(19)中华人民共和国国家知识产权局

(12)发明专利

(10)授权公告号 CN 102314342 B
(45)授权公告日 2016.08.17

(21)申请号 201110177689.1

(22)申请日 2011.06.17

(30)优先权数据
12/819,108 2010.06.18 US

(73)专利权人 微软技术许可有限责任公司
地址 美国华盛顿州

(72)发明人 L·张 W·朱 Y·莱瓦诺尼 P·F·林塞斯 C·D·卡拉汉二世

(74)专利代理机构 上海专利商标事务所有限公司 31100
代理人 顾嘉运

(51)Int.Cl.
G06F 9/44(2006.01)

(56)对比文件
CN 1499362 A,2004.05.26,
US 2009307655 A1,2009.10.10,
Gautier et al..Re-scheduling invocations of services for RPC grids.《COMPUTER LANGUAGES,SYSTERMS & STRUCTURES》.2006,第33卷第168至178页.

审查员 汪见晗

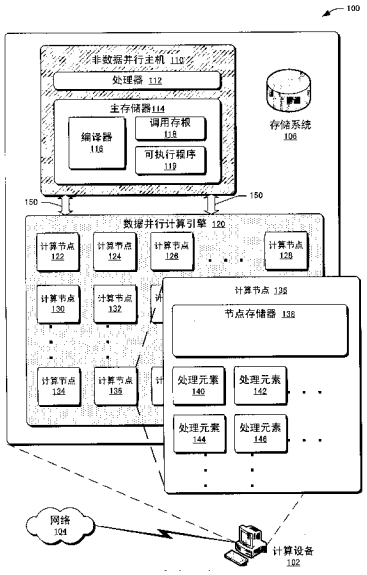
权利要求书4页 说明书53页 附图5页

(54)发明名称

用于数据并行编程模型的编译器生成的调用存根

(57)摘要

在此描述的是用于生成用于数据并行编程模型的调用存根以使得用静态编译的高级编程语言编写的数据并行程序可以比传统方法更加声明性、可重用和可移植的技术。借助于所描述的技术中的某一些,由编译器(116)生成调用存根(118),且那些存根将数据并行计算的逻辑排列桥接到用于数据并行计算的目标数据并行硬件的实际物理排列。



CN 102314342 B

1.一种促进数据并行(DP)可执行程序的产生的方法,所述方法包括:

获得(202)对DP函数的调用的表示,其中所述表示包括与对所述DP函数的所述调用相关联的自变量的指示符;

至少部分地基于所述DP函数表示及其关联的自变量生成(400)调用存根,所述调用存根包括通过基于物理线程索引和逻辑计算域计算每一DP活动的逻辑索引来将所述DP函数的DP计算的逻辑排列桥接到要在一个或多个计算设备的DP硬件上执行的DP计算的物理排列的计算机可执行指令。

2.如权利要求1所述的方法,其特征在于,所述生成包括确定所述DP硬件中将对所述DP计算的逻辑排列进行所述DP计算的物理位置。

3.如权利要求1所述的方法,其特征在于,所述生成包括用目标相关资源设置用于所述DP函数的字段,其中所述字段是要由所述DP函数的DP计算操作的数据集。

4.如权利要求1所述的方法,其特征在于,所述生成包括用目标相关资源设置用于所述DP函数的字段,其中所述字段包括要由所述DP函数的DP计算操作的数据集,并且其中所述设置包括检查与所述DP函数相关联的自变量以及基于所述检查配置所述数据集。

5.如权利要求1所述的方法,其特征在于,所述生成包括用目标相关资源设置用于所述DP函数的字段,其中所述字段包括由所述DP函数的DP计算操作的数据集,并且其中所述设置包括声明应将线程组中的多少个线程部署在多个维度中。

6.如权利要求1所述的方法,其特征在于,所述生成包括用目标相关资源设置用于所述DP函数的字段,其中所述字段包括由所述DP函数的DP计算操作的数据集,并且其中所述设置包括将所述DP硬件的线程位置映射到所述DP计算的逻辑排列中的逻辑DP活动位置。

7.如权利要求1所述的方法,其特征在于,所述生成包括用目标相关资源设置用于所述DP函数的字段,其中所述字段包括由所述DP函数的DP计算操作的数据集,并且其中所述设置包括将所述DP硬件的线程位置映射到所述DP计算的所述逻辑排列中的逻辑DP活动位置,所述映射包括变换索引展平和索引提高。

8.如权利要求1所述的方法,其特征在于,所述生成包括基于内核期望的内容来准备所述内核的参数,所述内核是所述DP函数的单元DP计算。

9.如权利要求1所述的方法,其特征在于,所述生成包括基于内核期望的内容来准备所述内核的参数,所述内核是所述DP函数的单元DP计算,其中所述准备包括:

确定所关联的自变量是否包括一个或多个特殊索引;

响应于所述确定,基于一个或多个特殊索引的类型创建索引实例。

10.如权利要求1所述的方法,其特征在于,所述生成包括基于内核期望的内容来准备所述内核的参数,所述内核是所述DP函数的单元DP计算,其中所述准备包括:

确定所关联的自变量是否匹配所述内核的实际参数;

响应于所述确定,广播一值以供所述内核使用。

11.如权利要求1所述的方法,其特征在于,所述生成包括基于内核期望的内容来准备所述内核的参数,所述内核是所述DP函数的单元DP计算,其中所述准备包括:

确定所关联的自变量是否匹配实际参数;

响应于所述确定,投影字段,其中所述字段是由所述DP函数的DP计算操作的数据集。

12.如权利要求1所述的方法,其特征在于,还包括将所述调用存根输出到存储器中。

13.如权利要求1所述的方法，其特征在于，所述DP硬件能够执行DP计算，并且所述DP计算的逻辑排列至少部分地由具有其所关联的自变量的所述对DP函数的调用的表示来定义。

14.一种方法，包括：

获得(202)对DP函数的调用的表示，其中所述表示包括与对所述DP函数的调用相关联的自变量的指示符；

至少部分地基于所述DP函数表示及其关联的自变量生成(400)调用存根，所述调用存根包括将所述DP函数的DP计算的逻辑排列桥接到要在一个或多个计算设备的DP硬件上执行的DP计算的物理排列的计算机可执行指令，所述DP硬件能够执行DP计算且所述DP计算的逻辑排列至少部分地由具有其所关联的自变量的所述对DP函数的调用的表示来定义，

所述生成包括：

确定所述DP硬件中将对所述DP计算的逻辑排列进行所述DP计算的物理位置；

选择适当的线程部署策略；

用目标相关资源设置用于所述DP函数的字段，其中所述字段包括由所述DP函数的DP计算操作的数据集，所述设置包括：

检查与所述DP函数相关联的自变量并基于所述检查配置所述数据集；

声明应将线程组中的多少个线程部署在多个维度中；

基于内核期望的内容来准备所述内核的参数，所述内核是所述DP函数的单元DP计算，所述准备包括：

确定所关联的自变量是否包括一个或多个特殊索引；

响应于关于特殊索引的所述确定，基于所述一个或多个特殊索引类型创建索引实例；

确定所关联的自变量是否匹配实际参数；

响应于关于自变量匹配参数的所述确定，广播一值以供所述内核使用或投影字段，其中所述字段是由所述DP函数的DP计算操作的数据集。

15.一种方法，包括：

获得(202)对DP函数的调用的表示，其中所述表示包括与对所述DP函数的所述调用相关联的自变量的指示符；

至少部分地基于所述DP函数表示及其关联的自变量生成(400)调用存根，所述调用存根包括将所述DP函数的DP计算的逻辑排列桥接到要在一个或多个计算设备的DP硬件上执行的DP计算的物理排列的计算机可执行指令，所述DP硬件能够执行DP计算且所述DP计算的逻辑排列由具有其所关联的自变量的所述对DP函数的调用的表示来定义；

所述生成包括：

用目标相关资源设置用于所述DP函数的字段，其中所述字段包括由所述DP函数的DP计算操作的数据集；

基于内核期望的内容来准备所述内核的参数，所述内核是所述DP函数的单元DP计算；

将所述调用存根输出到存储器中。

16.如权利要求15所述的方法，其特征在于，所述设置包括：

检查与所述DP函数相关联的自变量并基于所述检查配置所述数据集；

声明应将线程组中的多少个线程部署在多个维度中；以及

将所述DP硬件的线程位置映射到所述DP计算的逻辑排列中的逻辑DP活动位置。

17.如权利要求15所述的方法，其特征在于，所述设置包括将所述DP硬件的线程位置映射到所述DP计算的逻辑排列中的逻辑DP活动位置，所述映射包括变换索引展平和索引提高。

18.如权利要求15所述的方法，其特征在于，所述准备包括：

确定所关联的自变量是否包括一个或多个令牌；

响应于关于令牌的所述确定，基于所述一个或多个令牌创建索引实例。

19.如权利要求15所述的方法，其特征在于，所述准备包括：

确定所关联的自变量是否匹配实际参数；

响应于关于自变量匹配参数的所述确定，广播一值以供所述内核使用或投影字段，其中所述字段包括由所述DP函数的DP计算操作的数据集。

20.如权利要求15所述的方法，其特征在于，所述准备包括：

确定所关联的自变量是否包括一个或多个令牌；

响应于关于令牌的所述确定，基于所述一个或多个令牌创建索引实例；

确定所关联的自变量是否匹配实际参数；

响应于关于匹配参数的自变量所述确定，广播一值以供所述内核使用或投影字段，其中所述字段包括由所述DP函数的DP计算操作的数据集。

21.一种系统，包括：

用于获得(202)对DP函数的调用的表示的装置，其中所述表示包括与对所述DP函数的调用相关联的自变量的指示符；

用于至少部分地基于所述DP函数表示及其关联的自变量生成(400)调用存根的装置，所述调用存根包括将所述DP函数的DP计算的逻辑排列桥接到要在一个或多个计算设备的DP硬件上执行的DP计算的物理排列的计算机可执行指令，所述DP硬件能够执行DP计算且所述DP计算的逻辑排列至少部分地由具有其所关联的自变量的所述对DP函数的调用的表示来定义，

所述生成包括：

确定所述DP硬件中将对所述DP计算的逻辑排列进行所述DP计算的物理位置；

选择适当的线程部署策略；

用目标相关资源设置用于所述DP函数的字段，其中所述字段包括由所述DP函数的DP计算操作的数据集，所述设置包括：

检查与所述DP函数相关联的自变量并基于所述检查配置所述数据集；

声明应将线程组中的多少个线程部署在多个维度中；

基于内核期望的内容来准备所述内核的参数，所述内核是所述DP函数的单元DP计算，所述准备包括：

确定所关联的自变量是否包括一个或多个特殊索引；

响应于关于特殊索引的所述确定，基于所述一个或多个特殊索引类型创建索引实例；

确定所关联的自变量是否匹配实际参数；

响应于关于自变量匹配参数的所述确定，广播一值以供所述内核使用或投影字段，其中所述字段是由所述DP函数的DP计算操作的数据集。

22.一种系统，包括：

用于获得(202)对DP函数的调用的表示的装置，其中所述表示包括与对所述DP函数的所述调用相关联的自变量的指示符；

用于至少部分地基于所述DP函数表示及其关联的自变量生成(400)调用存根的装置，所述调用存根包括将所述DP函数的DP计算的逻辑排列桥接到要在一个或多个计算设备的DP硬件上执行的DP计算的物理排列的计算机可执行指令，所述DP硬件能够执行DP计算且所述DP计算的逻辑排列由具有其所关联的自变量的所述对DP函数的调用的表示来定义；

所述生成包括：

用目标相关资源设置用于所述DP函数的字段，其中所述字段包括由所述DP函数的DP计算操作的数据集；

基于内核期望的内容来准备所述内核的参数，所述内核是所述DP函数的单元DP计算；

用于将所述调用存根输出到存储器中的装置。

用于数据并行编程模型的编译器生成的调用存根

技术领域

[0001] 本发明一般涉及数据并行编程，尤其涉及用于数据并行编程模型的编译器生成的调用存根。

背景技术

[0002] 在数据并行计算中，并行性起源于跨多个同时分离并行计算操作器或节点而分布大的数据集。作为对比，任务并行计算涉及跨多个同时分离并行计算操作器或节点而分布多个线程的执行。通常，硬件被具体地设计为执行数据并行操作。因此，数据并行程序是为数据并行硬件专门编写的程序。传统上，数据并行编程要求理解数据并行概念的非直观本质并密切熟悉所编程的具体数据并行硬件的高度熟练的程序员。

