作很可能会失败。特别地，gang 之间的屏障操作无法以可移植的方式实现，因为所有的 gang 可能永远不会在同一时刻处于活动状态。相似地，执行模型允许编译器执行完一个 gang 中的一些 worker 或一个 worker 中的一些向量通道之后，再开始执行其它的 worker 或向量通道。也允许编译器将一些 worker 或向量通道挂起，直到其它 worker 或向量通道执行完毕。这意味着在 worker 或向量通道之间实施同步操作很可能会失败。特别地，使用原子操作和一个忙碌-等待循环来在 worker 或向量通道间实现一个屏障或关键区域可能永远不

[1] Single Instruction Multiple Data，单指令多数据。

会成功，这是因为调度器可能将拥有锁的 worker 或向量通道挂起，导致等待这个锁的 worker 或向量通道永远无法完成。

某些设备上，加速器也可以创建和启动并行内核，并允许嵌套并行。这种情况下，OpenACC 导语可以被一个主机线程执行，也可以被一个加速器线程执行。本规范使用术语*本地线程*和*本地内存*来表示执行导语的线程和与该线程关联的内存，无论该线程是在主机上还是在加速器上。

相对于主机线程，大多数加速器可以异步操作。对这种设备，加速器有一个或多个活动队列。主机线程将数据移转、过程执行等操作压入活动队列。压入操作完成后，当设备正在独立、异步地操作时，主机线程继续向后执行。主机线程可以查询活动队列状态并等待某个队列的所有操作完成。某个活动列上的操作完成后，才会执行同一队列上的下一个操作；不同活动队列上的操作可以同时处于活动状态，并且可以以任意顺序完成。

1.3 存储模型

一个仅在主机上运行的程序与一个在主机+加速器上运行的程序，它们最大的区别在于加速器上的内存可能与主机内存完全分离。例如目前大多数 GPU 就是这样。这种情况下，设备内存可能无法被主机线程直接读写，因为它没有被映射到主机线程的虚拟存储空间。主机内存与设备内存间的所有数据移动必须由主机线程完成，主机线程通过系统调用在相互分离的内存间显式地移动数据。数据移动通常采用直接内存访问(Direct Memory Access, DMA)技术。类似地，不能假定加速器能读写主机内存，虽然有些加速器设备支持这样的操作，但常常有严重的性能损失。

在 CUDA C 和 OpenCL 等低层级加速器编程语言中，主机和加速器存储器分离的概念非常明确，内存间移动数据的语句甚至占据大部分用户代码。在 OpenACC 模型中，内存间的数据移动可以是隐式的，编译器根据程序员的导语管理这些数据移动。然而程序员必须了解背后这些相互分离的内存，理由包含但不限于：

- 有效加速一个区域的代码需要较高的计算密度，而计算密度的高低取决于主机内存与设备内存的存储带宽；
- 设备内存空间有限，因此操作大量数据的代码不能卸载到设备上。
- 主机上指针内存储的主机地址可能仅在主机上可用，设备上指针内存储的地址可能仅在设备上可用。主机指针在设备上解引用或设备指针在主机上解引用很可能不可用。

OpenACC 通过设备数据环境来揭示相互分离的内存。设备数据有一个显式生存期，从分配空间（或创建）直到被删除。如果设备与本地线程共享物理内存或虚拟内存，那么设备数据环境也被本地线程共享。这种情况下，编译器不必为设备创建新的数据副本，也不需要移动数据。如果设备内存与本地线程的内存物理地或虚拟地分离，那么编译器将在设备内存中创建新的数据副本并将数据从本地内存复制到设备环境中。

一些加速器(例如目前的 GPU)使用一个较弱的存储模型。特别地，它们不支持不同线程上操作的内存一致性；甚至，在同一个执行单元上，只有在存储操作语句之间显式地内存栅

栏才能保证内存一致性。否则，如果一个线程更新一个内存地址而另一个线程读取同一个地址，或者两个操作向同一个位置存入数据，那么硬件可能不保证每次运行都能得到相同的结果。尽管编译器可以检测到一些这样的潜在错误，但仍有可能编写出一个产生不一致数值结果的加速器并行区域或内核区域。

目前，一些加速器有一块软件管理的缓存，一些加速器有多块硬件管理的缓存。大多数加速器具有仅在特定情形下使用的硬件缓存，并且仅限于存放只读数据。在 CUDA C 和 OpenCL 等低层级语言的编程模型中，这些缓存交由程序员管理。在 OpenACC 模型中，编译器会根据程序员的导语管理这些缓存。

1.4 本文档中的约定

作为现行规范的一部分，关键字和标点符号以等宽字体出现：

```
#pragma acc
```

必须使用的关键字或其它名字采用*楷体*：

```
#pragma acc 导语名字
```

对 C 和 C++，*换行*意思为一行结尾处的换行符：

```
#pragma acc 导语名子  换行
```

方括号内的是可选语法；一个选项可以重复出现一次以上，这里用省略号表示：

```
#pragma acc 导语名字 [子语 [[,] 子语]…] 换行
```

为简化规范、传达合适的约束信息，一个 pgr *列表*就是一串用逗号分隔的 pgr 项目。例如*整数表达式列表*就是一串用逗号分隔的一个或多个整数表达式。一个*变量列表*就是一串用逗号分隔的变量名字或数组名字；在某些子语中，一个变量列表可能包含带下标范围的子数组，也可能包含写在两个斜杠内的公用块名字。*子语表列*是个例外，它是一个或多个子语，但不是必须用逗号分隔[1]。

```
#pragma acc 导语名字 [子语列表] 换行
```

1.5 本文档组织结构

本文档后续部分组织如下：

第二章，*导语*，讲述 C/C++和 Fortran 导语，它们用于构建加速器区域，以及提供更多信息以帮助编译器调度循环、对数据分类。

第三章，*运行时库*，定义供用户调用的函数、查询加速器设备参数的库例程、控制加速器程序运行时形为的库例程。

第四章，*环境变量*，定义供用户设置的环境变量，它们控制加速器程序执行时的行为。

[1] 也可以用空格。--译者注。

第五章，词汇表，定义本文档用到的术语。

附录 A，对支持特定设备编译器的建议，给编译器开发者的一些建议，以更好地支持跨编译器移植性、与其它加速器接口的互操作性。

1.6 参考文献

American National Standard Programming Language C, ANSI X3.159-1989 (ANSI C).
ISO/IEC 9899:1999, *Information Technology – Programming Languages – C* (C99).
ISO/IEC 14882:1998, *Information Technology – Programming Languages – C++*.
ISO/IEC 1539-1:2004, *Information Technology – Programming Languages – Fortran Part 1: Base Language*, (Fortran 2003).
OpenMP Application Program Interface, version 3.1, July 2011
PGI Accelerator Programming Model for Fortran & C, version 1.3, November 2011
NVIDIA CUDA™ C Programming Guide, version 5.0, October 2012.
The OpenCL Specification, version 1.2, Khronos OpenCL Working Group, November 2011.

1.7 1.0 版到 2.0 版的变化

- `_OPENACC` 的值更新为 `201306`
- `parallel` 和 `kernels` 导语上的 `default(none)` 子语
- `parallel` 构件中标量的隐式数据属性发生了改变
- 在带有 `loop` 导语的循环内，澄清了标量的隐式数据属性，明确具有独立数据属性的情形。
- `async` 子语的值 `acc_async_sync` 和 `acc_async_noval`
- 澄清 `reduction` 子语在一个 `gang` 循环上的行为
- 澄清允许的循环嵌套(`gang` 不能出现在 `worker` 里面，`worker` 不能出现在 `vector` 里面)
- `Parallel`、`kernels` 和 `update` 导语上的 `wait` 子语
- `wait` 导语上的 `async` 子语
- `enter data` 和 `exit data` 导语
- 现在多个数据子语中都可以指定 Fortran 公共块名字
- `declare` 导语的 `link` 子语
- `declare` 导语对全局数据的形为
- 对带有一个 C 或 C++指针变量的数据子语，澄清它的形为
- 预先确定数据属性
- 支持 C/C++多维动态数组

- loop 构件的 tile 和 auto 子语
- 引入 update host 的同义子语 update self
- routine 导语，并支持分别编译
- device_type 子语，并支持多种设备类型
- 以一个 parallel 或 kernels 区域包含另一个 parallel 或 kernels 区域的方式来嵌套并行
- atomic 构件
- 新概念：gang 冗余、gang 分裂； worker 单独、 worker 分裂；向量单独、向量分裂；线程
- 新的接口例程：
  - acc_wait 和 acc_wait_all 分别代替 acc_async_wait 和 acc_async_wait_all
  - acc_wait_async
  - acc_copyin、acc_present_or_copyin
  - acc_create、acc_present_or_create
  - acc_copyout、acc_delete
  - acc_map_data、acc_unmap_data
  - acc_deviceptr、acc_hostptr
  - acc_is_present
  - acc_memcpy_to_device、acc_memcpy_from_device
  - acc_update_device、acc_update_self
- 定义使用多个主机线程时的形为，例如使用 OpenMP
- 对特定编译器的建议

1.8 未来修订的主题

下列主题正在考虑加入未来的修订版本。我们已经知道其中一些主题很重要，而其它主题还需要听取用户的反馈意见。读者如果有反馈意见或想参与讨论，可以在 www.openacc.org 论坛上发帖，也可以向 feedback@openacc.org 发送电子邮件。不保证或暗示这些意见都会被添加到下一版之中。

- 完全支持 C 和 C++结构体和结构体成员，包括指针成员。
- 完全支持 Fortran 派生类型和派生类型成员，包括动态分配成员和指针成员。
- 确定支持多主机线程。
- 可以选择去掉 vecotr 循环和 worker 循环尾部的同步或屏障。
- device_type 子语后面允许有一个 if 子语。
- loop 导语的 default(none)子语。
- loop 导语的 shared 子语（或类似的名字）。
- 为性能分析器或追踪器或其它运行时数据收集工具提供一个标准接口。
- 单线程更好地支持多个设备，不管是相同类型的设备或不同类型的设备。

第2章　导语

本章讲述 OpenACC 导语的语法和形为。在 C 和 C++中，用语言本身提供的#pragam 机制来引入 OpenACC 导语。在 Fortran 中，用带有独特前导符的注释来引入 OpenACC 导语。如果不支持或者关闭了 OpenACC 功能，编译器将会忽略 OpenACC 导语。

限制

- Fortran 中，OpenACC 导语不能出现在 PURE 和 ELEMENTAL 过程中。

2.1 导语格式

在 C/C++中，使用#pragma 机制指定 OpenACC 导语，语法是：

```
#pragma acc 导语名字 [子语列表] 换行
```

每个导语都以#pragma acc 开始。导语的其它部分都遵守 C/C++中 pragma 的使用规范。#的前后都可以使用空白字符；导语中使用空白字符来分隔各字段。#pragma 后面的预处理标记使用宏替换。导语区分大小写。一个 OpenACC 导语作用于紧接着的语句、结构块和循环。

Fortran 自由格式源文件中，用下列格式指定 OpenACC 导语：

```
!$acc 导语名字 [子语列表]
```

第一个注释字符(!)可以放在任意列，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。每行长度、空白字符、续行符规则同样适用于导语行。导语起始行的前导符后必须有空白字符。待续行[1]中导语部分的最后一个非空白字符必须是连字符(&)，连字符后面仍然可以写注释；接续行中导语必需以前导符开始(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。导语行上也可以放注释，注释以感叹号开始，直至行尾。如果前导符后面的第一个非空白字符是一个感叹号，那么该行被忽略。

在固定格式 Fortran 源代码文件中，OpenACC 导语可以采取下列形式中的一个：

```
!$acc 导语名字 [ 子语列表 ]
c$acc 导语名字 [ 子语列表 ]
*$acc 导语名字 [ 子语列表 ]
```

前导符(!$acc、c$acc 或*$acc)必须写在 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导语行。导语起始行第 6 列必须是空格或 0，接续行导语在第 6 列不能是空格或零. 导语行也可以添加注释，注释可以从第 7 列（包含）之后的任意列以感叹号开始，至行尾结束。

[1] 如果单个程序语句被分写在多行之上，最后一行之外的每一行都称为待续行，第一行之外的每一行都称为接续行。--译者注。

在 Fortran 中，导语不区分大小写。分写在多行的单个程序语句中间也不能混入导语，同样地，分写在多行的单个导语中间不能混入程序语句。本文档中所有 Fortran OpenACC 导语例子都采用自由格式。

每个导语中只能有一个*导语名字*，一个例外组合导语名字，它被视为单个*导语名字*。如无特别规定，多个子语出现的顺序无关紧要。如无特别规定，子语可以重复多出现多次。有些子语的参数会包含一个*列表*。

2.2 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。当且仅当 OpenACC 导语功能打开时，编译器必须定义这个宏。这里讲述的版本是 201306.

2.3 内部控制变量

一个符合 OpenACC 标准的编译器表现得好像有一些内部控制变量(Internal control variables, ICVs)控制着程序的行为。编译器通过环境变量或通过调用 OpenACC 例程将这些内部控制变量初始化。通过调用 OpenACC 例程，程序可以获得这些变量的值。

这些外部控制变量是：

- `acc-device-type-var` 控制使用哪种类型的加速器。
- `acc-device-num-var` 控制使用所选类型中的哪个加速器。

2.3.1 修改和取出内部控制变量值

下表列出了修改内部控制变量的环境变量和例程、取出内部控制变量值的例程：

内部控制变量	修改值的方法	取出值的方法
acc-device-type-var	`ACC_DEVICE_TYPE` `acc_set_device_type`	`acc_get_device_type`
acc-device-num-var	`ACC_DEVICE_NUM` `acc_set_device_num`	`acc_get_device_num`

这些变量的初值由编译器定义。赋初值之后，执行任何 OpenACC 构件、例程之前，用户设定的所有环境变量值都已经被读出，并依此修改了内部控制变量的值。OpenACC 构件上的子语不会修改这内存控制变量的值。每个主机线程都会保存所有内部控制变量的一个副本。无法修改设备线程的内存控制变量值。

2.4 特定设备的子语

使用 `device_type` 子语，OpenACC 导语可以为不同类型的加速器指定不同的子语和子语参数。`device_type` 子语的参数是一个用逗号分隔的列表，列表是一个或多个加速器架构名称标识符，或者是一个星号。单个导语可以有一个或几个 `device_type` 子语。导语上不带 `device_type` 子语的子语可以适用于所有加速器类型。跟在一个 `device_type`

子语后面，直到导语结束或直到下一个 device_type 子语之间的所有子语，都与这个 device_type 子语相关联。与 device_type 子语关联的子语仅在为对应加速器设备类型编译时才起作用。与带星号参数的 device_type 子语相关联的子语，对没有出现在本导语上其它 device_type 子语中的加速器设备类型起作用。device_type 子语可以以任意顺序出现。对每个导语，只有特定的子语可以跟在 device_type 子语后面。

在所有 device_type 子语之前的子语是缺省子语。如果某个缺省子语的同名子语与 device_type 子语相关联，那么该子语后一次出现的参数值起作用。如果没有某个设备的 device_type 子语，或者有该设备的 device_type 子语但没有缺省子语的同名子语与这个 device_type 子语相关联，那么使用缺省值。

支持哪些加速器设备类型由编译器决定。取决于编译器和编译环境，一个编译器可能只支持一个加速器设备类型，也可能支持多个加速器设备类型，但一次只支持一个，还可能在单次编译中支持多种加速器设备类型。

加速器的体系结构名字可以比较笼统，例如某个制造厂商，或者比较具体，如设备的某个特定型号；可以到附件 A.1 目标设备查看推荐的名字。device_type 子语可以指定大多数特定体系架构名字，当为某个特定的设备编译时，编译器使用与这个设备类型相关联的子语；与其它所有 device_type 子语相关联的子语将被忽略。在这个语境中，星号是最宽泛的体系结构名字。

语法

device_type 子语的语法是

```
device_type( * )
device_type(设备类型列表)
```

device_type 子语可以缩写为 dtype.