[0003] 在超级计算的领域之外，数据并行编程的常见用途是图形处理，这是因为这样的处理是规则的、数据密集的，且可以获得专用的图形硬件。尤其，图形处理单元(GPU)是被设计为从计算机的主中央处理单元(CPU)分担复杂的图形渲染的专用多核处理器。多核处理器是其中核的数量足够大以致传统的多处理器技术不再有效的处理器——此阀值在几十个核的范围内的某处。尽管多核硬件并不必定与数据并行硬件相同，但数据并行硬件可以通常被认为是多核硬件。

[0004] 其他现有的数据并行硬件包括可从现代主要处理器生产商获得的x86/x64处理器中的单指令多数据(SIMD)、流化SIMD扩展(SSE)单元。

[0005] 典型的计算机历史上是基于不是专门设计或能够承担数据并行性的传统单核通用CPU。因此，用于传统CPU的传统的软件和应用程序并不使用数据并行编程技术。然而，传统的单核通用CPU正被多核通用CPU代替。

[0006] 尽管多核CPU能够承担数据并行功能，但很少利用这样的功能。由于传统的单核CPU不能够承担数据并行，大多数程序员不熟悉数据并行技术。即使程序员感兴趣，仍然存在程序员完全地理解数据并行概念和学习足够多以便充分熟悉多核硬件以实现那些概念的大的障碍的需要。

[0007] 如果程序员清除了那些障碍，他们必须为他们希望他们的程序在其中运行的每一特定的多核硬件配置重新创建这样的编程。也就是说，因为常规的数据并行编程是硬件专用的，对一个数据并行硬件有效的特定解决方案将并不必定对另一数据并行硬件有效。由于程序员为具体的硬件编程他们的数据并行解决方案，程序员面临不同的硬件引起的兼容性问题。

[0008] 目前，不存在允许普通程序员执行数据并行编程的被广泛地采纳的、有效的和通用的解决方案。普通程序员是不完全理解数据并行概念且不密切熟悉每一不兼容的数据并行硬件场景的程序员。此外，不存在允许程序员(普通的或不不普通的)可以聚焦于正被编程的应用程序的高级逻辑而非聚焦于目标硬件级的具体实现细节的现有解决方案。

发明内容

[0009] 在此描述的是用于生成用于数据并行编程模型的调用存根以使得以静态编译的高级编程语言编写的数据并行程序可以比传统方法更加声明性、可重用和可移植的技术。借助于所描述的技术中的某一些，由编译器生成调用存根，且那些存根将数据并行计算的逻辑排列桥接到用于该数据并行计算的目标数据并行硬件的实际物理排列。

[0010] 在所描述的一些其他技术中，编译器映射由数据并行函数的单元数据并行计算(即，“内核”)期望的给定输入数据。

[0011] 提供本发明内容以便以简化形式介绍下面在具体实施方式中进一步描述的一些概念。本发明内容不旨在标识所要求保护的本主题的关键特征或必要特征。例如，术语“工具”可以指上述上下文和通篇文档所准许的设备、系统、方法、和/或计算机可读介质。

附图说明

[0012] 参考附图描述具体实施方式。附图中，附图标记最左边的数位标识该附图标记首次出现的图。在各附图中，使用相同的附图标记来指示相同的特征和组件。

[0013] 图1示出可用于实现在此描述的用于数据并行编程模型的技术的示例计算环境。

[0014] 图2和图3是一个或多个示例过程的流程图，每一流程图实现在此描述的技术中的某一些，尤其是涉及数据并行编程模型的那些技术。

[0015] 图4和图5是一个或多个示例过程的流程图，每一流程图实现在此描述的技术，包括涉及用于数据并行编程模型的编译器生成的调用存根的那些技术。

具体实施方式

[0016] 在此描述的是用来生成用于数据并行编程模型的调用存根以使得用静态编译的高级编程语言编写的数据并行程序可以比传统的方法更加声明性、可重用和可移植的技术。借助于所描述的技术中的某一些，编译器生成调用存根，且那些存根将数据并行计算的逻辑排列桥接到用于该数据并行计算的目标数据并行硬件的实际物理排列。换言之，调用存根桥接将通用化和逻辑数据并行实现(例如，数据并行程序)之间的间隙桥接到具体的和物理的数据并行实现(例如，在数据并行硬件中)。在一些所描述的技术中，编译器生成将给定的输入数据映射到数据并行函数的单元数据并行计算(即，“内核”)所期望的那些数据的代码。

[0017] 为达到一定程度的硬件无关性，各实现被描述为可以扩展为支持数据并行编程的通用编程语言的一部分。C++编程语言是在此描述的此类语言的主要示例。C++是静态类型的、自由形式的、多风格的编译性通用编程语言。C++可以也被描述为命令式的、过程式、面向对象的和泛型的。C++语言被认为是中级编程语言，这是因为它包括高级和低级语言特征两者的组合。本发明的概念不限于以C++编程语言的表达式。相反，C++语言可用于描述本发明的概念。可以使用的一些替换编程语言的示例包括Java™、C、PHP、Visual Basic、Perl、Python™、C#、Ruby、Delphi、Fortran、VB、F#、OCaml、Haskell、Erlang和JavaScript™。也就是说，所要求保护的本主题中的某一些可以覆盖以C++类型语言、命名法和格式的具体编程表达式。

[0018] 所描述的实现中的某一些提供了将软件开发者置于对与数据并行资源的交互的许多方面的显式控制中的基本编程模型。开发者分配数据并行存储器资源并启动访问该存

储器一系列数据并行调用点。在非数据并行资源和数据并行资源之间的数据传递是显式的且通常是异步的。

[0019] 所描述的实现提供了与以下各项的深度集成：可编译的通用编程语言(例如，C++)，以及适合根据问题域实体(例如，多维数组)而非硬件或平台域实体(例如，将偏移量捕捉到缓冲器中的C指针)来表达解决方案的抽象级。

[0020] 可以在诸如使用多核处理器或x64处理器中的SSE单元的那些数据并行硬件等的数据并行硬件上实现所描述的实施方式。可以在互连计算机的群集上实现一些所描述的实施方式，互连计算机中的每一个可能具有多个GPU和多个SSE/AVX™(高级矢量扩展)/LRBni™(Larrabee新指令)SIMD和其他数据并行协处理器。

[0021] 以下共同所有的美国专利申请通过引用合并于此，并构成本申请的以部分：于2010年6月18日提交的美国专利申请第12/918,097号，[其标题为：“数据并行编程模型”，于与本申请相同的日期提交，且具有共同的发明人权]。

[0022] 示例计算基础设施

[0023] 图1示出可以实现在此描述的技术的示例计算机体系结构100。体系结构100可以包括至少一个计算设备102，计算设备102可以经由网络104被耦合在一起以便与其他设备形成分布式系统。尽管不示出，但用户(通常是软件开发者)可以在编写数据并行(“DP”)程序的同时操作计算设备。也不示出的是，计算设备102具有输入/输出子系统，例如键盘、鼠标、监视器、扬声器等等。其间，网络104表示相互连接的充当单个大型网络(例如，因特网或内联网)的多中不同类型的网络中的任何一个或组合。网络104可以包括基于线缆的网络(例如，有线电视网络)和无线网络(例如，蜂窝式网络、卫星网络等等)。

[0024] 这一示例计算机体系结构100的计算设备102包括存储系统106、非数据并行(非DP)主机110和至少一个数据并行(DP)计算引擎120。在一个或多个实施方式中，非DP主机110运行通用的多线程的和非DP的工作量，且执行传统的非DP计算。在替换实施方式中，非DP主机110能够执行DP计算，但不能够执行所述DP编程模型所聚焦的计算。主机110(无论是DP或非DP)控制DP计算引擎120。主机110是操作系统(OS)运行在其上运行的硬件。尤其，主机在它正执行代码时提供OS进程和OS线程的环境。

[0025] DP计算引擎120执行DP计算和其他DP功能。DP计算引擎120是被优化为用于执行数据并行算法的硬件处理器抽象。DP计算引擎120也可以被称为DP设备。DP计算引擎120可以具有不同于主机的存储器系统。在替换实施方式中，DP计算引擎120可以与主机共享存储器系统。

[0026] 存储系统106是用于存储程序和数据的位置。存储系统106包括计算机可读介质，例如但不限于磁存储设备(例如，硬盘、软盘、磁条)、光盘(例如，紧致盘(CD)、数字多用盘(DVD))、智能卡和闪速存储器设备(例如，卡、棒、密钥驱动器)。

[0027] 非DP主机110表示非DP计算资源。那些资源包括例如一个或多个处理器112和主存储器114。驻留在主存储器114中的是编译器116和一个或多个可执行程序，例如程序118。编译器116可以是例如包括在此描述的实现的用于通用编程语言的编译器。更具体地，编译器116可以是C++语言编译器。调用存根118可以是来自编译器116的中间结果。调用存根118可以是例如以HLSL(高级着色语言)、C++或诸如通用中间语言(CIL)等的另一中间表示生成的代码。取决于编译模型是静态的还是动态的，在程序执行的同时调用存根118可以驻留在存

储器中。对于静态编译模型，这些存根可以与内核函数组合起来，并变成设备可执行的，在这一情况中，在进行编译之后，这些存根的中间形式可以不驻留在存储器中。在动态编译模型中，编译本身程序执行的部分；因此，在程序正执行的同时，中间存根可以驻留在存储器中。动态编译器可以将它与对应的内核函数组合，以便在运行时产生设备可执行指令。调用存根是设备可执行程序的入口点(在微软的应用程序编程界面的DirectX®平台上被称为着色器)。可以为每一并行调用点生成各存根。各存根基于物理线程索引和逻辑计算域计算每一并行活动的逻辑索引(即，它桥接了逻辑排列和物理排列)。然后，各存根经由投影或广播中的任一个将在并行调用点(例如，forall调用点)处提供的自变量映射到所调用的内核函数所期望的参数，并最终调用该内核函数。各存根包括用不同的输入数据和逻辑计算域为给定的并行调用定制内核函数。

[0028] 另一方面，程序119可以至少部分地是从编译器116的编译所得到的可执行程序。编译器116、调用存根118和程序119是计算机可执行指令的模块，计算机可执行指令是可以在计算机、计算设备或计算机的处理器上执行的指令。尽管在这里被示出为模块，但组件可以被具体化为硬件、软件或其任何组合。而且，尽管在这里被示出为驻留在计算设备102上，但组件可以跨分布式系统中的多个计算设备而分布。

[0029] 或者，可以把调用存根118看作是可执行程序119的一部分。

[0030] DP计算引擎120表示具有DP功能的计算资源。在物理层面上，具有DP功能的计算资源包括能够执行DP任务的硬件(例如GPU或SIMD及其存储器)。在逻辑层面上，具有DP功能的计算资源包括被映射到例如执行DP计算的多个计算节点(例如，122-136)的DP计算。通常，每一计算节点在能力方面是彼此相同的，但是每一节点被分离地管理。类似于图，每一节点具有其自己的输入和其自己的所期望的输出。节点的输入和输出的流是去往/来自非DP主机110或去往/来自其他节点。

[0031] 计算节点(例如，122-136)是DP硬件计算资源的逻辑排列。逻辑上，每一计算节点(例如，节点136)被排列为具有其自己的局部存储器(例如，节点存储器138)和多个处理元素(例如，元素140-146)。节点存储器138可以被用来存储是节点的DP计算的一部分且可以保存以往的一次计算的值。

[0032] 通常，节点存储器138分离于非DP主机110的主存储器114。计算引擎120的DP计算操纵的数据语义上分离于非DP主机110的主存储器114。如箭头150所指示，显式地将来自主存储器114中的通用(即，非DP)数据结构的各值复制到与DP计算引擎120相关联的数据的聚集(数据的聚集可以被存储为像节点存储器138那样的局部存储器的集合)，且显式地从与DP计算引擎120相关联的数据的聚集复制来自主存储器114中的通用(即，非DP)数据结构的各值。数据值到存储器位置的详尽映射可以受系统的控制(正如由编译器116指导的)，这将允许在存在充足的存储器资源时开发并行性。

[0033] 处理元素中的每一个(例如，140-146)表示DP内核函数(或简单地称为“内核”)的性能。内核是要执行的基本数据并行任务。

[0034] 操作输入数据集的内核被定义为字段。字段是所定义的基本类型的数据的多维聚集。基本类型可以是例如整数、浮点数、布尔或可用在计算设备102上的任何其他类别的值。

[0035] 在这一示例计算机体系结构100中，非DP主机110可以是带有其存储器的传统单核中央处理器单元(CPU)的一部分，且DP计算引擎120可以是在分立外围组件互连(PCI)卡上

或在与CPU相同的板上的一个或多个图形处理单元(GPU)。GPU可以具有分离于CPU的存储器空间的局部存储器空间。因此,DP计算引擎120具有其自己的局部存储器(如每一计算机节点的节点存储器(例如,138)所表示的),该局部存储器分离于非DP主机本身的存储器(例如,114)。借助于所描述的实现,程序员拥有对这些分离的存储器的访问权。

[0036] 替代示例计算机体系结构100,非DP主机110可以是多个CPU或GPU中的一个,其DP计算引擎120可以是剩余的CPU或GPU中的一个或多个,其中CPU和/或GPU在相同的计算设备上或在群集中操作。或者,多核CPU的核可以构成非DP主机110和一个或多个DP计算引擎(例如,DP计算引擎120)。

[0037] 借助于所描述的实现,程序员拥有使用熟悉的主流的函数调用的语法以及概念和传统的非DP编程语言(例如C++)来借助于支持DP的硬件创建DP功能的能力。这意味着普通程序员可以编写将传统的不支持DP的硬件(例如,非DP主机110)的操作用于任何支持DP的硬件(例如,计算引擎120)的程序。至少部分地,可执行程序118表示由普通程序员编写且由编译器116编译的程序。

[0038] 程序员为DP功能编写的代码在语法、命名法和方法上类似于为传统的非DP功能编写的代码。更具体地,程序员可以使用传递用于函数的数组变量的熟悉的概念来描述用于DP计算的基本函数的规范。

[0039] 根据所描述的实现产生的编译器(例如,编译器116)处理用于在支持DP的硬件上实现DP功能的许多细节。换言之,编译器116生成将DP计算引擎120的逻辑排列映射到物理DP硬件(例如,具有DP功能处理器和存储器)的代码。因此,程序员不需要考虑DP计算的所有特征来捕捉DP计算的语义。当然,如果程序员熟悉可以在其上运行程序的硬件,则该程序员仍然具有指定或声明如何执行特定操作和如何处理其他资源的能力。

[0040] 另外,程序员可以使用熟悉的数据集大小的观念来推断资源和成本。超出认知的熟悉性,对于软件开发者,这种新方法允许在非DP主机110和DP计算引擎120之间的类型和操作语义的公共规范。这种新方法简化了产品开发并使得DP编程和功能更平易近人。

[0041] 借助于这种新方法,介绍这些编程概念:

[0042] ●字段:预先定义的维度和元素数据类型的数据的多维聚集。

[0043] ●索引:被用来索引数据的聚集(例如,字段)多维向量。

[0044] ●网格:索引实例的聚集。具体地,网格指定多维矩形,多维矩形表示在该矩形中的所有索引实例。

[0045] ●计算域(例如,网格):描述可以请求数据并行计算的所有可能的并行活动的索引实例的聚集。

[0046] ●DP调用点函数:为四个DP调用点函数定义语法和语义;即,forall、reduce、scan和sortforall。

[0047] 字段

[0048] 在为传统的非DP硬件编程时,软件开发者常常定义包含应用程序数据的用户数据结构,例如列表和字典。为了使得从数据并行硬件和功能可能获得的好处最大化,新的数据容器给DP程序提供容纳的方式,且是指程序的数据的聚集。DP计算操作被称为“字段的这些新的数据容器。

[0049] 字段是DP代码操纵和变换的通用数据数组类型。它可以被看作是指定的数据类型

(例如,整数和浮点数)的元素的多维数组。例如,浮点数的一维字段可以被用来表示稠密浮点数向量。颜色的二维字段可以被用来表示图像。

[0050] 更多具体地,令float4是表示在计算机监视器上的像素的红、绿、蓝和图形保真值的4个32位浮点数的向量。假定监视器具有分辨率1200x1600,则:

[0051] field<2,float4>screen(grid<2>(1200,1600))是用于屏幕的好模型。

[0052] 字段不需要是定义的矩形网格。尽管通常在是仿射的索引空间上定义它,在它是多边形和多面体(polyhedral)或多面体(polytope)的意义上——即是说,它被形成为以下形式的有限数量空间的交集:

[0053] $f(x1,x2,...,xn)>=c$

[0054] 其中x1,x2,...,xn是N维空间中的坐标且'f'是这些坐标的线性函数。

[0055] 字段被分配在具体的硬件设备上。在编译时定义它们的基本类型和维数,同时在运行时定义它们的范围。在一些实现中,字段的指定的数据类型可以是整个字段的统一类型。字段可以以此方式表示:field<N,T>,其中N是数据的聚集的维数且T是基本数据类型。具体地,字段可以由类的这一通用家族来描述:

[0056]
```
template<int N, typename element_type>
class field {
public:
  field(domain_type & domain);
  element_type & operator[](const index<N>&);
  const element_type& operator[](const index<N>&) const;
……
};
```

[0057] 伪代码1

[0058] 字段被分配在具体的硬件设备基础(例如,计算设备102)上。在编译时定义字段的基本类型和维数,同时在运行时定义它们的范围。通常,字段充当数据并行计算的输入和/或输出。而且,通常,这样的计算中的每一并行活动负责计算输出字段中的单个元素。

[0059] 索引

[0060] 字段中的维数也被称为字段的秩。例如,图像具有为2的秩。字段中的每一维具有下界和范围。这些属性定义允许作为在给定的维度处的索引的数字的范围。通常,在C/C++数组中是这样的情况,下界默认为0。为了获取或设定字段中的特定元素,使用索引。索引是N元组,其中其组建中的每一个落在由对应的下界和范围值所建立的界限内。索引可以如下表示:索引<N>,其中索引是大小为N的向量,它可以被用来索引秩为N的字段。有效的索引可以以此方式定义:

[0061] $valid\ index=\{<i_0,...,i_{N-1}>|where\ i_k>=$

[0062] $lower\_bound_k\ and\ i_k<lower\_bound_k+extent_k\}$

[0063] 伪代码2

[0064] 计算域

[0065] 计算域是描述数据并行设备在执行内核的同时必须实例化的所有可能的并行线程的索引实例的聚集。计算域的几何形状与正处理的数据(即字段)强相关,这是由于每一数据并行线程做出关于它负责处理字段的什么部分的假设。通常,DP内核将具有单个输出字段且该字段的底层网格将被用作计算域。但是在每一线程负责计算16个输出值时它也可以是网格的一部分(例如1/16)。

[0066] 抽象地，计算域是对象索引值的集合的描述。由于计算域描述数据的聚集(即，字段)的形状，它也描述用于对数据的聚集迭代的隐含循环结构。字段是其中每一变量与某一域中的索引值的一一对应的变量的集合。字段被定义在域上，且逻辑上具有用于每一索引值的标量变量。在此，计算域可以简单地被称为“域”。由于计算域指定字段的每一维度的长度或范围，它也可以被称为“网格”。

[0067] 在典型的场景中，索引值的集合简单地对应于多维数组索引。通过将索引值的规范分解为分离的概念(被称为计算域)，可以跨多个字段使用该规范，且可以附加附加的信息。

[0068] 网格可以被这样表示：grid<N>。网格描述字段或循环嵌套的形状。例如，可以用二维网格描述在外部循环从0运行到N且然后在内部循环从0运行到M的双重嵌套循环，且第一维度的范围从0(包括在内)跨到N(不包括在内)，且第二维度的范围为在0和M之间。网格也被用来指定字段的范围。网格不持有数据。它们仅描述它的形状。

[0069] 基本域的示例是整数运算序列的叉积。运算序列被存储在这一类中：

[0070]
```
struct arithmetic_sequence {
   int lb, extent, stride;
   ArithmeticSequence(int n);
   ArithmeticSequence(int nl, int nu);
   ArithmeticSequence(int nl, int nu, int s);
} ;
```

[0071] 伪代码3

[0072] 这表示值的集合{lb+i*stride|0≤i<extent}。这些集的叉积被称为网格，其中集的数量被称为网格的秩。网格可以由以它们的秩区分的类型家族来表示。

[0073]
```
template<unsigned rank>
class grid : domain<rank> {
arithmetic_sequence values[rank];
public:
...
size_t extent(); //分量范围的积
arithmetic_sequence dimension(int i) { return
values[i]; }
   } ;
```

[0074] 伪代码4

[0075] 说到这里，各种构造器已经被省略，且各种构造器是专业特异的。域的秩或维数是该类型的一部分，因此可以在编译时获得它。

[0076] 资源视图

[0077] resource_view表示在给定的计算设备上数据并行处理引擎。compute_device是物理数据并行设备的抽象。在单个compute_device可以存在多个resource_view。事实上，resource_view可以被看作是执行的数据并行线程。

[0078] 如果不显式地指定resource_view，则可以创建默认的resource_view。在创建默

认值之后，在其上隐含地需要资源视图的所有将来的操作系统(OS)线程将获取先前所创建的默认值。可以从不同的OS线程使用resource_view。

[0079] 同样借助于这种新方法，资源视图允许在计算引擎120的上下文内指定和强加诸如优先级、最后期限调度和资源限制等的概念。域构造器可以可选地由资源视图参数化。这标识要用来持有数据的聚集的一组计算资源并执行计算。这样的资源可以拥有私有存储器(例如，节点存储器138)以及与非DP主机110的主存储器114非常不同的特性。作为逻辑构建，计算机引擎是指这一资源集。在此简单地将其看作是不透明类型：

[0080] typedef...resource_view;

[0081] 伪代码5

[0082] DP调用点函数

[0083] 借助于这种新方法，DP调用点函数调用可以被应用到与支持DP的硬件(例如，计算引擎120)相关联的数据的聚集以便描述DP计算。所应用的函数被注释为允许在DP上下文中使用。函数可以本质上是标量，这是因为期望它们使用和产生标量值，尽管它们可以访问数据的聚集。在并行调用中各函数被基本地应用到至少一个数据的聚集。在某种意义上，函数指定循环的主体，其中从数据的结构推断循环结构。到该函数一些参数只是被应用到数据的元素(即，流化)，尽管对于索引访问(即，非流化)来说数据的聚集也可以像数组一样被传送。

[0084] DP调用点函数将被称为内核的可执行代码片段应用到由计算域表示的每一虚拟的数据并行线程。此代码片段被称为“内核”，且是计算节点的每一处理元素(例如，140-146)执行的内容。

[0085] 在此描述的是表示四个不同的DP原语的四个不同的具体DP调用点函数的实现：forall、reduce、scan和sort。所描述的DP调用点函数中的第一个是“forall”函数。使用forall函数，程序员可以用单次函数调用生成DP嵌套循环。嵌套循环是其中一个循环位于另一循环体内的逻辑结构。以下是嵌套循环的示例伪代码：

[0086] for(int i=0;i<N;i++)

[0087] for(int j=0;j<=i;j++)

[0088] x(i,j)=foo(y(i,j),z(i,j));

[0089] …

[0090] 伪代码6

[0091] 在上面的嵌套循环的传统的串行执行中，外部循环(即i循环)的第一迭代引起执行内部循环(即j循环)。因此，对于i循环的每一迭代，在内部j循环中的示例嵌套函数“foo(y(i,j),z(i,j))”连续地执行j次。代替串行执行以传统的方式编写的嵌套循环代码，该新方法提供被称为“forall”的新的DP调用点函数，在“forall”被编译和被执行时，逻辑上并行执行嵌套函数的每一迭代(例如，“foo(y(i,j),z(i,j))”)(这被称为“内核”)。