2.5 加速器计算构件

2.5.1 parallel 构件

概要

这个基本构件开启加速器设备上的并行执行。

语法

在 C 和 C++中，OpenACC parallel 导语的语法为

```
#pragma acc parallel [ 子语列表 ] 换行
    结构块
```

在 Fortran 中的语法是

```
!$acc parallel [ 子语列表 ]
    结构块
!$acc end parallel
```

这里的子语是下列中的一个:

```
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 )
vector_length(整数表达式 )
device_type(设备类型列表)
if(条件)
reduction(操作符:变量列表)
copy(变量列表)
copyin(变量列表)
copyout(变量列表)
create(变量列表)
present(变量列表)
present_or_copy(变量列表)
present_or_copyin(变量列表)
present_or_copyout(变量列表)
present_or_create(变量列表)
deviceptr(变量列表)
private(变量列表)
firstprivate(变量列表)
default(none)
```

描述

当遇到一个加速器 parallel 构件，程序就创建一个或多个 gang 来执行这个加速器并行区域。gang 的数量、每个 gang 中的 worker 数量和每个 worker 中的向量通道数量，在这个并行区域的存续期内保持为常数。每个 gang 都开始以 gang 冗余模式执行结构化块。这意味着，在并行区域之内，带 loop 导语且以 gang 层次共工作分担的循环之外的代码，将被所有的 gang 冗余地执行。

如果没有使用 async 子句，加速器并行区域结束处将会有一个隐式屏障，此时本地线程不再向前执行，直到所有的 gang 都已到达并行区域的结尾。

计算构件里引用某个变量，当该变量没有出现在本计算构件的任何数据子语中，也没有出现在包围本计算构件的数据构件中并且没有预先确定的数据属性，如果构件上没有 default(none)子语，那么编译器将隐式地确定变量的数据属性。如果一个聚合数据类型的变量(或数组)在 parallel 构件中被引用，并且没有出现在该 parallel 构件的数据子语中，也没有包含在任何 data 构件中，那么它等同于出现在 parallel 构件的 present_or_copy 子语中[1]。如果一个标量变量在 parallel 构件中被引用，并且没有出现在该 parallel 构件的数据子语中，也没有包含在任何 data 构件中，那么它等同于出现在 parallel 构件的 firstprivate 子语中。

[1]变量的缺省子语是 present_or_copy。—译者注。

限制

- 一个程序不能通分支转入或跳出 OpenACC `parallel` 构件。
- 一个程序决不能依赖于子语的求值顺序或求值的副作用。
- 只有 `async`、`wait`、`num_gangs`、`num_workers` 和 `vector_length` 这些子语可以跟在一个 `device_type` 子语后面。
- 至多出现一个 `if` 子语。Fortran 中，*条件*的运算结果必须是一个标量逻辑值；C 和 C++中，*条件*的运算结果必须是一个标量整数值。

数据子语 `copy`、`copyin`、`copyout`、`create`、`present`、`present_or_copy`、`present_or_copyin`、`present_or_copyout`、`present_or_create`、`deviceptr`、`firstprivate` 和 `private` 的详细描述在 2.6 节*数据环境*。`device_type` 子语在 2.4 节*设备相关子语*中有详细描述。

2.5.2 kernels 构件

概要

一个程序的 `kernels` 构件区域将被编译成一系列在加速器设备上执行的 `kernel`.

语法

C 和 C++中，OpenACC `kernels` 导语的语法是：

```
#pragma acc kernels [子语列表] 换行
    结构块
```

在 Fortran 中的语法是

```
!$acc kernels [子语列表]
    结构块
!$acc end kernels
```

这里的*子语*是下列中的一个：

```
async[(整数表达式)]
wait[(整数表达式列表)]
device_type(设备类型列表)
if(条件)
copy(变量列表)
copyin(变量列表)
copyout(变量列表)
create(变量列表)
present(变量列表)
present_or_copy(变量列表)
present_or_copyin(变量列表)
present_or_copyout(变量列表)
present_or_create(变量列表)
deviceptr(变量列表)
```

default(none)

描述

编译器将 kernels 区域内的代码分割为一系列的加速器 kernel．通常，每个循环成为一个单独的 kernel．当程序遇到一个 kernels 构件时，它在设备上按顺序启动这一系列 kernel．对不同的 kernel，gang 的数量、每个 gang 包含多少个 worker、vector 的长度以及三者的组织方式都可能不相同。

如果没有使用 async 子语，kernels 区域结束时将有一个隐式屏障，本地线程不再向前执行，直至所有的 kernel 都执行完毕。

计算构件里引用某个变量，若该变量没有出现在本计算构件的任何数据子语中，也没有出现在包围本计算构件的数据构件中并且没有预先确定的数据属性，如果构件上没有 default(none)子语，那么编译器将隐式地确定变量的数据属性。如果一个聚合数据类型的变量(或数组)在 kernels 构件中被引用，并且没有出现在该 kernels 构件的数据子语中，也没有包含在任何 data 构件中，那么它等同于出现在 kernels 构件的 present_or_copy 子语中[1]。如果一个标量变量在 kernels 构件中被引用，并且没有出现在该 kernels 构件的数据子语中，也没有包含在任何 data 构件中，那么它等同于出现在 kernels 构件的 copy 子语中。

限制

- 一个程序不能通分支转入或跳出 OpenACC kernels 构件。
- 一个程序决不能依赖于子语的求值顺序或求值的副作用。
- 只有 async 子语和 wait 子语可以跟在 device_type 子语后面。
- 至多出现一个 if 子语。Fortran 中，条件的运算结果必须是一个标量逻辑值；C/C++中，条件的运算结果必须是一个标量整数值。

数据子语 copy、copyin、copyout、create、present、present_or_copy、present_or_copyin、present_or_copyout、present_or_create 和 deviceptr 的详细描述在 2.6 节数据环境。device_type 子语在 2.4 节设备相关子语中详细描述。

2.5.3 if 子语

if 子语是 parallel 构件和 kernels 构件的可选项；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，编译器为本构件生成两份代码，一份在加速器上执行，另一份在遭遇的本地线程上执行。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，那份加速器端代码将被执行。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，遭遇的本地线程将执行这个构件。

2.5.4 async 子语

async 子语是可选项；更多信息参见 2.14 节异步形为。

[1]变量的缺省子语是 present_or_copy。—译者注。

2.5.5 wait 子语

wait 子语是可选项；更多信息参见 2.14 节异步形为。

2.5.6 num_gangs 子语

parallel 构件允许使用 num_gangs 子语。整数表达式的值规定了并行执行该区域的 gang 的数量。如果没有本子语，编译器将使用自定义的缺省值。缺省值可能依赖于构件内的代码。

2.5.7 num_workers 子语

parallel 构件允许使用 num_workers 子语。一个 gang 由 worker 单独模式转换为 worker 分裂模式后，每个 gang 中活动 worker 的数量由整数表达式的值确定。如果没有本子语，编译器将使用自定义的缺省值；缺省值可能是 1，每个 parallel 构件的缺省值可能不同.

2.5.8 vector_length 子语

parallel 构件允许使用 vector_length 子语。一个 worker 由向量单独模式转换为向量分裂模式后，每个 worker 中活动向量通道的数量由整数表达式的值确定。这个子语规定了所有 gang 中每个 worker 的向量长度或 SIMD 操作长度。如果没有本子语，编译器将使用自定义的缺省值。向量长度将被用在 loop 导语的 vector 子语上，编译器自动向量化循环时也使用这个长度。编译器可能会限制向量长度表达式的取值范围。

2.5.9 private 子语

parallel 构件允许使用 private 子语；对列表中的每一项，每个并行 gang 都会创建一个副本。

2.5.10 firstprivate 子语

parallel 构件允许使用 firstprivate 子语；对列表中的每一项，每个并行 gang 都会创建一个副本。当遇到 parallel 构件时，程序使用主机中的对应项来初始化 parallel 构件里的副本。

2.5.11 reduction 子语

reduction 子语可以用到 parallel 构件上。它指定一个归约操作符和一个或多个标量变量。对每个变量，每个并行运行的 gang 都会创建一个副本，并根据指定的运算符来初始化这个副本。在并行区域结束时，归约运算符使用每一个 gang 里的值、变量的原始值计算出归约结果，并将该结果存入原始变量。归约结果在并行区域之后可用。

下表列出了可用的操作符及相应的初始值，初始值跟数据类型有关。对 max 和 min 归约，初始化值分别为变量所属数据类型能表示的最小值、最大值。C 与 C++(int、float、double、complex)和 Fortran(integer、real、double precision、complex)的数

值数据类型都可以用在本子语中。

C 和 C++		Fortran	
操作符	初始值	操作符	初始值
+	0	+	0
*	1	*	1
max	最小值	max	最小值
min	最大值	min	最大值
&	~0	iand	所有位为 1
|	0	ior	0
^	0	ieor	0
&&	1	.and.	.true.
||	0	.or.	.false.
		.eqv.	.true.
		.neqv.	.false.

2.5.12 default(none)子语

default(none)子语是可选项。它告诉编译器不要隐式地决定任何变量的数据属性，而是要求计算区域内没有预先确定数据属性的所有变量或数组都必须显式地出现在一个数据子语中，数据子语可以在本计算构件上，也可以在字面包含这个 parallel 或 kernels 构件的数据构件上。

2.6 数据环境

本节讲述变量的数据属性。一个变量的数据属性可以是预定（预先确定）、隐定（隐式确定）或显定（显式确定）。具有预定数据属性的变量不能出现在与该数据属性冲突的数据子语中。具有隐定数据属性的变量可以出现在那些能覆盖预定属性的数据子语中。数据构件、计算构件或 declare 导语上的数据子语中出现的变量具有显定属性。

OpenACC 支持加速器内存与主机内存相分离的系统，也支持加速器与主机共享内存的系统。在前一种情形中，系统具有相互分离加速器内存与主机内存。前一种情形中，系统具有一个共享内存。后一种情形称为共享内存设备，因为加速与主机线程共享内存；前一种情形称为非共享内存设备。当一个嵌套的 OpenACC 构件在设备上执行时，该构件的缺省目标设备就是所遭遇加速器线程所在的设备。这种情况下，目标设备与遭遇线程共享内存空间。

2.6.1 预定数据属性的变量

一个 loop 导语关联的 C 语言 for 语句中或 Fortran 语言 do 语句中的循环变量被预先确定为每个线程的私有变量，这些线程执行循环的所有迭代步。在 parallel 构件或 kernels 构件内，Fortran 语言 do 语句中的循环变量被预先确定为执行该循环的线程的私有变量。

一个计算构件中，C 语言代码中声明的变量被预先确定为执行该代码块线程的私有变量。计算构件里，被调用过程中声明的变量，被预先确定为执行该过程调用线程的私有变量。

2.6.2 数据区域和数据生存期

数据区域有四种。当遇到一个数据构件，程序就创建一个数据区域。为数据构件而在设备上创建的数据，它的生存期与数据构件相关联；在程序退出这个数据区域之前，该数据一直可用。

若一个计算构件带有显示数据子语或带有编译器添加的隐式数据空间分配，那么当程序遇到它的时候，就会创建一个与该计算构件生存期相同的数据区域。为这个计算构件在加速器上创建的数据，它的生存期与构件关联区域的生存期相同，就像带有一个数据构件一样。

当进入一个过程，程序创建一个隐式数据区域，它的生存期与该过程相同。也就是，过程被调用时创建隐式数据区域，程序从过程调用点返回时退出这个隐式数据区域。隐式数据区域内，在加速器上创建的数据，它的生存期与被调用过程相同。

程序执行过程本身也有一个隐式数据区域。隐式程序数据区域的生存期就是程序的执行过程。加速器上创建的静态数据或全局数据，它们的生存期是程序的执行过程，或者从程序连接并初始化加速器开始，直到断开并关闭加速器结束。

除了数据生存期，一个程序还可以使用 `enter data` 导语和 `exit data` 导语或运行时接口例程在加速器上创建和删除数据。当程序执行一个 `enter data` 导语，或者调用运行时接口例程 `acc_copyin` 或 `acc_create`，程序就进入一个数据生存期，其作用范围包括相应导语上的所有变量、数组、子数组，或者相应运行时例程参数列表里的所有变量。用这种方法创建的加速器变量，它的生存期从执行相应导语或调用相应运行时例程时开始，直到执行一个 `exit data` 导语或者调用运行时例程 `acc_copyout` 或 `acc_delete` 结束；如果没有 `exit data` 导语或者没有调用合适的运行时例程，加速器上数据的生存期延续至程序退出。

2.6.3 data 构件

概要

无论是否需要在进入本区域时将数据从主机复制到设备内存，也无论是否需在离开本区域时将数据从设备复制到主机内存，在本区域的持续期内，`data` 构件指明的标量、数组和子数组都会在设备内存上开辟空间。

语法

C 和 C++中，OpenACC `data` 导语的语法是

```
#pragma acc data [子语列表] 换行
    结构块
```

在 Fortran 中的语法是

```
!$acc data [子语列表]
    结构块
!$acc end data
```

这里的子语为下列中的一个：

```
if(条件)
copy(变量列表)
copyin(变量列表)
copyout(变量列表)
create(变量列表)
present(变量列表)
present_or_copy(变量列表)
present_or_copyin(变量列表)
present_or_copyout(变量列表)
present_or_create(变量列表)
deviceptr(变量列表)
```

描述

根据需要，在设备内存上为数据开辟空间，并将数据从主机内存或本地内存复制到设备，或者复制回来。它的数据子语将在 2.6.5 节*数据子语*中讲述。

2.6.3.1 if 子语

`if` 子语是可选项；没有 `if` 子语的时候，编译器生成的代码将在加速器设备上开辟内存空间，并根据需要在设备内存与本地内存间移动数据。有 `if` 子语的时候，程序在某些条件下才会在设备上开辟内存空间，并将数据移动到设备上或将数据从设备上取回。当 `if` 子语中*条件*的值为零（对 C 和 C++）或 `.false.`（对 Fortran）时，不会开辟设备内存空间，也不会移动数据。当*条件*的值为非零（对 C 和 C++）或 `.true.`（对 Fortran）时，程序将按照指定的方式为数据开辟空间，并移动数据。至多可以出现一个 `if` 子语。

2.6.4 enter data 导语和 exit data 导语

概要

`enter data` 导语用来定义在设备上分配空间的标量、数组和子数组，它们在程序的存续期内一直存在，直至遇到一个 `exit data` 导语来撤销数据。它们还可以说明是否在 `enter data` 导语处将数据从主机内存复制到设备内存，是否在 `exit data` 导语处将数据从设备内存复制到主机内存。`enter data` 导语与匹配的 `exit data` 导语之间的程序动态范围是相应数据的生存期。

语法

C 和 C++中，OpenACC `enter data` 导语的语法是

```
#pragma acc enter data 子语列表  换行
```

Fortran 中，语法是

```
!$acc enter data 子语列表
```

这里的子语为下列中的一个：

```
if(条件)
async[(整数表达式)]
wait[(整数表达式列表)]
copyin(变量列表)
create(变量列表)
present_or_copyin(变量列表)
present_or_create(变量列表)
```