[0092] 完美的循环嵌套是使得存在单个外部循环语句且每一循环体准确地是一个循环或是非循环语句的序列的循环的集合。仿射循环嵌套是使得存在单个外部循环语句和每一循环体是可能的循环语句的序列的循环的集合。在循环归纳变量中，仿射循环中的每一循环的界限是线性的。

[0093] DP调用点函数forall的至少一个实现被设计为将仿射循环嵌套映射到数据并行

代码。通常，以外部循环开始且只要接下来的循环是完美的就继续的仿射循环嵌套的一部分被映射到数据并行计算域，且然后，仿射嵌套的剩余部分被放置到内核中。

[0094] 在这里示出forall函数的伪代码格式：

```
template<typename index_domain,typename kernelfun,
typenameFields... >
```

[0095]

```
void forall(index_domain d,
kernelfun foo,
Field... fields) { ... }
```

[0096] 伪代码7

[0097] 对于由域“d”指定的每一索引，此函数调用的基本语义将来自对应于的字段的元素的自变量评估函数“foo”，就像在原始循环中一样。

[0098] 这是forall函数的伪代码的替换格式：

```
    grid<2> cdomain(height, width);
    field<2, double> X(cdomain), Y(cdomain),
Z(cdomain);
```

[0099]

```
    forall(cdomain,
     [=] __declspec(vector) (double &x,
                 double y,
                 double z ) {
        x = foo(y,z);
     }, X, Y, Z);
```

[0100] 伪代码8

[0101] 在上面的示例伪代码中，forall函数被示出为由λ操作符“[＝]”指示的λ表达式。λ表达式是可以构建表达式和语句的匿名函数且可以被用来创建委托或表达式树类型的匿名函数。

[0102] 另外，使用传值来修改双精度“y”和“z”的效果具有好处。在程序员以此方式标记自变量时，它将变量映射到只读存储器空间。因此，程序可以更快和更高效地执行，这是由于被写到该存储器区域的值维持它们的完整性，尤其在被分布到多个存储器系统时。因此，使用此“const”标签或另一等效标签，程序员可以在不存在对写回到该存储器区域中的需要时提高效率。

[0103] 在此描述另一具体的DP调用点函数是“reduce”函数。使用reduce函数，程序员可以计算非常大的值的数组的总和。在这里示出reduce函数的伪代码格式的两个示例：

```
template<typename domain_type,
 typename reducefun, typename result_type,
 typename kernelfun, typename Fields....>
void reduce(domain_type d,
    reducefun r, scalar<result_type> result,
    kernelfun f, Fields... fields) { ... }
```

[0104]

```
template<unsigned dim, unsigned rank,
 typename reducefun, typename result_type,
 typename kernelfun, typename Fields....>
void reduce(grid<rank> d,
    reducefun r, field<grid<rank-1>,result_type>
result,
    kernelfun f, Fields... fields) { ... }
```

[0105] 伪代码9

[0106] 在这些示例中，“f”映射到“result_type”的值。函数“r”组合此类型的两个实例并返回新的实例。它被假设为是关联的和可交换的。在第一种情况中，此函数被穷举地应用到对被存储在“result”中的单个结果值的归约。这一第二形式限于“网格”域，其中(由“dim”)选择一个维度，且通过归约来消除该维度，如上所述。同样地，经由函数“r”将“result_field”输入值与所生成的值组合。例如，此模式匹配矩阵乘累加:A＝A+B*C，其中计算网格对应于基本元组的3维空间。

[0107] 在此描述的又一个具体的DP调用点函数是“scan”函数。scan函数也被称为数据并行计算的“并行前缀”原语。给定值的数组，程序员可以使用scan函数来计算其中每一元素是输入数组中在它之前的所有元素的总和的新数组。在这里示出scan函数的示例伪代码格式：

```
template<unsigned dim, unsigned rank,
 typename reducefun, typename result_type,
 typename kernelfun, typename Fields....>
```

[0108]

```
void scan(grid<rank> d,
    reducefun r, field<grid<rank>,result_type>
result,
    kernelfun f, Fields... fields) { ... }
```

[0109] 伪代码10

[0110] 在归约情况中，“DIM”自变量选择通过该数据的“线束”。“线束”是数据集的较低的维度投影。例如，考虑10X10的范围的二维矩阵。则，线束将是第五列。或考虑数据的三维立方体，则，线束将是在Y＝Y0处的XZ平面。在归约情况中，该线束被减少为标量值，但是在这里该线束定义使用操作符“R”对其执行并行前缀计算的值的序列，操作符“R”在这里被假设为是关联的。这产生值的序列，然后值的序列被存储在“RESULT”的对应元素中。

[0111] 在此描述的四个具体的DP调用点函数中的最后的函数是“sort”函数。这如此函数的名称所暗示的那样，程序员可以使用一个或多个已知的数据并行排序算法来排序大的数据集。sort函数由比较函数、要排序的字段和可以由比较引用的附加字段。在这里示出sort函数的示例伪代码格式：

```
template<unsigned dim, unsigned rank
 typename record_type, typename Fields... >
void sort(grid<rank> d,
    int cmp(record_type&, record_type&),
    field<grid<rank>, record_type> sort_field,
    Fields... fields) { ... }
```

[0112]

[0113] 伪代码11

[0114] 如上所述，这一排序操作被应用到“DIM”维度中的线束并在其位置更新“SORT_FIELD”。

[0115] 其他编程概念

[0116] 基于DP调用点的自变量，DP调用点函数可以操作两种不同类型的输入参数：元素参数和非元素参数。因此，基于DP调用点生成的计算节点(例如，122-136)操作那两种不同类型的参数中的一种。

[0117] 对于元素输入，计算节点操作单个值或标量值。对于非元素输入，计算节点操作数据的聚集或值的向量。也就是说，计算节点具有在数据的聚集中任意索引的能力。对DP调用点函数调用将具有是元素输入或非元素输入的自变量。这些DP调用点调用将生成逻辑计算节点(例如，122-136)基于值与相关联的函数的自变量。

[0118] 一般地，元素计算节点的计算可以重叠，但是非元素计算机节点通常不这样。在非元素的情况中，将需要在任何节点访问数据的聚集中的任何特定元素之前在计算引擎存储器(例如，节点存储器138)中完全实现数据的聚集。因为基于非元素参数的数据的聚集可以递增地产生和使用，相比于使用元素输入参数，需要较少的存储器。

[0119] 对于DP调用点函数，内核形式参数类型不必匹配传递进来的自变量的实际类型。假设实际类型是秩Ra的字段且计算域具有秩Rc。

[0120] 1.如果形式的的类型不是字段，而是与实际的字段的基本类型相同的类型(模数常量和引用)，则，当：

[0121] $R_a = R_c$

[0122] 时存在有效的转换。

[0123] 2.如果形式类型是秩R_f的字段(模数常量和引用)，则，当：

[0124] $R_a = R_f + R_c$

[0125] 时存在有效的转换。

[0126] 3.为了完整起见，存在身份转换，其中形式类型和实际类型匹配：

[0127] $R_a = R_f$

[0128] 作为示出转换1的示例，考虑借助于内核的向量加法：

[0129]
```
_declspec(kernel)
```

[0130]
```
void vector_add(double&c,double a,double b)
```

[0131]
```
    c=a+b;
```

[0132]
```
}
```

[0133] DP调用点函数的实际参数是：

[0134]
```
grid domain(1024,1024);
```

[0135]
```
field<2,double>A(domain),B(domain),C(domain);
```

[0136] 然后，调用点采取以下形式：

[0137] forall(C.get_grid( ),vector_add,C,A,B);

[0138] 下列转换：

[0139] C->double&c

[0140] A->double a

[0141] B->double b

[0142] 通过将整个字段聚集与内核vector_add的完全相同地对待而工作。换言之，对于域中两个索引中的每一个：

[0143] index<2>idx1,idx2

[0144] C[idx1]和C[idx2](分别是A[idx1]和A[idx2]、B[idx1]和B[idx2])在内核vector_add中被同等地对待。

[0145] 这些转换被称为元素投影。如果内核不具有是字段的参数类型，内核被称为是元素的。元素的内核的优点中的一个包括可能竞争条件、死锁或活锁的完全缺失，这是因为在实际字段的任何元素的处理之间不做区分。

[0146] 作为示出转换2的示例，考虑借助于内核的向量加：

```
__declspec(kernel)
void sum_rows(double& c, const field<1, double>& a)
{
int length = a.get_extents(0);
//创建临时变量以便存取寄存器
//代替全局存储器
```

[0147]

```
double c_ret = 0.0;
//加上向量a
for (int k = 0; k < length; ++k)
c_ret += a(k);
//将结果指派给全局存储器
c = c_ret;
}
```

[0148] 伪代码12

[0149] DP调用点函数的实际参数是：

[0150] grid domain(1024,1024),compute_domain(1024);

[0151] field<2,double>A(domain),C(comput_domain);

[0152] 然后，调用点采取以下形式：

[0153] forall(C.get_grid( ),sum_rows,C,A);

[0154] 下列转换中：

[0155] C->double&c

[0156] A->const field<1,double>&a,

[0157] 第一个是覆盖在转换1上的元素投影。对于第二个，内核sum_rows作用于A的元素的左边的索引，同时计算域填充正确的索引。换言之，对于compute_domain中给定的‘索引<1>idx’,sum_rows的主体采取以下形式：

```
int length = a.get_extents(0);
        //创建临时变量以便存取寄存器
        //代替全局存储器
        double c_ret = 0.0;
[0158]  // sum the vector a
        for (int k = 0; k < length; ++k)
  c_ret += a(k, idx[0]);
        //将结果指派给全局存储器
        C[idx] = c_ret;
```

[0159] 这被称为部分投影，且优点之一包括在由计算域提供的索引汇中不存在共同并发缺陷的可能性。部分投影的一般形式使得内核作用于A的元素的索引的最右边的'Rf'数字，且剩余的索引由计算域填充，因此要求：

[0160] $R_a = R_f + R_c$。

[0161] 作为转换的稍微更复杂的示例，考虑：

```
[0162]  __declspec(kernel)
        void sum_dimensions(double& c, const field<Rank_f,
double>& a)
        {
          double c_ret = 0.0;
          for (int k0 = 0; k0 < a.get_extents(0); ++k0)
          for (int k1 = 0; k1 < a.get_extents(1); ++k1)
          .....
[0163]    for (int kf = 0; kf < a.get_extents(Rank_f - 1);
        ++kf)
          c_ret += a(k0, k1, ..., kf);
          c = c_ret;
          }
```

[0164] 伪代码13

[0165] 借助于实际参数：

[0166] const int N,Rank_f;

[0167] int extents1[N],extents2[N+Rank_f];

[0168] grid domain(extents2),compute_domain(extents1);

[0169] field<2,double>A(domain),C(comput_domain);

[0170] 然后，调用点采取以下形式：

[0171] forall(C.get_grid( ),sum_dimensions,C,A);

[0172] 对于以下转换：

[0173] A->const field<rank_f,double>&a,

[0174] 内核的体的一个解释包括：

[0175]
```
Let index<N> idx;
i0 = idx[0];
i1 = idx[1];
...
iN = idx[N-1]

double c_ret = 0.0;
for (int k0 = 0; k0 < a.get_extents(0); ++k0)
for (int k1 = 0; k1 < a.get_extents(1); ++k1)
.....
for (int kf = 0; kf < a.get_extents(Rank_f - 1);
++kf)
c_ret += a(k0, k1, ..., kf, i0, i1, ..., iN-1);
c(i0, i1, ..., iN-1) = c_ret;
```

[0176] 伪代码14

[0177] 稍微更复杂的示例是使用通信操作符转置和展形的矩阵乘法。

[0178] 假定‘field<N,T>A,transpose<i,j>(A)是用维度j对换维度i的结果。例如，在N＝2时transpose<0,1>(A)是正规矩阵转置：transpose<0,1>(A)(i,j)->A(j,i)。

[0179] 另一方面，spread<i>(A)是在索引I处添加空维的结果，这将所有随后的索引向右平移一。例如，在N＝2时，spread<1>(A)的结果是其中旧存储槽0保持相同但旧存储槽1被移动到存储槽2且存储槽1是空的三维字段：spread<1>(A)(i,j,k)＝A(i,k)。

[0180] 使用以下内核：

```
__declspec(kernel)
void inner_product(float& c,
        const field<1, float>& a,
        const field<1, float>& b) {
```

[0181]
```
double c_ret = 0.0;
for (int k = 0; k < a.get_extents(0); ++k)
c_ret += a(k)*b(k);
c = c_ret;
}
```

[0182] 伪代码15

[0183] 借助于实际参数：

[0184] grid domain(1024,1024);

[0185] field<2,double>C(domain),A(domain),B(domain);

[0186] 然后，矩阵乘法是以下DP调用点函数：

[0187]

```
forall(C.grid(), inner_product, C,
// spread<2>(transpose(A))(k,i,j) -> transpose(A)(k,i)
-> (i,k)
spread<2>(transpose(A)),
// spread<1>(B)(k,i,j) -> (k,j)
spread<1>(B));
```

[0188] 伪代码16

[0189] inner_product内核在最左边的存储槽(即k)处作用于A和B,且计算域正确地填充两个存储槽。实质上,展形只是被用来保持索引操纵的清洁和连贯。

[0190] 局部投影的最后的示例使用DP调用点函数'reduce'来计算矩阵乘法。

[0191] 例如,采取以下:

[0192]

```
reduce<1>(grid<3>(
// i, j, k
c.grid(0),c.grid(1),a.grid(1)),

//执行实际规约的变换
[=](double x, double y)->double{ return x + y; },

//目标
c,

//映射
[=](double x, double y)->double{ return x * y; },

// spread<1>(a)(i,j,k) => i,k
spread<1>(a),

// spread<0>(transpose(b))(i,j,k) => transpose(b)(j,k)
=> (k,j))
spread<0>(transpose(b));
```

[0193] 伪代码17

[0194] 由于reduce作用于最左边的索引,转置和展形的使用与先前不同。解释是reduce<1>减少了计算域最左边的维度。

[0195] 两个函数(即λ)被类似地用于map-reduce:

```
    map:{{x0,y0},{x1,y1},...,{xN,yN}}=>{map(x0,y0),
  map(x1, y1), ..., map(xN, yN) };
    reduce: => transform(map(x0, y0), transform(map(x1,
  y1), transform(... , map(xN, yN))...));
```

[0196] 或：

```
    map: { {x0, y0}, {x1, y1}, ..., {xN, yN} } => { x0*y0,
  x1*y1, ..., xN*yN };
    reduce: => x0*y0 + x1*y1 + ... + xN*yN;
```

[0197] 伪代码18

[0198] 作为示出转换3的示例，令N=K+M，且考虑：

[0199]

```
__declspec(kernel)
void sum_dimensions(const index<K>& idx,
          field<K, double>& c,
               const field<N, double>& a) {
double c_ret = 0.0;
for (int k0 = 0; k0< a.get_extents(0); ++k0)
for (int k1 = 0; k1 < a.get_extents(1); ++ k1)
.....
for (int kM-1 = 0; kM-1< a.get_extents(M - 1); ++kM-1)
c_ret += a(k0, k1, ..., kM-1, idx[0], idx[1], ..., idx[K-1]);
c[idx] = c_ret;
}
```

[0200] 伪代码19

[0201] const int K,M,N;//N=K+M

[0202] int extents1[K],extents2[N];

[0203] grid domain(extents2),compute_domain(extents1);

[0204] field<2,double>A(domain),C(compute_domain);

[0205] 伪代码20

[0206] 然后，调用点采取以下形式：

[0207] forall(C.get_grid( ),sum_dimensions,C,A);

[0208] 且所有转换都是身份转换。

[0209] 在设备上创建存储器时，它原始地开始，且然后可以具有只读或读写的的视图。只读的优点之一包括在多个设备之间分解问题时(有时被称为核外(OUT-OF-CORE)算法)不需要检查只读存储器以便了解它是否需要更新。例如，如果设备1正操纵存储器块字段1，且设备2正使用字段1，则，不存在设备1检查是否字段1已经被设备2改变的需要。相似的图片适用于将存储器块用作字段的主机和设备。如果存储块是读写，则，在设备1和设备2上的动作之间将需要存在同步协议。

[0210] 在第一次创建字段时，它只是原始存储器且并没有为访问做好准备；也就是说它

还不具有‘视图’。在字段被传送到DP调用点函数处的内核时，参数类型的签名确定它是否将具有只读视图或读写视图(可以存在相同的存储器的两个视图)。

[0211] 如果参数类型是传值或传址常量，则将创建只读视图，即是说，对于某种类型的‘element_type’

[0212] element_type x

[0213] field<N,element_type>y

[0214] const field<N,element_type>&z

[0215] read_only_field<field<2,element_type>>w

[0216] 伪代码21

[0217] 如果该参数类型是非常量引用类型，则将创建读写视图：

[0218] element_type&x

[0219] field<N,element_type>&y.

[0220] 通过使用通信操作符，字段可以显式地被限制为只具有只读视图，其中它不具有读写视图，通信操作符为：

[0221] read_only

[0222] READ_ONLY操作符通过仅定义常量存取器和索引操作符以及下标操作符来工作，且因此：

[0223] read_only(A)

[0224] 以引起写的方式被使用。尤其，如果被传送到内核中(通过DP调用点函数)，则可以发生编译器错误。

[0225] 例如，在一种实施方式中，以下之间将存在差别：

[0226] element_type x

[0227] field<N,element_type>y

[0228] const field<N,element_type>&z

[0229] 以及

[0230] element_type&x

[0231] field<N,element_type>&y.

[0232] 伪代码22

[0233] 同时在另一实施方式中，以下之间将存在差别：

[0234] element_type x

[0235] read_only_field<field<2,element_type>>w

[0236] 以及

[0237] element_type&x

[0238] field<N,element_type>y.

[0239] 第一实施方式使用传值而非传址以及常量而不是非常量以便区分只读和读写。第二实施方式仅对元素形式参数使用传值而非传址，另外对字段形式参数，它使用READ_ONLY_field而非字段，以便以便区分只读和读写。第二实施方式的理由在于，在设备和主机具有不同的存储器系统时，引用实际上是引用谎言。

[0240] 关于DP Forall函数的更多内容

[0241] forall函数是高度通用的DP原语。可以使用forall函数来起动DP活动的主机。编译器116将函数的调用扩展到准备数据结构的代码序列，且最终起动并行活动。编译器166也生成将起动点粘合到计算(即，forall函数的内核参数)的存根(例如，调用存根118)，将物理线程索引映射到计算域中的当前并行活动逻辑索引，且如果内核函数请求就将这些专用的逻辑索引传递给内核函数，实现投影语义，并且最终调用内核函数。编译器116执行至少两个函数：

[0242] ●编译器116将DP函数的调用(例如forall)扩展到准备必要的数据结构的非DP可执行的代码的序列(其将在非DP主机110上执行)，且最终起动并行活动。

[0243] ●编译器116也生成存根(即设备代码，例如HLSL)将起动点粘合到单元计算(DP函数的内核参数)，实现投影语义和DP函数的特殊索引令牌以及目标设备的所有其他安装。

[0244] 编译器116使得程序员免受这样的负担，并使得他们聚焦于编程程序本身的功能。另外，将此工作留给编译器开发出数据并行程序开发一次就在多个目标上运行的机会。也就是说，编译器可以为不同的目标生成不同的调用存根，且取决于运行时请求，使得运行时选择适当的存根。

[0245] 以下是DP原语forall的伪代码的另一示例，且在这一示例中，用三个参数调用内核：

[0246]
```
            /// <summary>
            ///用三个自变量并行执行内核
            /// </summary>
            template <typename _Callable_type,
    typename _Actual_type1,
    typename _Actual_type2,
    typename _Actual_type3,
    size_t _Compute_rank>
            void forall(const grid<_Compute_rank>&
_Compute_grid,
    const _Callable_type & _Kernel,
    const _Actual_type1 & _Actual1,
    const _Actual_type2 & _Actual2,
    const _Actual_type3 & _Actual3);
```

[0247] 伪代码23

[0248] 第一参数(“const grid<_Compute_rank>&_Compute_grid”)是计算域，它描述如何部署并行活动。也就是说，如何可以并行完成工作以及部署如何映射回到数据，等等。第二参数(“const_Callable_type&_Kernel”)内核，该内核是单元计算，且在并行中将被所有并行活动调用。其余的参数(Actual1,Actual2,Actual3)是对应于内核的参数的自变量。所描述的技术的至少一种实现在此聚焦于在这些forall自变量和内核的参数之间进行映射。

[0249] 在此使用这些术语：

[0250] ●参数秩。(即，类型的DP秩)这是第一模板自变量。例如，field<2,int>的秩是2。对于不同于字段的类型，其秩是0。

[0251] ●实际参数秩(R_a)。被传送给DP函数的实际自变量的秩，例如forall(上面示例伪代码13中的Actual_type1～Actual_type3)。

[0252] ●形式参数秩(R_f)。内核的形式参数类型的秩。

[0253] ●计算秩(R_c)。在网格的实例被用作forall的第一参数时。例如，如果第一自变量的类型是grid<2>，则计算秩是2。

[0254] 类型的DP秩引入被称为“可索引的类型”的概念，该概念是字段的泛化。可索引的类型是像字段那样具有秩(被命名为秩的编译时已知的静态常量成员)的类型，和实现操作符[]that采取an实例of索引<秩>as输入，和返回an元素值或引用。可选地，它可以实现完全的或局部投影函数。这样的类型的参数秩是给定的可索引的类型的秩。不可索引的类型的参数秩是0。

[0255] 实际参数秩(R_a)并不必定与形式参数秩(R_f)相同。以下是只是添加两个值并将其和指派给第一参数的内核的简单示例：

[0256] _declspec(vector)void add(int&c,int a,int b)

[0257] {

[0258] c=a+b;

[0259] }

[0260] 伪代码24

[0261] 可以以这一方式中调用这一内核：

[0262]
```
grid<1> vector(rv, N);
field<1, int> fA(vector);
field<1, int> fB(vector);
field<1, int> fC(vector);
…
forall(vector, add, fC, fA, fB);
```

[0263] 伪代码25

[0264] 伪代码15的这一示例起动N个并行活动，每一并行活动添加分别来自两个向量(fA和fB)的一个元素，且将结果存储回到结果向量(fC)的对应存储槽。在这一示例中，R_c是1，R_a是(1,1,1)且R_f是(0,0,0)，且编译器116生成将R_a桥接到R_f的粘合代码。编译器116标识当前并行活动在整个计算中位置，从fA和fB载入对应的元素，调用内核函数，并将结果存储fC的正确元素。

[0265] 注意，对于不同的数据可以调用相同的内核。例如，程序员可以编写下列代码来调用对而非向量的加法计算：

[0266]
```
grid<2> matrix(rv, M, N);
field<2, int> fA(vector);
field<2, int> fB(vector);
field<2, int> fC(vector);
…
forall(matrix, add, fC, fA, fB);
```

[0267] 伪代码26

[0268] 不需要编译器116关心调用存根，程序员需要编写用于数据输入的每一可能的种类的核计算算法(在这里是add函数)的包装器或多次复制带有轻微不同的结构的相同算法。使得程序员免受此负担显著地改善了核内核代码的可重用性，且使得整个程序整洁得多和简洁得多。

[0269] 当然，对于一些算法，每一内核需要看到整个输入数据以便执行所要求的计算，在这种情况中，R_f是与R_a相同的。在这一实例中，调用存根的任务只是标识当前并行活动的位置且然后，用所传递的参数调用内核。

[0270] 有时，并行活动在整个并行部署中的位置是编程内核所需要的。在这一情况中，内核可以经由定义具有计算秩的相同秩的索引类型参数且在forall的对应参数位置处使用专用的令牌(例如，_index)来获取这一位置信息。另一备选方案涉及允许内核函数具有被提供给forall调用的自变量多一个的参数，且内核函数的第一参数必须是index<compute rank>。在这一情况中，编译器将在存根中生成将逻辑索引(并行活动在整个并行部署中的位置)传递给内核函数的代码。