C 和 C++中，OpenACC exit data 导语的语法是

```
#pragma acc exit data 子语列表  换行
```

Fortran 中，语法是

```
!$acc exit data 子语列表
```

这里的子语为下列中的一个：

```
if(条件)
async[(整数表达式)]
wait[(整数表达式列表)]
copyout(变量列表)
delete(变量列表)
```

描述

一个 enter data 导语，将在设备内存上分配空间，并可选择将数据从主机内存或本地内存复制到设备。这个动作进入了这些变量、数组和子数组的一个生存期，在数据生存期内，这些数据可以用在构件的 present 子语中。

一个 exit data 导语，可选地将数据从设备内存复制到主机内存或本地内存，然后从设备内存中撤销数据。这个动作退动相应的数据生存期。exit data 导语只能用于由 enter data 导语或运行时例程创建的数据生存期。它们的数据子语在 2.6.5 节数据子语中讲述。

2.6.4.1 if 子语

if 子语是可选项；没有 if 子语的时候，编译器生成的代码将在加速器设备上开辟或撤销内存空间，并在设备内存与本地内存之间移动数据。有 if 子语的时候，程序有条件地分配或撤销设备内存，并将数据移动到设备上或将数据从设备上取回。当 if 子语中条件的值为零（对 C 和 C++）或.false.（对 Fortran）时，不会分配或撤销设备内存，也不会移动数据。当条件的值为非零（对 C 和 C++）或.true.（对 Fortran）时，数据将按照指定的方式创建、撤销和移动。

2.6.4.2 async 子语

async 子语是可选项；更多信息参见 2.14 节异步形为。

2.6.4.3 wait 子语

`wait` 子语是可选项；更多信息参见 2.14 节*异步形为*。

2.6.5 数据子语

这些数据子语可以出现在 `parallel` 构件、`kernels` 构件、`data` 构件、`enter data` 导语和 `exit data` 导语上。每个子语的参数列表是一串用逗号分隔的变量名字、数组名字或给定下标范围的子数组。如果一个由两个斜杠包围的 Fortran 共公块名字出现在一个 `declare` 导语的 `link` 子语中，那么对 `deviceptr` 和 `present` 之外的所有子语，参数列表中也可以包含这个公共块的名字。在所有情形下，编译器都会在设备内存上为这些变量或数组创建、管理一个副本，即为变量和数组创建一个可见设备副本。

这样做是为了支持那些在物理内存上和逻辑内存上都与本地线程相分离的加速器。然而，如果加速器可以直接访问本地内存，编译器可以通过简单地共享本地内存中的数据来避免开辟内存空间和数据移动。所以，如果一个程序在主机端使用并为数据赋值，在设备上也使用相同的数据并为之赋值，且没有使用 `update` 导语来管理两个副本的一致性，那么使用不同的编译器、不同的加速器都有可能得到不同的结果。

限制

- 数据子语不能跟在 `device_type` 子语后面。

2.6.5.1 数据子语内的数据规范

C 和 C++中，子数组就是数组名后跟一个方括号，方括号内的起始下标和长度用来指定数组范围，例如

```
AA[2:n]
```

下边界的缺省值为零。如果没有指定长度且数组的尺寸已知，那么使用数组的声明尺寸；否则必须指定长度。子数组 `AA[2:n]`表示元素 `AA[2], AA[3],…, AA[2+n-1].`

C 和 C++中，二维数组的声明方法至少有四种：

- 静态尺寸数组：`float AA[100][200];`
- 指向静态尺寸行的指针：`typedef float row[200]; row* BB;`
- 指针的静态尺寸数组：`float* CC[200];`
- 二重指针：`float** DD;`

每一个维度都可以是静态尺寸，也可以是一个动态分配内存的指针。它们都可以包含在一个数据子语中，并可以用子数组符号来指定一个矩形数组：

- `AA[2:n][0:200]`
- `BB[2:n][0:m]`
- `CC[2:n][0:m]`
- `DD[2:n][0:m]`

C 和 C++中，可以指定数组的多维矩形子数组，数组的各个维度可以是静态的也可以动态分配的。对静态尺寸的维度，除了第一个维度之外的所有维度都必须指定尺寸，以符合连续数据块的要求，下文会有进一步讨论。对动态分配的维度，编译器将在设备上分配与主机

指针相对应的指针，并对设备指针进行适当填充。

Fortran 中，子数组就是一个数组名后跟一个指定范围的圆括号，圆括号内用逗号分隔上、下边界，例如

```
arr(1:high,low:100)
```

没有指定上边界或下边界的时候，如果知道数组在声明或开辟空间时的边界，就使用这些边界。为保持连续的数据约束，除最后一个维度之外所有维度都必须指定为长，详细说讨论在下面。

限制

- Fortran 中，*不定尺寸形参数组*[1]的最后一个维度必须指定上边界。
- C 和 C++中，必须显式指定动态分配数组的各维度长度。
- C 和 C++中，无论是在主机上还是在设备上，在数据生存期之内修改指针数组里的指针，都将带来未定义的行为。
- 如果一个数据子语中指定了一个子数组，编译器可能选择仅为该子数组分配加速器内存。
- Fortran 中，可以在子语中指定指针，但设备内存中不再保留指针的指向关系。
- 动态分配的多维 C 语言数组除外，数据子语中的任何数组、子数组、Fortran 数组指针，都必须是一块连续的内存。
- C 和 C++中，如果指定了一个变量、结构数组、或者类，那么该结构或类的成员都会被酌情分配空间、复制数据。如果结构或类的一个成员是指针类型，那么不会隐式复制该指针指向的数据。
- Fortran 中，如果指定了一个派生类型的变量或数组，那么会酌情为该派生类型的所有成员分配空间和复制数据。对任何具有 `allocatable` 或 `pointer` 属性的成员，通过该成员访问的数据都不会被复制。
- `data` 构件上，如果数据子语内子数组的下标中用到了一个表达式，无论表达式中的变量值是否在这个数据区域发生改变，退出数据区域复制数据时，都会使用进入数据区域时的变量值。

2.6.5.2 deviceptr 子语

`deviceptr` 子语用来声明*变量列表*中的指针是设备指针，因此不必再为该指针指向的数据分配空间，也不必在主机与设备之间移动数据。

C 和 C++中，*变量列表*中的变量必须是指针。

Fortran 中，*变量列表*中的变量必须是形式参数（数组或标量），并且不能具有 Fortran `pointer`、`allocatable` 或 `value` 属性。

对一个共享内存设备，主机指针与设备指针是一样的，因此这个子语不起作用。

---

[1] assumed-size dummy array，这种数组会显式地声明除了最后一个维度以外的所有维度的长度，最后一个维度的长度用星号来代替。它是 Fortran 语言早期版本的一种延续，新程序不应该再使用它。--译者注。

2.6.5.3 copy 子语

copy 子语用来声明变量列表中的变量、数组、子数组或公共块存在于本地内存。对于非共享内存加速器，需要将它们复制到设备内存，在加速上给它们赋值后再复制回本地内存。如果声明的是一个子数组，那么只需复制该数组的子数组部分。该子语用在一个数据构件或计算构件上的时候，进入区域时为数据分配空间并将数据复制到设备内存，离开区域时将数据复制回本地内存并撤销数据空间。如果设备与本地线程共享内存，那么 copy 子语中的数据也是共享的；不会发生内存分配和数据复制。

2.6.5.4 copyin 子语

copyin 子语用来声明变量列表里的变量、数组、子数组和公共块存在于本地内存。对于非共享内存的加速器，需要将它们复制到设备内存。如果声明的是一个子数组，那么只需复制该数组的子数组部分。一个变量、数组或子数组出现在一个 copyin 子语中意味着，即使它们的值在加速器上改变了，也不必将它们从设备内存复制回本地内存。该子语用在一个数据构件或计算构件上的时候，进入区域时为数据分配空间并将数据复制到设备内存，离开区域时将撤销数据空间。该子语用在 enter data 导语上的时候，数据被分配空间并复制到设备内存。如果设备与本地线程共享内存，那么 copyin 子语中的数据也是共享的；不会发生内存分配和数据复制。

2.6.5.5 copyout 子语

copyout 子语用来声明，变量列表里的变量、数组、子数组和公共块被在设备上赋值或者它们的值保存在设备内存中。对于非共享内存的加速器，需要在加速器区域结束处将它们复制回本地内存。如果声明的是一个子数组，那么只需复制该数组的子数组部分。一个变量、数组或子数组出现在一个 copyin 子语中意味着，即使加速器上会用到它们的值，也不必将它们从本地内存复制到设备内存。该子语用在一个数据构件或计算构件上的时候，进入区域时为数据分配空间，离开区域时将数据复制回本地内存并撤销数据空间。该子语用在 exit data 导语上的时候，数据被复制回本地内存，然后撤销。如果设备与本地线程共享内存，那么 copyout 子语中的数据也是共享的；不会发生内存分配和数据复制。

2.6.5.6 create 子语

create 子语用来声明变量列表里的变量、数组、子数组或公共块需要在非共享内存的加速器上分配设备内存，但加速器不需要本地内存中的变量值，加速器上计算得到的任何值、对变量的赋值都不需要复制回本地内存。该子语用在一个数据构件或计算构件上的时候，进入区域时为数据分配设备内存，离开区域时撤销数据的设备内存。该子语用在 enter data 导语上的时候，在设备内存中为数据分配空间。本子语不引发本地内存与设备内存间的数据复制。如果设备与本地线程共享内存，那么 copyout 子语中的数据也是共享的；不会发生内存分配和数据复制。

2.6.5.7 delete 子语

delete 子语用在 exit data 导语上，不把数组、子数组或公共块的值复制回本地内存，就直接撤销它们。在非共享内存设备上，数据被撤销。如果设备与本地线程共享内存，无动作。

2.6.5.8 present 子语

在非共享内存设备上，present 子语告诉编译器变量列表中的变量、数组已经存在于设备内存，因为已经有数据区域或数据生存期包含着本区域，例如本构件所在例程的主调例程内的数据构件、本例程前面的 enter data 导语、在本例程前面调用的运行时接口例程。编译器将会发现并使用这些已经存在于加速器上的数据。如果没有活动的数据生存期已经将所有的变量或数组放置到加速器上，形为未定义，特别地，程序将报错中止。

如果包含本构件的数据区域指定的是一个子数组，那么 present 子语必须指定相同的子数组，或指定该数据生存期内子数组的一个适当子集构成的子数组[1]。如果 present 子语中的子数组的某些元素不包含在数据生存期的子数组中，那么它是一个运行时错误(runtime error)。

2.6.5.9 present_or_copy 子语

在非共享内存加速器上，present_or_copy 子语用来告诉编译器去检测变量列表中的每一个变量或数组是否已经存在于加速器内存中，就像有个 present 子语一样。

如果这些数据已经存在 程序行为和带 present 子语时一样。不会分配新的设备内存，没有数据在设备内存上移入、移出。

如果数据不存在，程序形为和带 copy 子语时一样。进入区域时分配数据空间并将数据复制到设备内存，离开区域时将数据复制回本地内存并撤销数据内存。

这个子语可以缩写为 pcopy. present 子语中对子数组的限制同样适用于本子语。

2.6.5.10 present_or_copyin 子语

在非共享内存加速器上，present_or_copyin 子语用来告诉编译器去检测变量列表中的每一个变量或数组是否已经存在于加速器内存中，就像有个 present 子语一样。

如果这些数据已经存在 程序行为和带 present 子语时一样。不会分配新的设备内存，没有数据在设备内存上移入、移出。

如果这些数据不在加速器上，程序的行为和带 copyin 子语时一样。该子语用在一个数据构件或计算构件上的时候，进入区域时为数据分配空间并将数据复制到设备内存，离开区域时将撤销数据空间。该子语用在 enter data 导语上的时候，数据被分配空间并复制到设备内存。

该子语可以缩写为 pcopyin. present 子语中对子数组的限制同样适用于本子语。

2.6.5.11 present_or_copyout 子语

在非共享内存加速器上，present_or_copyout 子语用来告诉编译器去检测变量列表中的每一个变量或数组是否已经存在于加速器内存中，就像有个 present 子语一样。

如果这些数据已经存在 程序行为和有 present 子语时一样。不会分配新的设备内存，没有数据在设备内存上移入、移出。

如果这些数据不在加速器上，程序的行为和带 copyin 子语时一样。在进入区域时为数据分配空间，退出区域时将数据复制回本地内存并撤销数据空间。

该子语可以缩写为 pcopyout. present 子语中对子数组的限制同样适用于本子语。

[1] 就是子数组的子数组。--译者注。

2.6.5.12 present_or_create 子语

在非共享内存加速器上，present_or_copycreate 子语用来告诉编译器去检测变量列表中的每一个变量或数组是否已经存在于加速器内存中，就像有个 present 子语一样。

如果这些数据已经存在，程序的行为和有 present 子语时一样。不会分配设备内存。

如果这些数据不在加速器上，程序的行为和带 create 子语时一样。该子语用在一个数据构件或计算构件上的时候，进入区域时为数据分配设备内存，离开区域时撤销数据的设备内存。该子语用在 enter data 导语上的时候，在设备内存中为数据分配空间。

这个子语可以缩写为 pcreate. present 子语中对子数组的限制同样适用于本子语。

2.6.6 host_data 构件

概要

host_data 构件使设备上数据的地址在主机上可用。

语法

C 和 C++中，OpenACC host_data 导语[1]的语法是

```
#pragma acc host_data 子语列表  换行
    结构化块
```

Fortran 中，语法是

```
!$acc host_data 子语列表
    结构化块
!$acc end host_data
```

这里只有一个可用的子语：

```
use_device( 变量列表 )
```

描述

本构件使数据的设备地址在主机代码里可用。

2.6.6.1 use_device 子语

use_device 子语告诉编译器，对变量列表中的任何变量或数组，在构件内的代码里使用它们的设备地址。特别地，本子语可以用来传递变量或数组的设备地址，以优化那些用低层接口编写的例程。变量列表里的变量或数组必须已经存在于加速器内存中，这可以用数据区域或数据生存期包含本构件来实现。在共享内存加速上，设备地址可能与主机地址相同。

2.7 loop 构件

概要

OpenACC loop 导语作用于紧跟该导语的一个循环。loop 导语可以描述执行这个循

[1] 原文为 data directive，有误，应为 host_data directive，已更正。--译者注。

环的并行类型，还可以声明循环的私有变量、数组和归约操作。

语法

C 和 C++中，loop 导语的语法是

```
#pragma acc loop [子语表表] 换行
    for 循环
```

Fortran 中，loop 导语的语法是

```
!$acc loop [子语列表]
    do 循环
```

这里的子语是下列中的一个：

```
collapse(n)
gang [(gang 参数列表)]
worker[([num:]整数表达式)]
vector[([length:]整数表达式)]
seq
auto
tile(尺寸表达式列表)
device_type(设备类型列表)
independent
private(列表)
reduction(操作符:列表)
```

这里的 gang 参数是下列中的一个：

```
[num:] 整数表达式
static: 尺寸表达式
```

并且 gang 参数列表至多只能有一个 num 或 static 参数，这里的尺寸表达式是下列中的一个：

```
*
整数表达式
```

一些子语只能用在 pararllel 区域的上下文中，而另一些子语只能用在 kernels 区域的上下文中；详述见下文。

限制

- 仅有 collapse、 gang、 worker、 vector、 seq、auto 和 and tile 这几个子语可以跟在一个 device_type 子语后面。
- worker 和 vector 子语的参数整数表达式必须在 kernels 区域内保持不变。

2.7.1 collapse 子语

collapse 子语用来指定有多少层紧密嵌套的循环与 loop 构件相关联。collapse 子语的参数必须是一个常量正整数表达式。如果没有 collapse 子语，只有紧接着的循环才与 loop 导语相关联。

如果有一个以上的循环与 loop 构件关联，那么关联循环中的所有循环体都会按照剩余的子语来调度。与 collapse 子语相关联的所有循环的路程步数必须是可计算的，并在所有循环中保持不变。

编译器自行决定是否将导语的 gang、worker 或 vector 子语应用到每一个循环、线性化迭代空间。

2.7.2 gang 子语

在一个加速器 parallel 区域中，gang 子语要求将循环的迭代步分摊到为 parallel 构件创建的多个 gang 上，从而实现并行执行。一个带 gang 子语的 loop 构件将一个计算构件从 gang 冗余模式切换到 gang 分裂模式。gang 的数量由 parallel 构件控制，只允许使用 static 参数。循环的所有迭代步必须是数据独立的，但在 reduction 子语中指定的变量除外。除非在一个嵌套的 parallel 区域内或嵌套的 kernels 区域内，否则不允许一个带 gang 子语的 loop 区域包含另外一个带 gang 子语的 loop 区域。

在一个加速器 kernels 区域中，gang 子语指明，关联循环的迭代步需要在为内核创建的所有 gang 上并行执行。如果指定一个不带关键字的参数，或者指定一个放在 num 关键字后面的参数，那么该参数指明使用多少个 gang 来执行本循环的迭代步。除非在一个嵌套的 parallel 区域内或嵌套的 kernels 区域内，否则不允许一个带 gang 子语的 loop 区域包含另外一个带 gang 子语的 loop 区域。

除非出现 static 参数，否则循环的迭代步对各个 gang 的调度方式不确定。如果出现一个带星号的 static 参数，那么编译器将自行决定块的尺寸。循环的所有迭代步以选定的尺寸分割成多块，所有 gang 从零号块开始轮流获得若干块。同一个 parallel 区域内，循环步数相同的两个 gang 循环，如果使用带相同参数的 static 子语，那么循环步分配到 gang 上的方式相同。同一个 kernels 区域内，循环步数相同的两个 gang 循环，如果使用相同数量的 gang，static 子语的参数也相同，那么循环步分配到 gang 上的方式相同。

2.7.3 worker 子语

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。带 worker 子语的 loop 构件将一个 gang 由 worker 单独模式切换为 worker 分裂模式。与 gang 对比，worker 子语先激活额外的 worker 层次并行，然后将循环分摊到这些 worker 上。不允许使用参数。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。除非在一个嵌套的 parallel 区域内或嵌套的 kernels 区域内，否则不允许一个带 worker 子语的 loop 区域包含另外一个带 gang 子语的 loop 区域。

在一个加速器 kernels 区域内，worker 子语要求，关联循环的迭代步在为内核创建的一个 gang 的所有 worker 上并行执行。如果指定了一个参数，本子语指明每个 gang 中使用多少个 worker 来执行每个循环的迭代步。除非在一个嵌套的 parallel 区域内或嵌套的 kernels 区域内，否则不允许一个带 worker 子语的 loop 区域包含另外一个带 gang 子语的 loop 区域。

在任何 worker 继续处理本循环后面的代码之前，所有 worker 都已将各自承担的迭代步执行完毕。

2.7.4 vector 子语

在一个加速器 parallel 区域内，vector 子语指明，关联循环的迭代步以向量模式或 SIMD 模式执行。一个带有 vector 子语的 loop 构件将一个 worker 由向量单独模式切换为向量分裂模式。与 worker 子语相似，vector 子语先激活额外的向量层次并行，然后将循环的迭代步分摊到这些向量通道上。向量的长度由本子语指定或由编译器为本 parallel 区域自动选择。除非在一个嵌套的 parallel 区域内或嵌套的 kernels 区域内，否则不允许一个带 vector 子语的 loop 区域包含另外一个带 gang 子语、worker 子语或 vector 子语的循环。

在一个加速器 kernels 区域，vector 子语指明，关联循环的迭代步以向量模式或 SIMD 模式执行。如果指定了一个参数，迭代步将以这个长度的向量执行；如果没有指定参数，编译器将自行选择一个合适的向量长度。除非在一个嵌套的 parallel 区域内或嵌套的 kernels 区域内，否则不允许一个带 vector 子语的 loop 区域包含另外一个带 gang 子语、worker 子语或 vector 子语的循环。

在任何 vector 继续处理本循环后面的代码之前，所有 vector 都已将各自承担的迭代步执行完毕。

2.7.5 seq 子语

seq 子语指明关联循环需要在加速器上串行地执行。本子语将阻止所有的自动并行化、向量化。

2.7.6 auto 子语

auto 子语指明，编译器需要决定是否对这个循环实施 gang、worker 或 vector 并行化。受内、外层循环上带有 gang、worker 或 vector 子语的 loop 导语的影响，编译器可以实施的并行类型可能会受到限制。该子语本身并不会告诉编译器循环的迭代步都数据独立的，一般情况下编译器不支实施任何并行化动作，除非这个循环带有 independent 子语、或者因处于 parallel 构件之间而具有隐式独立属性、或者编译器可以分析这个循环并认定它的迭代步都是数据独立的。kernels 构件内，不带 gang、worker、vector 或 seq 子语的 loop 导语，与带有 auto 子语同样对待[1]。

2.7.7 tile 子语

tile 子语指明，编译器应将嵌套循环里的每个一个循环剖分成两个循环，一个外层 tile 循环，和一个内层 element 循环。tile 子语的参数是一个多分片尺寸列表，每一个分片尺寸都可以是一个常量正整数表达式，或是一个星号。如果列表中有 n 个分片尺寸，loop 导语后面必须紧跟着 n 个紧密嵌套的循环。*尺寸表达式列表*中的第一个参数对应 n 个关联循

[1] 此种情形下，auto 是缺省子语。--译者注。

环中最内层的循环，最后一个参数对应最外层的关联循环。如果分片尺寸被指定为星号，那么编译器将自主选择一个合适的值。嵌套循环内的每一个循环都将被分割为两个循环，一个外层循环和一个内层循环。element 循环的路程步数受限于*尺寸表达式列表*中的相应分片尺寸。tile 循环的将被调整到所有 element 循环的外层，而 element 循环将处于所有 tile 循环的内层。

如果 `loop` 导语上带有 `vector` 子语，那么这个 vector 子语会作用在 element 循环上。如果 loop 导语上带有 `gang` 子语，那么这个 `gang` 子语会作用到 tile 循环。`loop` 导语上带有 `worker` 子语时，如果没有出现 `vector` 子语，那么这个 `worker` 子语会作用在 element 循环上，否则作用在 tile 循环上。

2.7.8 device_type 子语

`device_type` 子语的详细描述在 2.4 节*特定设备的子语*。

2.7.9 independent 子语

在一个 `kernels` 构件内，`independent` 子语告诉编译器该循环的迭代步是彼此数据独立的。因此允许生成并行执行这些迭代步的代码，且不需要同步。在一个 `parallel` 构件内，所有不带 `seq` 子语的 `loop` 导语都暗含一个 `independent` 子语。

限制

- 一个 `kernels` 构件里面，如果有一个迭代步向一个变量（`reduction` 子语中的变量除外）或数组写数据，与此同时有另外一个迭代步在读或写相同的变量或数组元素，那么在循环上使用 `independent` 子语就是一个编程错误。

2.7.10 private 子语

`loop` 导语上的 `private` 子语指明，对*列表*中的每一项，每个线程都要创建一个副本，一个线程可能执行关联循环的中一个迭代步或多个迭代步。不在一个 private 子语中，也不具有预定私有属性却在循环中被引用的变量，执行循环迭代步的线程不会将它私有化。

2.7.11 reduction 子语

`reduction` 子语列出一个操作符和一个或多个标量变量。对每一个归约变量，执行关联循环迭代步的每一个线程都会创建一个私有副本，并根据操作符进行初始化；详情见 2.45.11 节 reduction *子语*中的表格。在循环结束处，用指定的归约操作符将每一个线程的值组合起来，得到的结果在 `parallel` 区或 `kernels` 区域结束时存入原始变量。

在 `parallel` 区域中，如果 `reduction` 子语用在带有 `vector` 或 `worker` 子语（并且不带 `gang` 子句）的 `loop` 构件上，并且标量变量出现在 `parallel` 构件的 `private` 子语中，那么该标量的私有副本的值将在循环退出时更新。如果这个标量变量没有出现在并行构件的某个 `private` 子语中 ,或者这个 `reduction` 子语用在一个带 `gang` 子语的循环上，那这个标量的值直到该 `parallel` 区域结束时才更新。

2.8 cache 导语

概要

cache 导语可以出现在一个循环内的顶部。它指定哪些数组元素或子数组需要为本循环体而预取到最高层级的缓存中。

语法

C 和 C++中，cache 导语的语法是

```
#pragma acc cache(变量列表) 换行
```

Fortran 中，cache 导语的语法是

```
!$acc cache(变量列表)
```

变量列表中的条目必须为单个数组元素或者简单子数组。C 和 C++中，简单子数组就是一个数组名字后跟一个标明数组范围的方括号，方括号内有起始下标和长度，例如

```
arr[下界:长度]
```

这里的下界可以是一个常量、循环内不变量、循环索引变量加减一个常量或加减一个循环内不变量，长度是一个常量。

Fortran 中，简单子数组就是一个数组名后跟一个圆括号，圆括号内是一个用逗号分隔的范围列表，范围用下界下标与上界下标指定，例如

```
arr(下界:上界,下界2:上界2)
```

下界必须是一个常量、循环内不变量、循环索引变量加减一个常量或加减一个循环内不变量；更多地，上界与相应下界的差值必须是一个常量。

2.9 组合导语

概要

如果 parallel 构件或 kernels 构件内立即嵌套一个 loop 导语，就可以组合起来简写为 OpenACC 的 parallel loop 构件或 kernels loop 构件。它的含义等同于在 parallel 导语或 kernels 导语内显式地包含一个 loop 导语。任何可以用在 parallel 或 loop 导语上的子语都可以用在 parallel loop 导语上。任何可以用在 kernels 或 loop 导语上的子语都可以用在 kernels loop 导语上。

语法

C 和 C++中，parallel loop 导语的语法是

```
#pragma acc parallel loop [子语列表] 换行
    for 循环
```

Fortran 中，parallel loop 导语的语法是

```
!$acc parallel loop [子语列表]
    do 循环
[!$acc end parallel loop]
```

紧接着导语的循环就是本导语的关联结构块。parallel 或 loop 的任意子语，只要可以用在 parallel 区域，就可以出现在这里。

C 和 C++中，kernels loop 导语的语法是

```
#pragma acc kernels loop [子语列表] 换行
    for 循环
```

Fortran 中，kernels loop 导语的语法是

```
!$acc kernels loop [子语列表]
    do 循环
[!$acc end kernels loop]
```

紧接着导语的循环就是关联结构块。kernels 或 loop 的任意子语，只要可以用在 kernels 区域，都可能出现在这里。

限制

- parallel、kernels、loop 构件的限制同样适用。

2.10 atomic 导语

概要

一个 atomic 构件保证指定的存储位置被不受干扰地访问或更新，防止多个 gang、worker 或向量线程同时访问同一位置，避免得到不确定的值。

语法

C 和 C++中，atomic 构件的语法是：

```
#pragma acc atomic [ atomic 子语 ]  换行
    表达式语句
```

或：

```
#pragma acc atomic update 换行
    结构化块
```

这里的 atomic 子语是 read、write、update 或 capture 中的一个。表达式语句的形式是下列中的一个：

如果 atomic 子语为 read:

```
v = x;
```

如果 atomic 子语为 write:

```
x = expr;
```

如果 atomic 子语为 update 或者为空：

```
x++;
x--;
++x;
```

```
--x;
x binop= expr;
x = x binop expr;
x = expr binop x;
```

如果 atomic 子语为 capture:

```
v = x++;
v = x--;
v = ++x;
v = --x;
v = x binop= expr;
v = x = x binop expr;
v = x = expr binop x;
```

结构化块的形式为下列之一：

```
{v = x; x binop= expr;}
{x binop= expr; v = x;}
{v = x; x = x binop expr;}
{v = x; x = expr binop x;}
{x = x binop expr; v = x;}
{x = expr binop x; v = x;}
{v = x; x = expr;}
{v = x; x++;}
{v = x; ++x;}
{++x; v = x;}
{x++; v = x;}
{v = x; x--;}
{v = x; --x;}
{--x; v = x;}
{x--; v = x;}
```

在前面的表达式中：

- x 和 v (如果用到了)都是标量类型的左值表达式。
- 在一个原子区域执行期间，在语法上多次出现的 x 必须指向相同的存储位置。
- v 和 *expr* (如果用到了)都不能访问 x 指向的存储位置。
- x 和 *expr*(如果用到了) 都不能访问 v 指向的存储位置。
- *expr* 是一个标量类型的表达式。
- *binop* 是 +、*、-、/、&、^、|、<<或>>中的一个。
- *binop*、*binop*=、++、和--不能是重载运算符。
- 表达式 x *binop expr* 必须在数学上等价于 x *binop* (*expr*). 如果 *expr* 中操作符的优先级高于 *binop*，或者在 *expr* 或 *expr* 子表达式的两边使用圆括号，就能够满足这个要求。
- 表达式 *expr binop* x 必须在数学上等价于(*expr*) *binop* x. 如果 *expr* 中操作符的优先级等于或高于 *binop*，或者在 *expr* 或 *expr* 子表达式的两边使用圆括号，就能够满足这个要求。
- 在那些 x 多次出现的形式中，x 的求值次数没有指定。

Fortran 中，atomic 构件的语法是：

```
!$acc atomic read
    capture 表达式
```

```
!$acc end atomic]
```

或

```
!$acc atomic write
    write 表达式
!$acc end atomic]
```

或

```
!$acc atomic [update]
    update 表达式
!$acc end atomic]
```

或

```
!$acc atomic capture
    update 表达式
    capture 表达式
!$acc end atomic
```

或

```
!$acc atomic capture
    capture 表达式
    update 表达式
!$acc end atomic
```

或

```
!$acc atomic capture
    capture 表达式
    write 表达式
!$acc end atomic
```

这里的 write 表达式具有如下形式（如果子语是 `write` 或 `capture`）：

`x = expr`

这里的 capture 表达式具有如下的形式（如果子语是 `write` 或 `capture`）：

`v = x`

这里的 update 表达式具有如下的形式（如果子语是 `update`、`capture`，或没有子语）：

`x = x 操作符 expr`

`x = expr 操作符 x`

`x = 内置过程名字(x,表达式列表)`

`x = 内置过程名字(表达式列表, x)`

在前面的表达式中：

- `x` 和 `v` (如果用到了)都是内置类型的标量变量。
- `x` 决不能是动态分配的变量。
- 在一个原子区域执行期间，在语法上多次出现的 `x` 必须指向相同的存储位置。
- `v` 和 *expr* (如果用到了)都不能访问 `x` 指向的存储位置。
- `x` 和 *expr*(如果用到了) 都不能访问 `v` 指向的存储位置。
- *expr* 是一个标量类型的表达式。
- 表达式列表是一个逗号分隔的非空标量表达式列表。如果内置过程名字是 `iand`、`ior` 或 `ieor`，那么必须有且只能有一个表达式出现在表达式列表里。
- 内置过程名字是 `max`、`min`、`iand`、`ior` 或 `ieor` 中的一个。操作符是+、*、-、/、`.and.`、`.or.`、`.eqv.`或`.neqv.`中的一个。