[0271] 以下内核示例计算矩阵乘法的一个元素：

[0272]

```
    __declspec(vector) void mxm_kernal_naive(index<2> cIdx,
 float & c, const field<2, float>& mA, const field<2, float>&
 mB)
    {
    c = 0.0;
    for (unsigned int k = 0; k < mA.get_extent(1); k++)
    {
    index<2> aIdx(cIdx[0], k);
    index<2> bIdx(k, cIdx[1]);
    c += mA[aIdx] * mB[bIdx];
    }
    }
```

[0273] 伪代码27

[0274] 给定以上内核，程序可以调用类似于此来它(例如)：

[0275]

```
          grid<2> gA(rv, M, W), gB(rv, W, N), gC(rv, M, N);
          field<2, float> mA(gA), mB(gB), mC(gC);
          ...
          forall(mC.get_grid(), mxm_kernel_naive, mC, mA,
  mB);
```

[0276] 伪代码28

[0277] 编译器116在表示当前并行活动的位置的调用存根中生成创建index<2>的正确实例的代码，并将它作为其第一自变量传递给mxm_kernel_naive。

[0278] 可以使用的两个其他特殊索引令牌包括："_tile_index"和"_local_index"。它们表示在平铺式计算域上调用内核时当前并行活动的位置。_tile_index给予当前并行活动所属于的平铺的索引，且_local_index提供当前并行活动在该tile内的位置。可以以与_index相似的方式使用这两个令牌，且如果在forall调用点请求它们中的任何之一，则编译器116在调用存根中生成代码以便适当地设置它们并将它们传递给内核。

[0279] 或者，代替专用的令牌，可以使用新的类型index_group来封装所有可能的特殊索引、全局变量、本地变量或组。各令牌可以是隐含的forall自变量，且如果目标内核请求则各令牌将被编译器插入。

[0280] forall调用的实际的自变量是将被传递给目标计算引擎的数据。它们与内核的参数一一对应，但每一自变量的类型并不必定与其对应的参数类型相同。对forall，允许三个自变量种类：

[0281] ●实例field<N,T>，其指示设备存储器上的存储器区域。这将被映射到设备资源，例如缓冲器。

[0282] ●非字段类型数据结构，其将被传值复制到设备。在一些平台上，经由专用的常量编组机制而传送这样的数据。

[0283] ●特殊索引令牌。这些令牌对应于当前并行活动在计算域中的位置。提供它们以使得程序员可以编写无需显式地被暴露给目标计算引擎的线程部署模型非元素内核。为使用_index，程序员将内核参数定义为是index<N>，其中N是计算域的秩，且在FORALL调用点的对应的自变量位置处使用所期望的令牌。_tile_index和_local_index被用于并行活动的tiled部署。

[0284] 参数投影是DP函数(例如，forall)的实际的自变量的秩，且并不必定与其对应的内核参数的秩相同。如果$R_f+R_c=R_a$，则编译器生成使用给定的并行活动的位置信息来将给定的字段自变量投影到内核期望的适当的秩的代码。如果$R_f+R_c>R_a$，则进行自动平铺也是可能的。

[0285] 线程部署策略和索引映射

[0286] 在一些场景(例如，微软的DirectCompute™平台)中，线程可以在三维度空间中分派，其中使用三个值来指定一组线程的形状，且使用三个附加的维度来指定通常被称为线程组的被调度在一起的线程的块的形状。

[0287] 在生成调用存根时，编译器(例如，编译器116)选择线程部署策略，且它生成基于目标相关计算单元身份(在运行时计算单元身份是可用的，对编译器来说是不可访问的)标识(例如，映射)当前并行活动在计算域中的位置的对应的索引映射代码。作为这一活动的一部分，编译器拾取物理域来覆盖逻辑域并这样分派它。编译器也生成代码，该代码将由每一线程执行，将三维分块物理线程域中给定的点映射回到逻辑计算域中的点。

[0288] 指导索引映射的选择一些考虑包括(作为示例而非限制)：

[0289] ●在映射应易于定义新的逻辑域和这些逻辑域到多样的物理域的映射的意义上，映射可以是容易地可扩展的。

[0290] ●存在用于线程组的范围的推荐大小。例如，它们不应太小的(存在许多其他这样的大小考虑，且不是这一背景描述的话题。)。

[0291] ●从物理到逻辑的逆映射应易于计算和优选地指导以使得：

[0292] a.在内核中的线程的实际计算的这一前奏中不耗费太多的时间。

[0293] b.在物理域和逻辑域之间的映射对编译器来说是可见的，以使得编译器可以允许优化，这取决于物理索引的用法。例如，如果使用整数变量i访问数组，则编译器知道i如何与物理域中的线程标识符相关是有益的。

[0294] ●所调度的冗余线程的量被消除或为0。

[0295] ●“覆盖”逻辑域所需要的平台API调用的次数是低的(或者，在理想情况是1比1)。

[0296] 逻辑计算域是0基稠密(单元跨距)立方体形状的域，以范围$E=<E_0,..,E_{N-1}>$的N维非负向量为特征，以使得当且仅当对i的每一分量i_j来说保持$0<=i_j<E_j$时index i=<

$i_0,...i_{N-1}$>是在该域中。例如，在二维的情况中，立方体简单地是带有锚固在<0,0>原点的角的矩形，且对于某种非负E_0和E_1，对角是在<E_0,E_1>处。在此，为简洁起见，立方体形状的域将被称为网格。

[0297] 为帮助描述，我们使用在这一文档中使用以下记法：

[0298] • (Gz,Gy,Gx)——线程组的范围(Gx是最不重要的维度)

[0299] • (Tz,Ty,Tx)——每一线程组的范围(Tx是最不重要的维度)

[0300] • gid——SV_GroupID,(gid[0],gid[1],gid[2])是(Gz,Gy,Gx)中的点

[0301] • gtid——SV_GroupdThreadID,(gtid[0],gtid[1],gtid[2])是(Tz,Ty,Tx)中的点

[0302] • dtid——SV_DispatchThreadID,(dtid[0],dtid[1],dtid[2])是(Gz*Tz,Gy*Ty,Gx*Tx)中的点

[0303] 存在用于在逻辑域和物理域之间映射的多种方法，下面提供这些方法。假定compute_grid的实例，即g。

[0304] 方法1：网格的Naïve映射

[0305] 线程分派：主机可以确定物理网格的维度以便因而分派(作为示例)：

[0306] 1.(Tz,Ty,Tx)=(1,1,256)，可以基于场景而调谐可这一预定义的数字；

[0307] 2.(Gz,Gy,Gx)=(1,1,div_ceil(g.total_elements(),256))；如果Gx>MAX_GROUPS，则报告错误。伪代码可以类似于此(例如)：

[0308]
```
template<size_t N>
void get_dispatch_extents1(extents<N> g, size_t & Gx,
size_t & Gy, size_t & Gz)
{
Gx = div_ceil(g.total_elements(), 256);
Gy = 1;
Gz = 1;

if (Gx >= 64K) throw exception(..);
}
```

[0309] 伪代码29

[0310] “div_ceil”采用两个整数a和b，且在C++整数math中执行(a+b-1)/b。换言之，它将各数字作为有理数字来除，并将结果上舍入到下一个整数。

[0311] 索引映射代码生成：给定gid/gtid/dtid，将其展平到线性偏移，且然后，基于g的范围将它提高到N维逻辑域中的索引。伪代码可以类似于此(例如)：

```
//给定0和线程总数减一之间的线性偏移，
//将其提高到器形状在'extents'中给出的N维网格中的索引

template <int N>
index<N> raise(__int64 offset, extents<N> shape)
{
index<N> idx;
__int64 extent = shape.total_elements();
for (size_t i = 0; i < N; i++) {
extent /= shape[i];
idx[i] = int(offset / extent);
offset = offset % extent;
}
return idx;
}
```

[0312]
```
// dtid -分配线程id, SV_DispatchThreadID
// g -- compute_grid示例
template<size_t N>
void stub1(index<3> dtid, compute_grid<N> g)
{
//在物理域中我们只使用最不重要的维度(DimX ).
size_t offset = dtid[DimX];

//过滤掉不能映射到逻辑域的额外物理线程
if (offset < g. total_elements())
{
index<N> logical_idx = raise<N>(offset, g.m_shape);

... //执行内核体
}
}
```

[0313] 伪代码30

[0314] 例如，如果计算网格是289X2050，则物理线程域可以被排列为：

[0315] (Tz,Ty,Tx)=(1,1,256)

[0316] (Gz,Gy,Gx)=(1,289,9)

[0317] SV_DispatchThreadID的维度将是(1,289,2304)，它与带有一些额外点的逻辑计算网格几乎相同。现在给定在空间(1,289,2304)中的dtid(该变量名称通常是指DirectCompute三维线程ID)，可以检测它是否在边界(289,2050)内(忽略dtid[DimZ]，这是由于在这一情况中它是0)。如果它确实在边界内，则dtid可以被直接地用作逻辑域中的索引而无需任何索引展平和提高(在处理逻辑域和物理域之间的维度排序的差异之后)。除了节省一些数学操作的成本之外，该二维物理ID也可以较好捕捉在二维线程网格和程序访问的二维数据结构之间的关系。这对向量机中的代码生成和编译时边界检查来说是相关的。

[0318] 方法2：为秩1、2和3优化

[0319] 线程分派：主机可以确定物理网格的维度以便因而分派(作为示例)：

[0320] 1.(TZ,TY,TX)=(1,1,256),可以基于场景调谐这一预定义数字。在这里假设这些范围是通过恒定向量THREAD_GROUP_EXTENTS传送的。

[0321] 2.如果秩是1、2和3,则试图将物理域中的相同维数用作逻辑域。分派决定(GZ,GY,GX)经由PHYSICAL_GROUP_GRID被传送出去。伪代码可以类似于此(例如):

[0322]

```
template<size_t N>
void get_dispatch_extents2(
compute_grid<N> g, extents<3>& physical_group_grid)
{
if (N <= 3)
{
for(i = 0; i < 3-N; i++)
{
physical_group_grid[i] = 1;
}

for(i = 3-N; i < 3; i++)
{
physical_group_grid[i] = div_ceil(g[i-(3-N)],
thread_group_extents[i]);
}
if (max(physical_group_grid) < MAX_GROUPS) return;
}

//均匀地将逻辑域中的点分不到3个物理域
__int64 total =
div_ceil(g.total_elements(),
thread_group_extents.total_elements());
physical_group_grid[0] = ceil(pow(double(total),
double(1)/dobule(3)));
__int64 leftover = div_ceil(total,
physical_group_grid[0]);
physical_group_grid[1] = ceil( pow(double(leftover),
double(1)/dobule(2)));
physical_group_grid[2]=div_ceil(leftover,physical_gro
up_grid[1]);

if (max(physical_group_grid) >= MAX_GROUPS) throw
exception(...);}
```

[0323] 伪代码31

[0324] 注意如果N<=3get_dispatch_extents2如何试图将每一逻辑维度映射到对应的物理维度。如果这是不可能的,方法跨三个物理维度“展形”整个所要求的组的集(由变量“total”捕捉)。Gz,Gy,Gx的值(它们是被存储在输出数组physical_group_grid中的值)被选择为接近必要的各组的总数的立方根。Gz、Gy和Gx的积可以实际上大于所要求的组的数量。通过使用首先舍入(在Gz进行)来进一步除以剩余的组的数量以便调度的结果,可以对

“浪费”组的数量进行归约。这可以得到用于Gy且随后用于Gx的稍低的剩余值。

[0325] 这只是以下一般任务中的优化的一种形式：

[0326] 查找G的值以使得

[0327] 每一Gi＜MAX_GROUPS

[0328] Gz*Gy*Gx＞＝所要求的组的总数

[0329] Gz*Gy*Gx是最小值

[0330] 使得Gz*Gy*Gz等于(准确地)“总数”是的完全匹配是最优的结果，但是这样的因素并不总是存在(例如如果“总数”是质数)。一般地，总数的值越大，运行时就越乐意视图优化Gz，Gy，Gx的选择。然而，借助于MAX_GROUPS等于64K且基于来自上述方法的基于立方根的解，浪费组的量是零点几个百分比。因此，立方根方法可用于实践目的。