- 表达式 x *binop expr* 必须在数学上等价于 x *binop* (*expr*). 如果 *expr* 中操作符的优先级高于 *binop*，或者在 *expr* 或 *expr* 子表达式的两边使用圆括号，就能够满足这个要求。
- 表达式 *expr binop* x 必须在数学上等价于(*expr*) *binop* x. 如果 *expr* 中操作符的优先级等于或高于 *binop*，或者在 *expr* 或 *expr* 子表达式的两边使用圆括号，就能够满足这个要求。
- *内置过程名字*不能是非内置程序。
- *操作符*必须是内存操作符，不能是用户自定义的运算符。所有的赋值必须是内置赋值。
- 在那些 x 多次出现的形式中，x 的求值次数没有指定。

一个带有 read 子语的 atomic 构件对 x 指向的位置做一次强制原子读。一个带有 write 子语的 atomic 构件对 x 指向的位置做一次强制原子写。

一个带有 update 子语的原子构件，使用指定的操作符或固有操作符强制原子更新 x 指向的位置。注意，没有子语等价于 atomic update. 对 x 指向的位置，仅有 read 和 write 能够相互原子地实施。对于需要读或读的 x 指向的位置，*表达式*或*表式达列表*不必原子地求值。

一个带有 capture 子语的原子构件，使用指定的操作符或固有操作符强制原子更新 x 指向的位置，与 atomic update 相比，它还能捕捉 x 指向位置的初始值或最终值。写入 v 指向位置的值究竟是 x 指向位置的初始值还是最终值，依赖于原子构件结构化块或语句的形式。对 x 指向的位置，仅有 read 和 write 能够相互原子地实施。无论是*表达式*还是*表达式列表*求值，以及 v 指向位置的写操作，都不必原子地实施。

对所有形式的原子构件，这些原子构件的两个或两个以上的任意组合，都会迫使互斥地访问 x 指向的位置。为避免数据竞争，所有对 x 指向位置访问，如果有可能并行发生，都必须用一个原子构件保护起来。

在原子区域外访问存储位置 x，如果在原子区域之外也访问同一个存储位置 x，那么不保证两个访问能够独占地执行，即使这些访问发现在一个归约子语的执行期间。

如果 x 指向的存储位置没有按尺寸对齐（即，如果 x 的对齐字节数不是 x 尺寸的整数倍），那么原子区域的形为由编译器定义。

限制

以下限制适用于 atomic 构件：

- 整个程序内，对 x 指向位置的所有原子访问要求有相同的类型和相同的类型参数。
- x 指向存储位置的尺寸必须小于或等于原生原子操作符的最大可用尺寸。

2.11 declare 导语

概要

一个 declare 导语用在 Fortran 子例程、函数或模块的声明部分，或者跟在 C/C++变

量声明之后。它指定的变量或数组需要在设备内存上开辟空间，在一个函数、子例程或程序的隐式数据区域的生存期之内，这些变量一直驻留设备内存中。它还能指定是否需要在隐式数据区域的入口处将数据值从主机传递到设备内存，是否需要在隐式数据区域的出口处将数据值从设备传递到主机内存。这个导语为变量或数组创建一个可见设备副本。

语法

C 和 C++中，declare 导语的语法是：

```
#pragma acc declare 子语列表 换行
```

Fortran 中，declare 导语的语法是：

```
!$acc declare 子语列表
```

这里的子语是下列中的一个：

```
copy(变量列表)
copyin(变量列表)
copyout(变量列表)
create(变量列表)
present(变量列表)
present_or_copy(变量列表)
present_or_copyin(变量列表)
present_or_copyout(变量列表)
present_or_create(变量列表)
deviceptr(变量列表)
device_resident(变量列表)
link( 变量列表 )
```

本导语所在函数、子例程或程序的隐式数据区域是其关联区域。如果本导语出现在 Fortran 模块的子程序中，或者出现在 C/C++全局作用域中，那么整个程序的隐式数据区域就是其关联区域。其它情形下，这些子语的使用效果，与显式地使用带有相同子语的 data 构件完全相同。数据子语的详细讲述在 2.6.5 节数据子语。

限制

- 一个函数、子例程、程序或模块中，一个变量或数组在所有 declare 导语的所有子语中只能出现至多一次。
- 子数组不允许出现在 declare 导语中。
- Fortran 中，不定尺寸形参数组[1]不能出现在 declare 导语中。
- Fortran 中，可以指定指针数组，但设备内存中不再保留指针的指向关系。
- 在 Fortran 模块的声明部分，只允许出现 create、copyin、device_resident 和 link 子语。
- 在 C 或 C++的全局作用域中，只允许出现 create、copyin、deviceptr、

[1]见 2.6.5.1 节脚注。

device_resident 和 link 子语。

- C 和 C++的外部变量只能出现在 declare 导语上的 create、copyin、deviceptr、 device_resident 和 link 子语中。

2.11.1 device_resident 子语

概要

device_resident 子语明确要求指定的变量需要在加速器设备内存上开辟空间，且不在主机内存上开辟空间。参数列表里的名字可以是变量名或数组名，可以是夹在斜杠之间的 Fortran 公共块名字；不能是子数组。主机不能访问 device_resident 子语中的变量。device_resident 中的全局变量或公共块的加速器数据生存期贯穿程序的整个执行过程。

Fortran 中，如果变量具有 allocatable 属性，当主机程序执行到这个变量的 allocate 或 deallocate 语句时，这个变量的存储空间将在加速器设备内存上开辟和释放。如果变量具有 Fortran 指针属性，主机可以为它在加速器设备内存中分配和释放空间；如果指针赋值语句的右手边变量出现在一个 device_resident 子语中，那么它也可以出现在指针赋值语句的左手边。

Fortran 中， device_resident 子语的参数可以是夹在斜杠之间的公共块名字；在这种情况下，公共块的所有声明都必须有一个匹配的 device_resident 子语。在这种情况下，公共块将在设备内存中静态分配空间，而不是在主机内存中。这个公共块将对加速器例程可用；参见 2.13 节计算区域内的过程调用。

在 Fortran 模块声明区，device_resident 子语中的变量或数组将对加速器例程可用。

在 C 或 C++全局作用域内，device_resident 子语中的变量或数组将对加速器例程可用。一个 C/C++外部变量可以出现在 device_resident 子语中，仅当变量的实际声明和所有外部声明后都跟着 device_resident 子语。

2.11.2 link 子语

在加速器例程中引用的大型全局主机静态数据必须有一个设备上的动态数据生存期，link 子语用于这些数据。link 子语指明，只有那些指定名字的变量需要在加速器内存创建一个全局链接。主机端数据结构仍然被静态分配并且全局可用。仅当全局变量出现在 data 构件、计算构件或加速器例程的数据子语中时，才会分配设备数据内存。link 子语的实参必须是全局数据。C 或 C++中，link 子语必须出现在全局作用域上，或者实参必须是外部变量。Fortran 中，link 子语必须出现在模块的声明部分，或者实参必须是夹在斜杠之间的公共块名字。无论在任何地方，只要在一个数据子语、计算构件或加速器例程中显式或隐式地用到了全局变量或公共块变量，都必须保证 declare link 子语可见。这些全局变量或公共块变量可以被用到加速器例程中。link 子语中指定的全局变量或公共块变量的加速器数据生存期可以是使用一个数据子语为这些变量或公共块分配空间的数据区域，也可以是从为该数据分配空间的 enter data 导语开始，直到撤销数据的 exit data 导语结束，或者直到本程序结束。

2.12 可执行导语

2.12.1 update 导语

概要

在加速器数据的生存期之内，update 导语从设备内存中取出相应的值，用来更新本地变量、数组的部分元素或全部元素；或者从本地内存中取出相应的值，用来更新设备变量、数组的部分元素或全部元素。

语法

C 和 C++中，update 导语的语法是：

```
#pragma acc update 子语列表  换行
```

Fortran 中，update 导语的语法是：

```
!$acc update 子语列表
```

这里的子语是下列中的一个：

```
async[(整数表达式)]
wait[(整数表达式列表)]
device_type(设备类型列表)
if(条件)
self(变量列表)
host(变量列表)
device(变量列表)
```

update 子语的参数变量列表是一串用逗号分隔的变量名、数组名或限定范围的子数组。同一个数组的多个子数组可以出现在同一个导语的同一个子语的变量列表内，也可以出在同一个导语的不同子语的参数列表内。对 update self，update 子语的效果是将加速器设备内存中的数据复制到本地内存；对 update device，update 子语的效果是将本地内存中的数据复制到加速器设备内存。数据的更新顺序与它们在子语中出现的顺序相同。出现在 host 或 device 子语的变量和数组必须有一份设备副本。self 子语、host 子语和 device 子语必须至少出现一个。

2.12.1.1 self 子语

对一个非共享内存加速器，self 子语明确要求将变量列表里的变量、数组或子数组从加速器设备内存复制到本地内存。如果加速器与遭遇的线程共享同一个内存，没有动作。

2.12.1.2 host 子语

host 子语是 self 子语的同义词。

2.12.1.3 device 子语

对一个非共享内存加速器，device 子语明确要求将变量列表里的变量、数组或子数组从本地内存复制到加速器设备内存。如果加速器与遭遇的线程共享同一个内存，没有动作。

2.12.1.4 if 子语

`if` 子语是可选项；没有 `if` 子语的时候，编译器将生成无条件更新的代码。有 `if` 子语的时候，编译将生成有条件更新的代码，只有当*条件*的值在 C/C++中为非零或在 Fortran 中为`.true.`时才进行更行。

2.12.1.5 async 子语

`async` 子语是可选项；详见 2.14 节*异步形为*。

2.12.1.6 wait 子语

`wait` 子语是可选项；详见 2.14 节*异步形为*。

限制

- `update` 导语是可执行的。在 C 和 C++中，它不准出现在紧跟 *if*、*while*、*do*、*switch*、*label* 的语句位置上；在 Fortran 中，它不准出现在紧跟逻辑 *if* 的语句位置上。
- 出现在 `update` 导语*列表*中的变量、数组必须有一个设备副本。
- 只有 `async` 和 `wait` 子语可以跟在一个 `device_type` 子语后面。
- 至多可以出现一个 `if` 子语。Fortran 中，*条件*的求值结果必须是一个标量逻辑值；C 或 C++中，条件的求值结果必须是一个标量整数值。
- 可以指定不连续的子数组。更新不连续区域时，可以每个连续的子区域传输一次，也可将所有不连续数据打包一次完成传输，然后解包。编译器自主选择采取哪种更新方式。
- C 和 C++中，可以指定结构或类的成员，包括成员的子数组。不能指定结构或类子数组的成员。
- C 和 C++中， 如果一个结构成员使用某个子数组记号，那么该结构成员的父类型就不能再使用子数组记号。
- Fortran 中，可以指定派生类型变量的成员，包括一个成员的子数组。不能指定派生类型子数组的成员。
- Fortran 中，如果用到了一个派生类型成员的数组或子数组记号，该派生类型成员的父类型就不能再使用数组或子数记号。

2.12.2 wait 导语

详见 2.14 *异步行为*。

2.12.3 enter data 导语

详见 2.6.4 节 enter data 和 exit data 导语。

2.12.4 exit data 导语

详见 2.6.4 节 enter data 和 exit data 导语。

2.13 计算区域内的过程调用

本节描述如何为一个加速器编译例程，及计算区域内的过程调用是如何编译的。

2.13.1 routine 导语

概要

routine 导语告诉编译器，将给定的过程编译得既能在主机上运行，也能在加速器上运行。 在一个带有过程调用的文件或例程中，当调用过程时，routine 导语告诉编译器这个被调用过程的属性。

语法

C 和 C++中，routine 导语的语法是：

```
#pragma acc routine 子语列表  换行
#pragma acc routine (名字) 子语列表  换行
```

C 和 C++中，不带名字的 routine 导语可以出现在一个函数定义的紧邻上方，或出现在一个函数定义的上方且作用于紧跟着的函数或原型。在允许函数原型出现在的任何地方，都可以出现带名字的 routine 导语，但必须出现在任何函数定义或函数使用的前面。

Fortran 中，routine 导语的语法是：

```
!$acc routine 子语列表
!$acc routine (名字) 子语列表
```

Fortran 中，不带名字的 routine 导语可以出现在一个子例程或函数内的参数说明区域，或者接口块之内的子例程和函数的接口体之内，并应用于包含的子例程和函数。

带着 routine 导语为一个加速器编译的 C/C++函数或函数子程序称为加速器例程。这里的子语是下列之一：

```
gang
worker
vector
seq
bind(名字)
bind(字符串)
device_type(设备类型列表)
nohost
```

限制

- 仅有 gang、worker、vector、seq 和 bind 子语可以跟在一个 device_type 子语后面。
- C 和 C++中， 在 routine 导语起作用的函数中不支持函数静态变量。
- Fortran 中，在 routine 导语起作用的子程序中，显式或隐式地带有 save 属性的变量都不支持。

2.13.1.1 gang 子语

gang 子语指明，本过程可以包含、调用另一个过程，内层过程里包含一个带 gang 子语的循环。调用本过程的代码必须以 gang 冗余模式执行，并且所有的 gang 都必须执行这个调用。例如 ,带有一个 gang 子语的循环里面不能调用一个带有 routine gang 的过程。每个设备类型只能指定 gang、worker、vector、seq 其中之一。

2.13.1.2 worker 子语

worker 子语指明 ,本过程可以包含、调用另一个过程 ,内层过程里包含一个带 worker 子语的循环，但不能包含或调用具有 gang 子语的循环的过程。这个过程里面带有 auto 子语的循环，究竟是以 worker 模式执行还是以 vector 模式执行，可以由编译器来选择决定。调用这个过程的代码必须以 worker 单独模式执行，尽管它可以在 gang 冗余或 gang 分裂模式之中。例如 ,带有一个 gang 子语的循环里面可以调用一个带有 routine worker 导语的过程，但带有一个 worker 子语的循环里面不可以调用一个带有 routine worker 导语的过程。每个设备类型只能指定 gang、worker、vector、seq 其中之一。

2.13.1.3 vector 子语

vector 子语指明 ,本过程可以包含、调用另一个过程 ,内层过程里包含一个带 vector 子语的循环，但不能包含或调用具有 gang 或 worker 子语的循环的过程。对这个过程里带有 auto 子语的循环，编译器可以选择以 vector 模式执行，但不能以 worker 模式执行。调用这个过程的代码必须以向量单独模式执行，尽管它可以在 gang 冗余或 gang 分裂模式之中，也可以在 worker 单独模式或 worker 分裂模式之中。举个例子，带有一个 gang 或 vector 子语的循环里面可以调用一个带有 routine vector 导语的过程，但带有 vector 子语的循环不能调用。每个设备类型只能指定 gang、worker、vector、seq 其中之一。

2.13.1.4 seq 子语

seq 子语指明，这个过程不包含也不调用另一个包含 gang、worker 或 vector 子语的循环的过程。这个过程里，带 auto 子语的循环将以 seq 模式执行。可以在任何模式中调用这个过程。每个设备类型只能指定 gang、worker、vector、seq 其中之一。

2.13.1.5 bind 子语

bind 子语指定编译或调用这个过程时使用的名字。如果这个名字被指定为标识符，它将被当作语言本身的名字来编译和调用。如果这个名字被指定为字符串，这个将字符将被用作这个过程的名字。

2.13.1.6 device_type 子语

device_type 子语的描述在 2.4 节特定设备的子语。

2.13.1.7 nohost 子语

nohost 子语告诉编译器不要为这个过程编译主机端版本。这个过程的所有调用必须出现在加速器计算区域之内。如果这个过程被别的过程调用，那么这些主调过程必须具有一个匹配的带有 nohost 子语的 routine 导语。

2.13.2 全局数据访问

C 或 C++中的全局变量、全局数组、文件静态变量、文件静态数组、外部变量、外部数组，Fortran 的模块变量、模块数组、共公块变量、共公块数组，它们被用在加速器例程中时，必须出现在带有 create、copyin、device_resident 或 link 子语的 declare 导语中。如果数据出现在 device_resident 中，那么该过程的 routine 导语必须包含 nohost 子语。如果数据出现在一个 link 子语中，那么该数据必须通过出现在数据构件、计算构件或 enter data 之中来获得一个活动的加速器数据生存期。

2.14 异步行为

本节描述 async 子语，描述那些使用异步数据移动和异步计算构件的程序的形为。

2.14.1 async 子语

async 子语可以出现在 parallel 或 kernels 构件上，也可以出现在 enter data、exit data、update 或 wait 导语上。在所有情况下，async 子语是可选项。没有 async 子语的时候，在执行任何后续代码之前，本地线程会一直等待直至计算构件结束或数据操作完成；用在 wait 导语上时，等待直至相应异步活动队列上的所有操作都已完成。如果有 async 子语，在本地线程继续执行构件或导语后面代码的同时，parallel 或 kernels 区域或数据操作可以被异步地处理。

async 子语可以有一个*异步参数*，这个*异步参数*是一个非负标量整数表达式（C/C++中的 int，Fortran 中的 integer），或者是一个下文定义的特殊*异步值*。带有一个负值*异步参数*，但不带下文定义的特殊*异步值*的 async 子语的形为由编译器定义。*异步参数*的值可以用在一个 wait 导语、wait 子语或多个运行时例程中，用以检测或等待操作的完成。

C 和 Fortran 头文件中及 Fortran openacc 模块中均定义了两个特殊*异步值*。他们值为负，以免与用户定义的非负*异步参数*相冲突。一个带有*异步参数* acc_async_noval 的 async 子语的作用与不带参数的 async 子语的作用相同。一个带有*异步参数* acc_async_sync 的 async 子语的作用与不出现 async 子语的作用相同。

如果使用了*异步参数*，那么任何操作的*异步值*都是*异步参数*的值；如果 async 子语没有值，那么任何操作的*异步值*都是 acc_async_noval；如果 async 子语没有出现，那么任何操作的*异步值*都是 acc_async_sync. 如果设备支持一个或更多设备活动队列，那么异步值用于选择将操作压入哪个队列。设备属性和编译器共同决定实际上支持多少个活动队列，及异步值如何映射到实际活动队列上。具有相同*异步值*的两个异步操作将被压入同一个活动队列，因而它们的执行顺序与它们遭遇本地线程的顺序相同。两个具有不同*异步值*的异步操作可能会被压入不同的活动队列，因而它们可以在设备上以任意的相对顺序执行。如果有两个或两个以上线程执行和共享同一个加速器设备，具有相同*异步值*的两个异步操作将被压入同一个活动队列，但除非线程间保持相互同步，这些操作可以以任何顺序压入队列，进而可以任意的顺序在设备上执行。

2.14.2 wait 子语

wait 子语可以出现在 parallel 或 kernels 构件上，或者 enter data、exit data、update 导语上。在所有情况下，wait 子语都是可选项。没有 wait 子语时，在设备上，关联的计算操作或更新操作会被立即压入队列、启动或执行。如果 wait 子语有一个实参，那么这个实参必须是由一个或多个*异步值*组成的列表。计算、数据或更新操作可能不会被启动或执行，直到本线程在这个位置前压入关联异步设备活动队列的所有操作均已完成。一个合法的实现是，本地线程等待所有关联的异步设备活动队列完成。另一个合法的实现是，本地线程将计算操作和更新操作以这样的方式压入队列：直到所有被压入关联异步设备活动的队列的操作都已完成，本操作才开始。

2.14.3 wait 导语

概要

wait 导语引发本地线程等待异步操作的完成，例如一个加速器 parallel 或 kernels 区域，或者一个 update 导语，或者引发一个设备活动队列与一个或多个活动队列间的同步。

语法

C 和 C++中，wait 导语的语法是：

```
#pragma acc wait [(整数表达式列表)] 子语列表 换行
```

Fortran 中，wait 导语的语法是：

```
!$acc wait [(整数表达式列表)] 子语列表
```

这里的子语是：

```
async [( 整数表达式)]
```

如果出现了等待参数，那么它必须是一个或多个*异步参数*。

如果没有等待参数与没有 async 子语，那么本地线程将等待，直至本线程压入任何设备活动队列中的操作均已完成。

如果有一个或多个*整数表达式*并且没有 async 子语，那么本地线程将等待，直至本线程压入每一个设备活动队列中的操作均已完成。

如果有两个或更多个线程执行和共享同一个加速器设备，一个不带 async 子语的 wait 导语将引发本地线程等待，直至在此之前压入队列的所有相关异步活均已完成。保证其它线程压入队列的操作已经完成，要求在那些线程间进行额外的同步操作。

如果有一个 async 子语，不可以在 async 设备活动队列上启动或执行新的操作，直至本线程使用这个等待参数在此点之前压入异步活动队列的所有操作都已经完成。一个合法的实现是，本地线程等待所有的相关异步设备活动队列。另一个合法的实现是，本线程以这样的方式压入一个异步操作：不开始新的操作，直至被压入相关异步设备活动队列的操作均已完成。

第3章　运行时库

本章讲述供程序员使用的 OpenACC 运行时库例程。在不支持 OpenACC 接口的系统上使用这些例程可能会影响移植性。利用预处理变量_OPENACC 进行条件编译可以保持移植性。

本章包括两部分：

- 运行时库的定义
- 运行时库例程

限制

Fortran 中，没有任何一个 OpenACC 运行时库例程可以在 PURE 或 ELEMENTAL 中调用。

3.1 运行时库的定义

对 C 和 C++，本章讲述的运行时库例程的原型保存在一个名为 openacc.h 的头文件中。所有的库例程都是用“C”连接的 extern 函数。这个文件中定义[1]：

- 本章中所有例程的原型。
- 这些原型中使用的所有数据类型，包括一个描述加速器类型的枚举类型。
- acc_async_noval 和 and acc_async_sync 的取值。

对 Fortran ,接口声明都保存在一个名为 openacc_lib.h 的 Fortran 包含文件中和一个名为 openacc 的模块文件中。这两个文件中定义：

- 本章中所有例程的接口。
- 整数参数 openacc_version，参数值为 *yyyymm*，其中 *yyyy* 和 *mm* 分别是年份和月份，表示所支持的加速器编程模型的版本。这个值与预处理变量_OPENACC 的值相同。
- 用以定义这些例程的整数参数长度的整数。
- 用以描述加速器类型的整数参数。
- acc_async_noval 和 and acc_async_sync 的取值。

很多例程都接受或返回一个表示加速器设备类型的值。C 和 C++中，表示设备类型值的数据类型是 acc_device_t ；Fortran 中，这个数据类型是 integer(kind=acc_device_kind)。具体用什么值表示设备类型由编译器决定，C 和 C++中，这些值在头文件 openacc.h 中列出；Fortran 中，这些值在 Fortran 包含文件 openacc_lib.h 和 Fortran 模块 openac 中列出。所有的编译器都支持四个值：acc_device_none, acc_device_default, acc_device_host 和 acc_device_not_host。若想了解其它的值，请查看编译器包含的相应文件，或者阅读编译器文档。acc_device_default 不会是任何函数的返回值；用作输入参数时，它告诉

[1] 编译器可能会在头文件添加额外的定义，例如 PGI 编译器就添加了一些自行定义的例程和宏。--译者注。

运行时库使用编译器的默认设备类型。

3.2 运行时库例程

在本节中，对 C 和 C++原型，指针类型 h_void*和 d_void*分别指向一个主机地址和设备地址，好像包含了下列定义：

```
#define h_void void
#define d_void void
```

acc_on_device 除外，这些例程只能在主机端使用。

3.2.1 acc_get_num_devices

概要

例程 acc_get_num_devices 返回主机上指定类型的加速器设备数量。

格式

C 或 C++：

```
int acc_get_num_devices( acc_device_t );
```

Fortran:

```
integer function acc_get_num_devices( devicetype )
integer(acc_device_kind) devicetype
```

描述

例程 acc_get_num_devices 返回主机上指定类型的加速器设备数量。输入参数说明对哪种类型的设备计数。

限制

- 加速器 parallel 或 kernels 区域里不能调用本例程。

3.2.2 acc_set_device_type

概要

例程 acc_set_device_type 告诉运行时环境使哪种类型的设备来执行加速器 parallel 区域和 kernels 区域。当编译器允许被编译程序在一种以上类型的加速器上运行时，这个例程很有用。

格式

C 或 C++：

```
void acc_set_device_type ( acc_device_t );
```

Fortran:

```
subroutine acc_set_device_type ( devicetype )
integer(acc_device_kind) devicetype
```

描述

在所有可用的设备类型中，例程 `acc_set_device_type` 告诉运行时环境使用哪种类型的设备。

限制

- 在加速器 `parallel` 区域、`kernels` 区域内不可调用本例程。
- 如果指定的加速器类型不可用，例程的行为将由编译器决定；特别地，程序可能中止。
- 如果某些加速器区域仅为某一种设备类型编译，那么以不同的设备类型调用本例程可能产生未定义的行为。

3.2.3 acc_get_device_type

概要

如果已经选定了一种设备类型，例程 `acc_get_device_type` 将告诉程序运行下一个加速器区域时使用的设备类型。当编译器允许将代码编译成能在一种以上加速器上运行的程序时，本例程有用。

格式

C 或 C++：

```
acc_device_t acc_get_device_type ( void );
```

Fortran:

```
function acc_get_device_type ()
integer(acc_device_kind) acc_get_device
```

描述

如果已经选定了一种设备类型，例程 `acc_get_device_type` 返回一个值，告诉程序运行下一个加速器区域时使用什么类型的设备。选定设备类型的方法可能是调用 `acc_set_device_type`、使用一个环境变量，或程序默认行为。

限制

- 在加速器 `parallel` 区域、`kernels` 区域内不可调用本例程。
- 如果还没有选定任何设备类型，返回值为 `acc_device_none`.

3.2.4 acc_set_device_num

概要

`acc_set_device_num` 告诉运行时环境使用哪一个设备。

格式

C 或 C++:

```
void acc_set_device_num( int, acc_device_t );
```

Fortran:

```
subroutine acc_set_device_num( devicenum, devicetype )
integer devicenum
integer(acc_device_kind) devicetype
```

描述

在指定类型的所有设备中，acc_set_device_num 告诉运行时库环境用哪一个设备。如果 devicenum 的值为负，运行时环境将恢复到编译器自行定义的默认行为。如果第二个参数是零，选定的设备编号将应用到所有的加速器类型。

限制

- 在加速器 parallel 区域、kernels 区域内不可调用本例程。
- 对指定的设备类型，如果 devicenum 的值大于等于 acc_get_num_devices 的返回值，那么例程行为由编译器定义。
- acc_set_device_num 会隐式调用 acc_set_device_type，使用设备类型作为参数。

3.2.5 acc_get_device_num

概要

acc_get_device_num 例程返回指定设备类型的一个设备编号，该设备将运行下一个加速器 parallel 或 kernels 区域。

格式

C 或 C++：

```
int acc_get_device_num( acc_device_t );
```

Fortran:

```
integer function acc_get_device_num( devicetype )
integer(acc_device_kind) devicetype
```

描述

acc_get_device_num 例程返回一个与指定类型设备编号相对应的整数，该设备将执行下一个加速器 parallel 区域或 kernels 区域。

限制

- 在加速器 parallel 区域、kernels 区域内不可调用本例程。

3.2.6 acc_async_test

概要

acc_async_test 例程检测所有关联的异步操作是否已经完成。

格式

C 和 C++：

```
int acc_async_test( int );
```

Fortran:

```
logical function acc_async_test( arg )
integer(acc_handle_kind) arg
```

实参必须是一个在 2.14.1 节 async 子语中定义的异步参数。如果这个值出现在一个或多个 async 子语中，并且所有这些异步操作都已经完成，那么在 C 或 C++中 acc_async_test 例程将返回一个非零值，在 Fortran 中，acc_async_test 例程的返回值是.true.。如果这些异步操作中有一些还没有完成，那么在 C 或 C++中 acc_async_test 例程将返回一个零值，在 Fortran 中，acc_async_test 例程的返回值是.false.。如果两个或更多个线程共享同一个加速器，那么仅当本线程发起的所有匹配异步操作都已经完成，acc_async_test 例程才返回零或.false.；但不保证由其它线程发起的所有匹配异操作都已经完成。

3.2.7 acc_async_test_all

概要

acc_async_test_all 例程检测所有异步操作是否已经完成。

格式

C 和 C++：

```
int acc_async_test_all( );
```

Fortran:

```
logical function acc_async_test_all( )
```

描述

如果所有异步操作都已经完成，acc_async_test_all 将在返回一个 C/C++中的非零值或 Fortran 中的.true.。如果某个异步操作尚未完成，acc_async_test_all 将返回 C/C++中的零值或 Fortran 中的.false.。如果两个或两个以上线程共享同一个加速器，仅当本线程发起的所有异步操作均已完成，acc_async_test_all 才返回非零值或.true.；不保证其它主机线程发起的异步操作均已完成。

3.2.8 acc_async_wait

概要

acc_async_wait 等待所有关联的异步操作完成。

格式

C 和 C++：

```
void acc_async_wait( int );
```

Fortran:

```
subroutine acc_async_wait( arg )
integer(acc_handle_kind) arg
```

描述

输入参数必须是一个异步参数，就像 2.14.1 节 async 子语中定义的那样。如果这个值出现在一个或更多个 async 子语中，acc_async_wait 将一直等待，直到最后一个异步操作完成才返回。如果两个或两个以上线程共享同一个加速器，本线程发起的所有匹配异步操作都完成以后，acc_async_wait 例程才返回；不保证其它线程发起的所有匹配异步操作都已完成。

3.2.9 acc_async_wait_async

概要

为了等待先前压入某个异步队列的操作，acc_async_wait_async 例程会在另一个异步队列压入一个等待操作。

格式

C 或 C++:

```
void acc_async_wait_async ( int, int );
```

Fortran:

```
subroutine acc_async_wait_async ( arg, async )
integer(acc_handle_kind) arg, async
```

描述

acc_async_wait_async 例程等价于带一个 async 子语的 wait 导语。实参必须是一个异步参数，就像 2.14.1 节 async 子语中定义的那样。本例程向与第二个参数相关联的合适设备队列压入一个等待操作，用来等待与第一个参数相关联的设备队列中被压入的操作。详见 2.14 节异步行为。

3.2.10 acc_async_wait_all

概要

acc_async_wait 例程等待所有异步操作完成。

语法

C 或 C++:

```
void acc_async_wait_all( );
```

Fortran:

```
subroutine acc_async_wait_all( )
```

描述

acc_async_wait_all 例程一直等待，直到最后一个异步操作完成才返回。如果两个或两个以上线程共享同一个加速器，本线程发起的所有异步操作都完成以后，acc_async_wait_all 例程才返回；不保证其它线程发起的所有异步操作都已完成。

3.2.11 acc_async_wait_all _async

概要

acc_async_wait_all_async 例程在一个异步队列中压入一个等待操作，用于等待先前压入其它所有异步队列中的操作。

格式

C 或 C++:

```
void acc_async_wait_all_async( int );
```

Fortran:

```
subroutine acc_async_wait_all_async ( async )
integer(acc_handle_kind) async
```

描述

acc_async_wait_all_async 例程等价于带一个 async 子语的 wait 导语，且这个 async 子语包含其它所有异步活动队列的值。实参必须是一个异步参数，就像 2.14.1 节 async 子语中定义的那样。本例程将向一个合适的设备队列压入一个等待操作，为于等待剩余的所有设备队列。

3.2.12 acc_init

概要

acc_init 例程告诉运行时环境将指定设备类型的运行时环境初始化。统计性能数据时，本例程用于分离初始化时间和计算时间。

格式

C 和 C++：

```
void acc_init ( acc_device_t );
```

Fortran:

```
subroutine acc_init ( devicetype )
integer(acc_device_kind) devicetype
```

描述

acc_init 例程会隐式地调用 acc_set_device_type.