[0331] 给定用于确定调度网格的维度的上述的代码，这是存根如何从物理索引恢复逻辑索引。给定gid/gtid/dtid，基于秩N，且给定上面选择哪种调度策略(直接映射还是非直接映射)，生成不同的索引计算代码。

[0332] 情况1:N＝＝1，直接地映射：

```
// dtid -分派线程id, SV_DispatchThreadID
// g -- compute_grid实例
void stub2(index<3> dtid, extents<1> g)
{
index<1> logical_idx(dtid[DimX]);
```

[0333]
```
//过滤掉不能映射到逻辑域的额外物理线程
if (logical_idx[0] < g.total_elements())
{
... //在这里执行内核体
}
}
```

[0334] 伪代码32

[0335] 情况2:N＝＝2，直接地映射：

```
  // dtid -分派线程id, SV_DispatchThreadID
  // g -- compute_grid实例
  void stub(index<3> dtid, extents<2> g)
  {
  index<2> logical_idx(dtid[DimY], dtid[DimX]);
```

[0336]
```
  //过滤掉不能映射到逻辑域的额外物理线程
  if (g. within_boundary(logical_idx))
  {
  ... //在这里执行内核体
  }
}
```

[0337] 伪代码33

[0338] 情况3:N＝＝3，直接地映射：

```
// dtid --分派线程id，SV_DispatchThreadID
// g -- compute_grid实例
void stub(index<3> dtid, extents<3> g)
{
index<3> logical_idx(dtid[DimZ], dtid[DimY],
dtid[DimX]);
//过滤掉不能映射到逻辑域的额外物理线程
if (g.within_boundary(logical_idx))
{
... //在这里执行内核体
}
}
```

[0339]

[0340] 伪代码34

[0341] 情况4:间接地映射,对于任何N:

```
// dtid --分派线程id，SV_DispatchThreadID
// g -- compute_grid实例
// Gx, Gy, Gz - The physical dispatch shape
void stub(index<3> dtid, compute_grid<N> g,
extents<3> physical_group_grid)
{
physical_group_grid
extents<3> physical_grid = physical_group_grid *
thread_group_extents;
__int64 offset = flatten<3>(dtid, physical_grid);
//过滤掉不能映射到逻辑域的额外物理线程
if (offset < g.total_elements())
{
index<N> logical_idx = raise<N>(offset, g);
... //在这里执行内核体
}
}
```

[0342]

[0343] 伪代码35

[0344] 方法“flatten”是提高的互逆,它从给定的网格中的点平移到整数范围[0,总的元素)中的线性偏移。在这里是其定义:

```
template <int N>
__int64 flatten(index<N> index, extents<N> shape)
{
__int64 multiplier = 1;
__int64 offset = 0;
for (int d=N-1;d>=0;d--)
{
   offset += multiplier * index[d];
   multiplier *= shape[d];
}
return offset;
}
```

[0345]

[0346] 伪代码36

[0347] 方法3:在max(Gx,Gy,Gz)>=64K时多次内核调用

[0348] 在方法#2中,物理维度匹配逻辑维度以使得对于秩=1、2和3索引映射是直接的,除非一个或多个维度需要展形。直接映射一般地更为有效,且较好地较好允许编译器优化。在展形必要时或在逻辑秩大于3时的情况中,计算必要的组的总数,且通过在所需要的组的总数的立方根周围绕轴旋转,这一数量跨三维度而相当平面地展形。因而,在它可以处理可能地可映射到单次计算调用中的硬件中的几乎所有输入意义上,先前的方法是相当通用的。给定MAX_GROUPS^3是2^48,来自先前的部分的方法#2所覆盖的范围将不满足任何计算是不太可能。

[0349] 方法#2仍然涉及非直接映射,非直接映射可以以存根中的分离的代码路径结束,且可以导致大的二进制大小,这是由于对于秩=1、2和3,两个存根的版本通常保持分别处理直接映射和非直接映射,这取决于对于给定的逻辑域中的动态范围直接映射是否可行。

[0350] 这一部分呈现缓和前述问题的替换方法#3。基本上,对于N<=3,如果逻辑域天然地落在向量G内,则,使用多次调用,每次"覆盖"来自逻辑网格的不同的部分。

[0351] 例如,假设逻辑网格=(258,258),但是对于每一维度的物理限制是256(即,MAX_GROUPS是256)。同样出于阐述的简单起见,假设线程组的范围是(1,1,1)。因此这意味着逻辑上使用(1,258,258)线程网格,但是,当然,此网格不能被调度,这是因为X维度和Y维度超出了MAX_GROUPS。相反,API平台可以被调用四次以便覆盖类似于此定义的逻辑空间(4次分派):

[0352] ●Gz,Gy,Gx)=(1,256,256),原点[0,0,0]

[0353] ●(Gz,Gy,Gx)=(1,256,2),原点[0,0,256]

[0354] ●(Gz,Gy,Gx)=(1,2,256),原点[0,256,0]

[0355] ●(Gz,Gy,Gx)=(1,2,2),原点[0,256,256]

[0356] 其他排列也是可能的,且系统将也有时具有并行执行这些分派中的某一些或在设备或计算机之间分布它们的机会。

[0357] 此原点信息可以被传送给存根,且在计算逻辑索引时被使用。现在,用于索引计算的伪代码可以类似于此(例如):

[0358] 情况1:N==1(总是直接地映射,带原点)

[0359]
```
// dtid -分配线程id, SV_DispatchThreadID
// dtid_origin -给定分派的原点
// g -- compute_grid实例
void stub3(index<3> dtid, index<3> dtid_origin,
extents<1> g)
{
index<1> logical_idx(dtid[DimX] +
dtid_origin[DimX]);
//过滤掉不能映射到逻辑域的额外物理线程
if (logical_idx[0] < g.total_elements())
{
... //在这里执行内核体
}
}
```

[0360] 伪代码37

[0361] 直接映射方法类似地适宜于N＝2和N＝3。因此，在情况N＜＝3的情况中，映射是直接加原点向量，这不同于先前的方法(#2)，方法(#2)在一些情况中即使在N＜＝3时(且总是在N＞3时)采用非直接映射。

[0362] 现在对于其中N＞3情况，存在两个选项。第一，可以使用方法#2中的非直接映射，但是更多地符合当前算法的精神，可以通过在N-3高次维度来“覆盖”N维度空间，且然后，N-3维度空间中的每一固定点，将3维索引映射算法映射到其余的剩余3个维度。伪代码可以类似于此(例如)：

```
//
//在N维网格上分派的主机代码，其中N>3
//
foreach(index<N-3> dtid_prefix in g.prefix<N-3>())
{
安排用被传递到下面的存根函数的"dtid_prefix"来g.suffix<3>上
分派
}
```

[0363]

```
// dtid --分派线程id，SV_DispatchThreadID
// dtid_origin --给定分派的原点
// dtid_prefix -N-3维前缀
// g -- compute_grid实例
void stub3(
index<3> dtid,
index<3> dtid_origin,
index<N-3> dtid_prefix
extents<N> g)
{
index<N>
logical_idx=dtid_prefix.concatenate(dtid_origin+dtid)
;
//过滤掉不能映射到逻辑域的额外物理线程
if (g.within_bounds(logical_idx))
{
... //在这里执行内核体
}
}
```

[0364] 伪代码38

[0365] 方法4:无需多次调用的更积极的线程利用

[0366] 在方法#3中，在秩是1、2或3时的情况，为了优化索引展平和提高计算，在逻辑域时不能很好地适应物理域时，一个逻辑分派被分成若干个。在此部分中，方法4提供另一选项：试图直接地将逻辑映射到物理，但是在不可能时，它"负载平衡"可用的物理域并将逻辑范围展形到它们上。

[0367] 因而，这种方法允许在一次分派中映射许多更多线程。毫无疑问，如果系统试图提供原子性的外观或分派的串行化，则，在将单次逻辑分派分成多次分派时需要小心维持此不变式。多次调用也具有它们的开销，尤其是在考虑它们被分派到其中的远程硬件时。

[0368] 另一方面，映射多次调用方法(#3)中的逻辑的逆线程ID捕捉在物理ID和逻辑ID之间的非常直接的关系，且因而将允许更多编译器优化。它也可以节省重构中所涉及的存根侧上的一些计算(尽管这将通常不太显著)。

[0369] 简而言之，使用哪种方法取决于系统的性能特性和底层编译器基础设施。正如所述的，在这一方法种，如果可能就优化直接映射，否则就以更为通用和非直接的索引映射代

码的额外成本来利用全线程域。由于每一种分派方案可以产生调用存根中的不同索引映射代码，且在运行时基于是否越界来拾取分派方案，所以可以在存根中生成索引映射代码的所有版本，且在运行时传递标志以便选择。

[0370] 由于在这一情况中通用一映射可以模糊关系在被用来访问数据结构的重构的逻辑线程ID和物理线程ID之间的关系，明智的是，复制内核函数调用以使得它在直接映射的情况中出现一次且在非直接映射情况中出现一次。这将得到较大的二进制，但是另一方面将在直接映射情况中允许更多向量化优化，直接映射情况通常更加常见。

[0371] 考虑它的另一方式是创建两个存根：一个用于直接映射情况，且另一个用于非直接情况，且在运行时选择正确的一个来调用。

[0372] 借助于这一方法，拾取thread_group_extents＝(Tz,Ty,Tx)＝(1,1,256)。可以基于场景和硬件特性调谐这一预定义的向量。该方法不取决于组件向量的数值(在它将用于可以选择的任何其他thread_group_extents值的意义上)。另外，可能生成假设这一值的不同值的多个存根，然后，在运行时为运行时选择准则选择特定的版本。

[0373] 对于其中N是1、2或3的情况，在每一维度中计算必要的组的数量，在直接映射中：

[0374] 1.如果在每一维度中要求少于MAX_GROUPS，则在物理组和逻辑组之间直接地映射，除非将得到过度的线程损耗量(见下)。

[0375] A.在每一维度中应用直接映射时，可能网格中的某一些维度将具有接近于或小于该维度中的线程组范围的值。例如，假定该网格指定X＝1但是X维度中的线程组范围是256。因此X维度中至少一个组被用来覆盖网格，但是每一组将仅在其中具有使得X＝0乘以逻辑网格内的有效点的那些值(例如，对于任何Vz或Vy，(Vz,Vy,13)将不被有效映射到逻辑网格)。这将得到实际上有助于网格的有用“覆盖”的线程中的仅1/256。为了防止这一情况，计算损耗量比率，损耗量比率被定义为我们将调度的线程的总数和网格实际上需要的线程的总数之间的比率。如果损耗量超出某个阀值(可由系统配置，例如它可以是5％)，则非直接映射比直接映射优选。

[0376] B.因而，如果损耗量太大，我们去往步骤5——网格的完全“负载平衡”的间接。

[0377] 2.否则，检查组的总数是否超过(MAX_GROUPS-1)^3。如果是，则分派是不可能的(当然，在这一不太可能的情况中可以借助多次调用)。

[0378] 3.否则，存在要求少于MAX_GROUPS个组的至少一个维度。存在所指定的维度是E1、E2和E3的假设，且不失一般性假设E1＜＝E2＜＝E3。无疑E3＞＝MAX_GROUPS(否则情况#1将适用)，且E1＜MAX_GROUPS(否则情况#2将适用)。

[0379] 4.现在，如果E2*E3＜MAX_GROUPS^2或如果E2＜MAX_GROUPS且E1*E3＜MAX_GROUPS^2，则，在这两个维度中必要的组跨两个维度而展形，且在逻辑域和物理域之间直接地映射第三个维度。

[0380] A.注意，在这一子句下，跨两个物理维度而展形两个逻辑维度的组，但是维持每一组内的结构。这得到下式(假定DimX是对应于E1的维度，且E2和E3对应于DimZ和DimY，且E2*E3＜MAX_GROUPS^2)：

[0381] i.logical_idx[DimX]＝gid[DimX]*Tx+gtid[DimX]

[0382] ii.logical_idx[DimZ..DimY]＝remap(gid[DimZ..DimY])*(Tz,Ty)+gtid[DimZ..DimY]