限制

- 在加速器 parallel 区域、kernels 区域内不可调用本例程。
- 如果指定的加速器类型不可用，例程的行为将由编译器决定；特别地，程序可能中

止。

- 如果本例程被以不同的设备类型参数调用一次以上，且中间没有调用 acc_shutdown，调用效果由编译器自行定义。
- 如果某些加速器区域仅为某一种设备类型而编译，那么以不同的设备类型调用本例程可能产生未定义的行为。

3.2.13 acc_shutdown

概要

acc_shutdown 告诉运行时环境关闭与指定加速器设备的连接，释放所有的运行时资源。

格式

C 或 C++：

```
void acc_shutdown ( acc_device_t );
```

Fortran:

```
subroutine acc_shutdown ( devicetype )
integer(acc_device_kind) devicetype
```

描述

acc_shutdown 例程断开程序与加速器设备的连接。

限制

- 一个加速器区域的执行过程中不可调用本例程。

3.2.14 acc_on_device

概要

acc_on_device 例程告诉程序它是否正在一个特定的设备上执行。

格式

C 或 C++：

```
int acc_on_device ( acc_device_t );
```

Fortran:

```
logical function acc_on_device ( devicetype )
integer(acc_device_kind) devicetype
```

描述

根据是否正在主机或某个加速器上运行，acc_on_device 例程可以用来执行不同的路径。如果 acc_on_device 有一个在编译时为常量的参数，那么编译时这个参数就会被计算成一个常量。参数必须为一个已定义的加速器类型。如果参数是 acc_device_host，对在主机处理器上运行的加速器计算区域或加速器例程，无论是加速器计算区域、加速器例

程的内部还是外部调用本例程，本例程在 C/C++中返回一个非零值，在 Fortran 中返回.true..如果实参是 acc_device_not_host，那么结果与实参为 acc_device_host 的结果相反。如果实参是任何设备类型，对在该类型设备上运行的加速器计算区域或加速器例程，无论是加速器计算区域、加速器例程的内部还是外部调用本例程，本例程在 C/C++中返回一个非零值，在 Fortran 中返回.true.；其它情况下，本例程在 C 或 C++中返回零，在 Fortran 中返回.false.. 实参 acc_device_default 的返回结果未定义。

3.2.15 acc_malloc

概要

acc_malloc 例程在加速器设备上分配内存。

格式

C 或 C++:

```
void* acc_malloc ( size_t );
```

描述

acc_malloc 例程可以用来在加速器设备上分配内存。本函数得到的指针可以用在 deviceptr 子语中，告诉编译器这个指针的目标已经驻留在加速器上。

3.2.16 acc_free

概要

acc_free 例程释放加速器设备上的内存空间。

格式

C 或 C++：

```
void acc_free ( void* );
```

描述

acc_free 例程将释放先前在加速器设备上分配的内存空间；输入参数必须是调用 acc_malloc 返回的指针。

3.2.17 acc_copyin

概要

在非共享内存加速器上，与指定的主机内存相对应，acc_copyin 例程在加速器设备上分配内存，并将对应数据复制到新分配的设备内存。

格式

C 或 C++:

```
void* acc_copyin ( h_void*, size_t );
```

Fortran:

```
subroutine acc_copyin ( a )
  type, dimension(:[,:]…) :: a
subroutine acc_copyin ( a, len )
  type :: a
  integer :: len
```

描述

acc_copyin 例程等价于带有一个 copyin 子语的 enter data 导语。在 C 中，实参是一个指向数据的指针和一个按字节计数的长度；像使用了 acc_malloc 一样，本函数返回一个已分配空间的指针。用本函数赋值的指针可以用在 deviceptr 子语中，以告诉编译器该指针的目标已经存在于加速器上。在 Fortran 中，本例程有两种形式。第一种，实参是一个内置类型数组的连续区块。第二种，第一个实参是一个变量或一个数组元素，且第二个实参是它的按字节计数的长度。在设备上分配内存，数据被从主机内存复制到新分配的设备内存。调用本例程将开始指定数据的数据生存期。使用 present 数据子语可以访问这个数据。如果设备上已经存在数据，调用本例程是一个运行时错误。

3.2.18 acc_present_or_copyin

概要

对非共享内存设备，acc_present_or_copyin 例程检测数据是否已经存在于设备中；如果没有，它将在加速器设备上为指定的主机内存分配对应的内存，并将数据复制到这块设备内存。

格式

C 或 C++:

```
void* acc_present_or_copyin ( h_void*, size_t );
void* acc_pcopyin ( h_void*, size_t );
```

Fortran:

```
subroutine acc_present_or_copyin ( a )
  type, dimension(:[,:]…) :: a
subroutine acc_present_or_copyin ( a, len )
  type :: a
  integer :: len
subroutine acc_pcopyin ( a )
  type, dimension(:[,:]…) :: a
subroutine acc_pcopyin ( a, len )
  type :: a
  integer :: len
```

描述

acc_present_or_copyin 例程等价于带有一个 present_or_copyin 子语的 enter data 导语。参数与 acc_copyin 例程相同。如果数据已经存在于设备上，或者设备与主调者共享内存，没有动作。在一个没有数据的非共享内存设备上，本例程在设备上分配内存，并将数据复制到新分配的设备内存中。在后一种情形中，对本例程的一次调用会开

启指定数据的一个数据生存期。可以使用 present 数据子语来访问这个数据。

3.2.19 acc_create

概要

在一个非共享内存加速器上，acc_create 例程为指定的主机内存分配对应的加速器设备内存。

格式

C 或 C++:

```
void* acc_create ( h_void*, size_t );
```

Fortran:

```
subroutine acc_create ( a )
  type, dimension(:[,:]…) :: a
subroutine acc_create ( a, len )
  type :: a
  integer :: len
```

描述

acc_create 例程等价于带有一个 create 子语的 enter data 导语。在 C 中，参数是一个指向数据的指针和一个按字节计数的长度；本函数返回一个已分配空间的指针，就像使用了 acc_malloc 一样。被本函数赋值的指针可以用在 deviceptr 子语中，以告诉编译器该指针的目标已经存在于加速器上。在 Fortran 中，本例程有两种形式。第一种，实参是一个内置类型数组的连续区块。第二种，第一个实参是一个变量或一个数组元素，且第二个实参是它的按字节计数的长度。在一个非共享内存设备上，内存在设备上分配。调用本例程会开启指定数据的一个数据生存期。可以使用 present 数据子语来访问这个数据。如果设备上已经存在数据，调用本例程是一个运行时错误。

3.2.20 acc_present_or_create

概要

在一个非共享内存的设备上，acc_present_or_create 例程检测数据是否已经存在于设备之上；如果不存在，它为指定的主机内存分配对应的加速器设备内存。

格式

C 或 C++:

```
void* acc_present_or_create ( h_void*, size_t );
void* acc_pcreate ( h_void*, size_t );
```

Fortran:

```
subroutine acc_present_or_create ( a )
  type, dimension(:[,:]…) :: a
subroutine acc_present_or_create ( a, len )
  type :: a
```

```
  integer :: len
subroutine acc_pcreate ( a )
  type, dimension(:[,:]…) :: a
subroutine acc_pcreate ( a, len )
  type :: a
  integer :: len
```

描述

acc_present_or_create 例程等价于带有一个 present_or_create 子语的 enter data 导语。参数与 acc_create 例程相同。如果数据已经存在于设备上，或者设备与主调者共享内存，没有动作。在一个没有数据的非共享内存设备上，本例程在设备上分配内存。在后一种情形中，对本例程的一次调用会开启指定数据的一个数据生存期。可以使用 present 数据子语来访问这个数据。

3.2.21 acc_copyout

概要

在非共享内存加速器上，acc_copyout 例程将数据从设备内存复制到对应的本地内存，然后在加速器设备上撤销内存。

格式

C 或 C++:

```
void acc_copyout ( h_void*, size_t );
```

Fortran:

```
subroutine acc_copyout ( a )
  type, dimension(:[,:]…) :: a
subroutine acc_copyout ( a, len )
  type :: a
  integer :: len
```

描述

acc_copyout 例程等价于带有一个 copyout 子语的 exit data 导语。在 C 中，实参是一个指向数据的指针和一个按字节计数的长度。在 Fortran 中，本例程有两种形式。第一种，实参是一个内置类型数组的连续区块。第二种，第一个实参是一个变量或一个数组元素，且第二个实参是它的按字节计数的长度。调用本例程会将数据从加速器设备内存复制到本地内存，然后撤销加速器内存。调用本例程将终结指定数据的数据生存期。如果指定的数据不在设备上或者在一个数据区域之内，调用本例程是一个运行时错误。

3.2.22 acc_delete

概要

在一个非共享内存加速器上，acc_delete 例程从加速器设备上撤销指定本地内存对应用的设备内存。

格式

C 或 C++:

```
void acc_delete ( h_void*, size_t );
```

Fortran:

```
subroutine acc_delete ( a )
  type, dimension(:[,:]…) :: a
subroutine acc_delete ( a, len )
  type :: a
  integer :: len
```

描述

acc_delete 等价于带有一个 delete 子语的 exit data 导语。参数与 acc_copyout 相同。调用本例程将撤销与指定本地内存相对应的加速器内存。调用本例程会终结指定数据的一个数据生存期。如果指定的数据不在设备上或者在一个数据区域之内，调用本例程是一个运行时错误。

3.2.23 acc_update_device

概要

在一个非共享内存加速器上，acc_update_device 用对应的本地内存来更新数据的设备复本。

格式

C 或 C++:

```
void acc_update_device ( h_void*, size_t );
```

Fortran:

```
subroutine acc_update_device ( a )
  type, dimension(:[,:]…) :: a
subroutine acc_update_device ( a, len )
  type :: a
  integer :: len
```

描述

acc_update_device 等价于一个带有 device 子语的 update 导语。在 C 中，参数是一个指向数据的指针和一个按字节计数的长度。在 Fortran 中，本例程有两种形式。第一种 ,参数是一个内置类型数组的连续区块。第二种 ,第一个参数是一个变量或一个数组元素，且第二个参数是它的按字节计数的长度。一个非共享内存的设备上，本地内存中的数据被复制到对应的设备内存。如果设备上没有数据，调用本例程是一个运行时错误。

3.2.24 acc_update_self

概要

在一个非共享内存加速器上，acc_update_self 例程用设备内存中的数据来更新对

应的主机内存数据[1]。

格式

C 或 C++:

```
void acc_update_self ( h_void*, size_t );
```

Fortran:

```
subroutine acc_update_self ( a )
  type, dimension(:[,:]…) :: a
subroutine acc_update_self ( a, len )
  type :: a
  integer :: len
```

描述

acc_update_self 等价于带有一个 self 子语的 update 导语。在 C 中，参数是一个数据指针和一个按字节计数的长度。在 Fortran 中，本例程有两种形式。第一种，参数是一个内置类型数组的连续区块。第二种，第一个参数是一个变量或一个数组元素，且第二个参数是它的按字节计数的长度。一个非共享内存的设备上，设备内存中的数据被复制到对应的本地内存[2]。如果设备上没有数据，调用本例程是一个运行时错误。

3.2.25 acc_map_data

概要

acc_map_data 例程将先前分配的设备数据映射到指定的主机数据。

格式

C 或 C++:

```
void acc_map_data ( h_void*, d_void*, size_t );
```

描述

acc_map_data 例程与带有一个 create 子语的 enter data 相似，但它不是通过分配新的设备空间以开启一个数据生存期，而是以指定的设备地址参数开启数据生存期。第一个参数是主机地址，后面跟着对应的设备地址和以字节计数的数据长度。调用这个函数之后，主机数据出现在一个数据子语中时，将会替代地使用指定的设备内存。如果数据已经存在于设备上，为主机调用 acc _map_data 将是一个错误。对一个已经映射到主机数据的设备地址调用 acc_map_data，调用后果未定义。设备地址可以是 acc_malloc 的调用结果，也可以来自其它特定设备接口例程。

[1] 原文对该例程的复制方向描述有误，已更正。--译者注。

[2] 原文是：the data in the local memory is copied to the corresponding device memory（本地内存中的数据被复制到对应的设备内存），有误。--译者注。

3.2.26 acc_unmap_data

概要

acc_unmap_data 例程解除从指定主机数据到设备数据的映射关联。

格式

C 或 C++:

```
void acc_unmap_data ( h_void* );
```

描述

acc_unmap_data 例程与带有一个 delete 子语的 exit data 导语相像，除了不撤销设备内存。参数是主机数据的一个指针。调用本例程会终结指定主机数据的数据生存期。设备内不会被撤销。除非主机地址已经通过 acc_map_data 映射到设备地址，否则一个带主机地址参数调用 acc_unmap_data 的形为未定义。

3.2.27 acc_deviceptr

概要

acc_deviceptr 例程返回与指定主机地址相关联的设备指针。

格式

C 或 C++:

```
d_void* acc_deviceptr ( h_void* );
```

描述

acc_deviceptr 例程返回与一个主机地址相关联的设备指针。参数是一个主机变量或数组的地址，这个变量或数组在当前设备上有一个活动生存期。如果数据不存在于设备上，本例程返回值为 NULL.

3.2.28 acc_hostptr

概要

acc_hostptr 例程返回与指定设备地址相关联的主机指针。

格式

C 或 C++:

```
h_void* acc_hostptr ( d_void* );
```

描述

acc_hostptr 例程返与一个主机地址相关联的设备指针。参数是一个设备变量或数组的地址，例如 acc_deviceptr、acc_create 或 acc_copyin 的返回地址。如果设备地址是 NULL，或设备地址不对应任何主机地址，那么本例程返回值为 NULL.

3.2.29 acc_is_present

概要

acc_is_present 例程检测一个主机变量或数组是否存在于设备上。

格式

C 或 C++:

```
int acc_is_present ( h_void*, size_t );
```

Fortran:

```
logical function acc_is_present ( a )
  type, dimension(:[,:]…) :: a
logical function acc_is_present ( a, len )
  type :: a
  integer :: len
```

描述

acc_is_present 例程检测指定的主机数据是否存在于设备上。在 C 中，参数是一个数据指针和一个以字节计数的数据长度；如果指定的数据在设备上完全存在，那么本函数返非零值，否则返回值为零。在 Fortran 中，有两种形式。第一种，参数是一个内置类型数组的连续区块。第二种，第一个参数是一个变量或一个数组元素，且第二个参数是它的按字节计数的数据长度。如果指定的数据完全存在，本函数返回.true.，否则返回.false..如果字节长度为 0，且给定的地址存在于所有的设备上，那么本函数在 C 中返回一个非零值，在 Fortran 中返回.true..

3.2.30 acc_memcpy_to_device

概要

acc_memcpy_to_device 例程将数据从本地内存复制到设备内存。

格式

C 或 C++:

```
void acc_memcpy_to_device ( d_void* dest, h_void* src,
                            size_t bytes );
```

描述

acc_memcpy_to_device 例程将字节数据从 src 中的本地地址复制到 dest 中的设备地址。终点地址必须是一个设备地址，例如可以是 acc_malloc 或 acc_deviceptr 的返回值。

3.2.31 acc_memcpy_from_device

概要

acc_memcpy_from_device 例程将数据从设备内存复制到本地内存。

格式

C 或 C++:

```
void acc_memcpy_from_device ( h_void* dest, d_void* src,
                              size_t bytes );
```

描述

`acc_memcpy_from_device` 例程将字节数据从 `src` 中的设备地址复制到 `dest` 中的本地地址。源地址必须是一个设备地址，例如可以是 `acc_malloc` 或 `acc_deviceptr` 的返回值。

第4章　环境变量

本章讲述用以改变加速器区域行为的环境变量。环境变量的名字必须大写。赋给环境变量的值是区分大小写的，值的前后都可以有空白字符。在程序开始之后，如果环境变量的值发生了改变，即使是程序自己修改的这些值，修改后形为仍有编译器定义。

4.1 ACC_DEVICE_TYPE

如果程序被编译成可以在一种以上类型的设备上执行的版本，环境变量 ACC_DEVICE_TYPE 控制执行加速器 parallel 区域或 kernels 区域时使用的默认设备类型。这个环境变量允许使用的值由编译器定义。请在发行说明里查看这个环境变量当前支持哪些值。

例子：

```
setenv ACC_DEVICE_TYPE NVIDIA
export ACC_DEVICE_TYPE=NVIDIA
```

4.2 ACC_DEVICE_NUM

环境变量 ACC_DEVICE_NUM 控制着执行加速器区域时使用的默认设备编号。这个环境变量的值必须是一个非负整数，取值范围下限是零，上限是主机上所用类型设备的数量。如果值为零，使用编译器定义的默认设备。如果值超过主机上的设备数量，行为由编译器定义。

例子：

```
setenv ACC_DEVICE_NUM 1
export ACC_DEVICE_NUM=1
```

第5章　词汇表

清晰一致的术语对讲述任何编程模型都很重要。你必须理解我们在这里定义的名词，以便有效利用本文档及文档中的编程模型。

加速器(accelerator) – 一个挂载在 CPU 上的特殊功能协处理器，CPU 可以将数据和计算核心卸载到它上面进行计算密集型运算。

加速器例程(accelerator routine) – 使用 routine 导语为加速器编译的一个 C/C++函数或 Fortran 子程序。

加速器线程(accelerator thread) – 一个在加速器上执行的线程；一个 gang 的一个 worker 的一个向量通道。

异步参数(async-argument) – 一个非负标量整数表达式(C 或 C++中 `int`，Fortran 中是 `integer`)，或者是特殊异步值 `acc_async_noval` 或 `acc_async_sync` 中的一个。

屏障(barrier) – 一种同步，在任何并行执行单元或线程被允许通过此屏障之前，所有的并行执行单元或线程必须到达该屏障；就像赛马场起点上的屏障。

计算密度(compute intensity) – 对一个给定的循环、区域或程序单元，计算密度就是在数据上完成的算术操作次数除以在两个内存层级间的内存存取次数得到的比值。

构件(construct) - 一个导语及其关联的语句、循环或结构块（如果有的话）。

计算区域(compute region) – 一个 parallel 区域或一个 kernels 区域。

CUDA – 来自英伟达的 CUDA 环境是一个类 C 编程环境，它用来显式地控制英伟达 GPU 和在英伟达 GPU 上编程。

数据生存期(data lifetime) – 设备上一个数据对象的生存期，开始位置可以是一个数据区域的入口处、一个 `enter data` 导语、`acc_copyin` 或 `acc_create` 等数据例程的调用点，结束位置可以是一个数据区域的出口处、一个 `exit data` 导语、`acc_delete` 或 `acc_copyout` 或 `acc_shutdown` 等数据例程的调用点、或者程序结尾处。

数据区域(data region) – 用加速器 `data` 构件定义的一个区域，或者是包含加速器导语的例程、程序的一个隐式数据区域。`data` 构件通常会开辟设备内存，在入口处将数据从主机复制到设备内存，在出口处将数据从设备复制回主机内存并撤销设备内存。数据区域可以包含其它数据区域和计算区域。

设备(device) – 所有类型加速器的统一称谓。

设备内存(device memory) – 加速器的附属内存，在物理上与逻辑上都与主机内存相分离。

导语(directive) – 在 C 或 C++中，一个`#program;` 在 Fortran 中，一个特殊格式的注释语句。编译器能从导语中得到控制程序形为的额外信息。

DMA – 直访（直接内存访问，Direct Memory Access），在物理上相互分离的内存间移动数据的一种方法；它通常由一个与主机 CPU 相分离的直访引擎来完成，直访引引擎可以像访问一个 IO 设备一样访问主机物理内存或其它物理内存。

GPU – 图形处理单元(Graphics Processing Unit)；加速器设备的一种。

GPGPU – 图形处理单元上的通用计算(General Purpose)。

主机(host) - 本文档中，主机是具有一个附属加速器设备的 CPU。主机 CPU 将程序区域和数据加载到设备上，并在设备上运行。

主机线程(host thread) – 一个在主机上执行的线程。

隐式数据区域(implicit data region) – 为一个 Fortran 子程或一个 C 函数隐式定义的数据区域。调用一个子程序或函数时进入这个隐式数据区域，从子程序或函数里返回时退出这个数据区域。

内核(kernel) - 被加速器并行执行的嵌套循环。通常循环被划分为一个并行区块，循环的主体也就成为内核的主体。

内核区域(kernel region) - 一个加速器 kernels 构件定义的区域。内核区域就是一个为加速器编译的结构块。编译器将内核区域内的代码分割为一系列内核；通常，每个循环成为一个独立的内核。一个内核区域可能会要求开辟设备内存空间，在区域入口处将数据从主机复制到设备，离开区域时将数据从设备内存复制到主机内存。

本地内存(local memory) – 与本地线程相关联的内存。

本地线程(Local thread) – 主机线程，或者是执行一个 OpenACC 导语或构件的加速器线程。

循环路程步数(loop trip count) - 某个特定的循环被执行的次数。

MIMD – 多指令多数据(Multiple Instruction, Multiple Data)，一种并行执行方法。不同的运算单元或线程相互间异步地执行不同的指令流。

OpenCL – 开放计算语言(Open Compute Language)的缩写，一个发展中的、可移植的类 C 语言编程标准，它能在 GPU 和其它加速器上进行低层通用编程。

并行区域(parallel region) - 加速器 parallel 构件定义的一个区域。并行区域就是专为加速器编译的结构化块。一个并行区域通常包含一个或多个工作分担(work-sharing)循环。一个并行区域可能会要求开辟设备内存空间，在区域入口处将数据从主机复制到设备，离开区域时将数据从设备内存复制到主机内存。

过程(procedure) – C 或 C++中，程序中的一个函数；Fortran 中，一个子例程或函数。

区域（region） - 一个构件的执行实例中遇到的所有代码。一个区域包括被调用例程的所有代码，也可以认为是一个构件的动态上下文。它可能是 parallel 区域、kernels 区域、data 区域或隐式 data 区域。

SIMD – 单指令多数据（Single-Instruction, Multiple-Data），一种并行执行方法，将单个指令同时用到多个数据单元上。

SIMD 操作（SIMD operation） - 一个用 SIMD 指令实现的向量操作。

结构块（structured block） - C 或 C++中，一个可执行语句，也可以是复合语句，该语句顶部有唯一入口，底部有唯一出口。Fortran 中，顶部有唯一入口且底部有唯一出口的一块可执行语句。

线程(thread) – 一个主机处理器上，一个程序计数器和栈位置定义一个线程；几个线程可以组成一个进程，并共享主机内存。在加速器上，线程是设备上的一个 `gang` 的一个 `worker` 的任意一个向量通道。

向量操作（vector operation） - 一致地作用于一个数组全部元素的单个操作或一系列操作。

可见设备副本（visible device copy） - 变量、数组、子数组在设备内存中的一个副本，直到编译时，这些副本对程序可见。

附录 A 对支持特定设备编译器的建议

为提高跨编译器移植性 ,本节为支持特定设备和特定平台的编译器提出一些标准名称和扩展名称方面的建议。尽管这个附录不是 OpenACC 规范的一部分，但强烈建议提供本附录中功能的编译器使用这里建议的名称。第一小节描述目标设备，例如 NVIDIA GPU 和 Intel Xeon Phi 协处理器。第二小节描述为目录平台额外设计的接口例程 ,例如 CUDA 和 OpenCL. 第三小节列出了几个编译器推荐选项。

A.1 目标设备

A.1.1 NVIDIA GPU 目标

本节为以 NVIDIA GPU 设备为目标的编译器提一些建议。

A.1.1.1 加速器设备类型

对 `acc_device_t` 类型和一个 OpenACC 运行时接口例程的返回值，这些编译器需要使用名字 `acc_device_nvidia`.

A.1.1.2 ACC_DEVICE_TYPE

对环境变量 `ACC_DEVICE_TYPE`，编译器需要使用不区分大小的名字 `NVIDIA`.

A.1.1.3 device_type 子语的参数

编译器应当使用名字 `nvidia` 或 `NVIDIA` 作为 `device_type` 子语的参数。

A.1.2 AMD GPU 目标

本节为以 AMD GPU 设备为目标的编译器提一些建议。

A.1.2.1 加速器设备类型

对 `acc_device_t` 类型和一个 OpenACC 运行时接口例程的返回值，这些编译器需要使用名字 `acc_device_radeon`.

A.1.2.2 ACC_DEVICE_TYPE

对环境变量 `ACC_DEVICE_TYPE`，这些编译器需要使用不区分大小的名字 `RADEON` .

A.1.2.3 device_type 子语的参数

编译器应当使用名字 `radeon` 或 `RADEON` 作为 `device_type` 子语的参数。

A.1.3 Intel Xeon Phi 协处理器目标

本节为以 Intel Xeon Phi 协处理器为目标的编译器提一些建议。

A.1.3.1 加速器设备类型

对 `acc_device_t` 类型和一个 OpenACC 运行时接口例程的返回值，这些编译器需要使用名字 `acc_device_xeonphi`.

A.1.3.2 ACC_DEVICE_TYPE

对环境变量 ACC_DEVICE_TYPE，这些编译器需要使用不区分大小的名字 XEONPHI.

A.1.3.3 device_type 子语的参数

编译器应当使用名字 xeonphi 或 XEONPHI 作为 device_type 子语的参数。

A.2 目标平台的接口例程

这些运行时例程允许 OpenACC 运行时例程与底下目标平台间的接口。一个编译器不必实现所有这些例程，但是如果提供了这个功能，它应该使用这些名字。

A.2.1 NVIDIA CUDA 平台

本节为那些支持 NVIDIA CUDA 运行时或驱动接口的编译器提供运行时接口例程。

A.2.1.1 acc_get_current_cuda_device

概要

acc_get_current_cuda_device 例程返回当前设备的 NVIDIA CUDA 设备句柄。

格式

C 或 C++：

```
void* acc_get_current_cuda_device ();
```

A.2.1.2 acc_get_current_cuda_context

概要

acc_get_current_cuda_context 例程返回当前设备正在使用中的 NVIDIA CUDA 上下文句柄。

格式

C 或 C++:

```
void* acc_get_current_cuda_context ();
```

A.2.1.3 acc_get_cuda_stream

概要

对指定异步值的当前设备，acc_get_cuda_stream 例程返回它正在使用的 NVIDIA CUDA 流句柄。

格式

C 或 C++:

```
void* acc_get_cuda_stream ( int async );
```

A.2.1.4 acc_set_cuda_stream

概要

对指定异步值的当前设备，acc_set_cuda_stream 例程设定它的 NVIDIA CUDA 流句柄。

格式

C 或 C++：

```
int acc_set_cuda_stream ( int async, void* stream );
```

A.2.2 OpenCL 目标平台

本节为那些在任何设备上都支持 OpenCL 编程接口的编译器提供运行时接口例程。

A.2.2.1 acc_get_current_opencl_device

概要

acc_get_current_opencl_device 例程返回当前设备的 OpenCL 设备句柄。

格式

C 或 C++:

```
void* acc_get_current_opencl_device ();
```

A.2.2.2 acc_get_current_opencl_context

概要

acc_get_current_opencl_context 例程返回当前设备正在使用中的 OpenCL 上下文句柄。

格式

C 或 C++:

```
void* acc_get_current_opencl_context ();
```

A.2.2.3 acc_get_opencl_queue

概要

对指定异步值的当前设备，acc_get_opencl_queue 返回正在使用中的 OpenCL 命令队列句柄。

格式

C 或 C++:

```
cl_command_queue acc_get_opencl_queue ( int async );
```

A.2.2.4 acc_set_opencl_queue

概要

对指定异步值的当前设备，acc_set_opencl_queue 例程设定 OpenCL 命令队列句柄[1]。

格式

C 或 C++:

```
void acc_set_opencl_queue ( int async, cl_command_queue
                            cmdqueue);
```

A.2.3 Intel 协处理器卸载架构接口（Coprocessor Offload Infrastructure，COI）

这些运行时例程允许访问 OpenACC 运行时例程与底层 Intel 协处理器卸载架构间的接口。

A.2.3.1 acc_get_current_coi_device

概要

acc_get_current_coi_device 例程返回当前设备的 COI 设备句柄。

格式

C 或 C++:

```
void* acc_get_current_coi_device ();
```

A.2.3.2 acc_get_current_coi_context

概要

acc_get_current_coi_context 例程返回当前设备正在使用中的 COI 上下文句柄。

格式

C 或 C++:

```
void* acc_get_current_coi_context ();
```

A.2.3.3 acc_get_coi_pipeline

概要

对指定异步值的当前设备，acc_get_coi_pipeline 例程返回正在使用中的 COI 管线句柄。

格式

C 或 C++:

```
void* acc_get_coi_pipeline ( int async );
```

[1] 原文是：… returns the OpenCL command queue handle in use for the current device for the specified async value(返回正在使用中的 OpenCL 命令队列句柄)，有误，已纠正。--译者注。

A.2.3.4 acc_set_coi_pipeline

概要

对指定异步值的当前设备，`acc_set_coi_pipeline` 例程设定 COI 管线的句柄[1]。

格式

C 或 C++:

```
void acc_set_coi_pipeline ( int async, void* pipeline );
```

A.3 推荐选项

下面是建议编译器使用的选项；例如，它们可以用作编译器的命令行选项，也可在一个 IDE 设置之中。

A.3.1 present 子语中的 C 指针

本版 OpenACC 澄清了这个构件：

```
void test(int n ){
    float* p;
    …
    #pragma acc data present(p)
    { // 代码写在这里 …
```

这个例子检测指针 `p` 自身是否在设备上。在这个版本之前，编译器通常靠检测指针目标 `p[0]` 是否在设备上来实现这个功能，而很多程序也假定编译器是这样工作的。在所有程序都进行修改以遵守这个版本规范之前，对 C 指针，提供一个选项像 `present(p[0])` 一样实现 `present(p)` 的功能，这将对用户很有帮助。

A.3.2 自动扫描

如果一个编译器实现了自动扫描功能，自动判定计算区域或循环的私有变量，或者自动识别计算构件或循环上的归约操作，一个控制打印信息的选项将对用户很有帮助，它能显示哪些变量受到了自动扫描的影响。一个关闭自动扫描分析功能的选项将有助于提高程序的跨编译器移植性。

[1] 原文是：… routine returns the COI pipeline handle in use for the current device for the specified async value(..返回正在使用中的 COI 管线句柄…)，有误，已纠正。--译者注。

中译版更新记录

2.0 版，2013-8-26

根据 OpenACC 2.0 标准规范英文版翻译，与 OpenACC 1.0 标准重合的内容，直接采用 1.0 中译版的内容。