[0383] B.在组水平进行再映射，以便确保X维度上的逆映射易于计算。

[0384] 然而，这引入了增加的损耗量的可能性，如步骤(1.A)中所解释，因此，类似地，检查与这一映射一起产生的损耗量的水平，且如果它超过阀值，则接着是下面的选项#5。

[0385] 5.否则，如果上面没有应用直接映射或半直接映射条件，则采用通用映射是。

[0386] 下面阐述用于分派的示例伪代码和存根(作为示例而非限制)：

```
struct physical_dispatch_info
{
extents<3> physical_group_grid;
int dispatch_mode;
extents<3> virtual_group_grid;

};

struct physical_thread_info
{
index<3> local_id;
index<3> global_id;
index<3> group_id;
};

template<size_t N>
void determine_dispatch_dims(
```

[0387]

```
const extents<N>& g,
physical_dispatch_info& r
)
{
C_ASSERT(N>0);

//验证输入
for (int i=0; i<N; i++)
{
if (g[i] <= 0) throw NULL;
}

//验证容量
__int64 max_possible_threads =
(__int64(MaxGroupsInDim)-1) *
thread_group_extents[DimZ] *
(__int64(MaxGroupsInDim)-1) *
thread_group_extents[DimY] *
```

```
   (__int64(MaxGroupsInDim)-1) *
thread_group_extents[DimX] ;

   if (g.total_elements() > max_possible_threads) throw
NULL;

   r.dispatch_mode = 0; //最初未经确定

   if (N <= 3)
   {
   for(int i = 0; i < 3-N; i++)
   {
   r.virtual_group_grid[i] = 1;
   }

   for(int i = 3-N; i < 3; i++)
   {
   r.virtual_group_grid[i]=div_ceil(g[N-3+i],thread_g
roup_extents[i]);
   }

   if (max(r.virtual_group_grid) < MaxGroupsInDim)
[0388]  {
   double wastage_metric =
   (double(r.virtual_group_grid[0])*
   double(r.virtual_group_grid[1])*
   double(r.virtual_group_grid[2])*
   double(thread_group_extents.total_elements()))/
   double(g.total_elements());

   if (wastage_metric > TooWasteful)
   goto load_balance;

   r.physical_group_grid = r.virtual_group_grid;
   r.dispatch_mode = N;
   return;
   }
   struct physical_dispatch_info
   {
   extents<3> physical_group_grid;
   int dispatch_mode;
   extents<3> virtual_group_grid;

   };

   struct physical_thread_info
```

```
{
index<3> local_id;
index<3> global_id;
index<3> group_id;
};

template<size_t N>
void determine_dispatch_dims(
const extents<N>& g,
physical_dispatch_info& r
)
{
C_ASSERT(N>0);

//验证输入
for (int i=0; i<N; i++)
{
if (g[i] <= 0) throw NULL;
}

//验证容量
__int64 max_possible_threads =
```

[0389]

```
(__int64(MaxGroupsInDim)-1) *
thread_group_extents[DimZ] *
(__int64(MaxGroupsInDim)-1) *
thread_group_extents[DimY] *
(__int64(MaxGroupsInDim)-1) *
thread_group_extents[DimX] ;

if (g.total_elements() > max_possible_threads) throw
NULL;

r.dispatch_mode = 0; //最初未经确定

if (N <= 3)
{
for(int i = 0; i < 3-N; i++)
{
r.virtual_group_grid[i] = 1;
}

for(int i = 3-N; i < 3; i++)
{
r.virtual_group_grid[i]=div_ceil(g[N-3+i],thread_g
roup_extents[i]);
}
```

```
        if (max(r.virtual_group_grid) < MaxGroupsInDim)
        {
        double wastage_metric =
        (double(r.virtual_group_grid[0])*
        double(r.virtual_group_grid[1])*
        double(r.virtual_group_grid[2])*
        double(thread_group_extents.total_elements()))/
        double(g.total_elements());
```

[0390]

```
        if (wastage_metric > TooWasteful)
        goto load_balance;

        r.physical_group_grid = r.virtual_group_grid;
        r.dispatch_mode = N;
        return;
        }
```

[0391] 伪代码39

[0392] 索引代码生成:给定physical_thread_info(这只是gid/gtid/dtid信息的另一再现),基于在运行时传递的dispatcn_mode,生成不同的索引计算代码。

[0393]

```
template <int N>
bool calculate_logical_index(
const extents<N>& g,
const physical_dispatch_info& r,
const physical_thread_info& th,
index<N>& logical_idx
)
{
//
//注意，基于编译时常数N的值和分派模式，下面的代码包含一些失效代码
。
//编译器将相应地消除下面的代码。
//为完整起见，我们以这种冗余的形式给出这些代码。

int dispatch_mode = r.dispatch_mode;

//直接映射
if (dispatch_mode >=1 && dispatch_mode <= 3)
{
//断言(dispatch_mode == N);
```

[0394]

```
for (int i=0; i<N; i++)
{
logical_idx[i] = th.global_id[3-N+i];
}
return g.within_boundaries(logical_idx);
}

//部分直接映射
if (dispatch_mode >= 4 && dispatch_mode <= 6)
{
size_t yz_group_offset =
r.physical_group_grid[1]*th.group_id[DimZ] +
th.group_id[DimY];
extents<2>
yz_group_grid(r.virtual_group_grid[DimZ],
r.virtual_group_grid[DimY]);
index<2> yz_group_id =
raise<2>(yz_group_offset, yz_group_grid);
index<3> xyz_group_id =
index<3>(yz_group_id[DimZ], yz_group_id[DimY],
th.group_id[DimX]);

//重新排序xyz_group，使得我们可以
  //恢复我们放置在存根中的逻辑线程组的初始次序。
int dims[3]; // 为各维度设置的置换

switch (dispatch_mode) {
//逆映射
//情况4：dims[DimZ]=DimZ; dims[DimY]=DimY; dims[DimX]=DimX;
break;
case 4: dims[DimZ]=DimZ; dims[DimY]=DimY; dims[DimX]=DimX;
break;

//逆映射
//情况5：dims[DimZ]=DimZ; dims[DimY]=DimX; dims[DimX]=DimY;
break;
case 5: dims[DimZ]=DimZ; dims[DimY]=DimX; dims[DimX]=DimY;
break;

//逆映射
//情况6：dims[DimZ]=DimY; dims[DimY]=DimX; dims[DimX]=DimZ;
break;
case 6: dims[DimZ]=DimX; dims[DimY]=DimZ; dims[DimX]=DimY;
break;
}
```

```
xyz_group_id =
index<3>( xyz_group_id[dims[DimZ]],
xyz_group_id[dims[DimY]],
xyz_group_id[dims[DimX]]);
index<3> xyz_logical_idx;
xyz_logical_idx[0] = xyz_group_id[0] *
thread_group_extents[0] + th.local_id[0];
xyz_logical_idx[1] = xyz_group_id[1] *
thread_group_extents[1] + th.local_id[1];
xyz_logical_idx[2] = xyz_group_id[2] *
thread_group_extents[2] + th.local_id[2];

//取悦编译器，事实上N = 3，因此下面是空操作
logical_idx = xyz_logical_idx.extend<N>(0);
return g.within_boundaries(logical_idx);
}

//间接映射(dispatch_mode == 7)
extents<3> physical_grid = r.physical_group_grid *
thread_group_extents;
__int64 global_offset = flatten<3>(th.global_id,
physical_grid);
if (global_offset >= g.total_elements())
return false;
logical_idx = raise<N>(global_offset, g);
return true;
}
```

[0396] 伪代码40

[0397] 为进一步改善性能(避免存根中的if/else的成本),可以生成存根函数的多个版本,每一dispatch_mode一个版本,且在运行时拾取对应于由分派算法拾取的dispatch_mode的版本。也可能生成存根的一个通用版本以及几个更具体的版本(即两个选项的混合)。

[0398] 再次,如果想要适应逻辑计算域中的点的总数大于或等于(MAX_GROUPS-1)^3的情况,我们可以将多次内核调用用于该情况。

[0399] 方法5:将网格用作用于一般化计算域的中间嵌入域

[0400] 借助于这一方法,要求所有计算域类型定义这两个成员:

[0401] 1.extents<K>get_intermediate_grid()。这通过K的值同时地声明计算域类型将希望使用的中间嵌入网格的秩,对于线性嵌入来说,K的值可以是1,且它给运行时提供关于中间嵌入网格的范围信息。

[0402] 2.bool map_intermediate_to_logical(index<K>intermediate_idx,index<N>&idx)。这一函数将来自中间嵌入网格中的任何点的逆映射提供回给逻辑ID,且带有可选的过滤。即,如果没有向嵌入网格中的点返回保留地映射逻辑索引,则该函数返回假。

[0403] 对于给定的K,可以与在逻辑网格和物理网格之间映射的任一种先前的算法结合

使用嵌入网格的秩、秩K的相应的嵌入网格。相反，为进行分派，我们将首先询问布置其中间嵌入网格的类型，然后使用如在先前的算法中描述的所供应的网格来分派。然后，在存根处，将使用任一种先前的方法来重建中间嵌入网格索引，且将使用该类型所提供的函数map_intermediate_to_logical来从中间索引映射到逻辑索引。如果不过滤该点，则然后用给定的逻辑索引值来调用用户所提供的内核函数。

[0404] 方法6：调度线程分组的计算域

[0405] 这种方法涉及线程分组的计算域的物理调度。这是带有以下改变的方法#5的改编：

[0406] 1.组可以不是“部分的”，组中的所有线程都执行内核，或什么都不做。如先前所解释，这种类型的分散是的线程分组的目的受挫。

[0407] 2.所指定的中间计算网格的范围可被所指定的分组的线程的范围除尽。如果它们不是，则报告错误。

[0408] 3.由于上面的#2，用于边界条件和除法的舍入的一些测试变得不必要。

[0409] 在下面的代码中，术语“逻辑”是指中间网格，从调度程度的观点来看，中间网格是逻辑网格的替身。这只是通过重新使用较早地介绍的术语的愿望来提示的语法约定，但应理解，到此阶段的输入是中间网格。

[0410] 分派代码：

[0411]
```
template<size_t N>
void determine_blocked_dispatch_dims(
const extents<N>& g,
const extents<N>& thread_group_extents,
```

```
physical_dispatch_info& r
)
{
C_ASSERT(N>0);

//验证输入
for (int i=0; i<N; i++)
{
if (g[i] <= 0) throw NULL;
if (thread_group_extents[3-N+i] <= 0) throw NULL;
if ((g[i] % thread_group_extents[3-N+i]) != 0) throw
NULL;
```

[0412] }

```
//验证容量
__int64 max_possible_groups =
__int64(MaxGroupsInDim-1) *
__int64(MaxGroupsInDim-1) *
__int64(MaxGroupsInDim-1);
__int64 required_groups = g.total_elements() /
thread_group_extents.total_elements();

if (required_groups > max_possible_groups) throw NULL;

r.dispatch_mode = 0; //最初未经确定
```

```
if (N <= 3)
{
for(int i = 0; i < 3-N; i++)
{
r.virtual_group_grid[i] = 1;
}

for(int i = 3-N; i < 3; i++)
{
r.virtual_group_grid[i] = div_ceil(g[N-3+i],
thread_group_extents[i]);
}
if (max(r.virtual_group_grid) < MaxGroupsInDim)
{
double wastage_metric =
(double(r.virtual_group_grid[0])*
double(r.virtual_group_grid[1])*
double(r.virtual_group_grid[2])*
double(thread_group_extents.total_elements())) /
double(g.total_elements());

if (wastage_metric > TooWasteful)
```

[0413]

```
goto load_balance;

r.physical_group_grid = r.virtual_group_grid;
r.dispatch_mode = N;
return;
}

//在这里的情况试图将一个维度直接映射到Gx，并且
//将剩余的两个维度跨Gy和Gz而展形
if (N==3)
{
for (int disp=4; disp<=6; disp++)
{
int dims[3]; //为各维度设置的置换

//优选保留维度#2 ("x"
dimension),因此首先尝试情况'4'
switch (disp) {
case 4: dims[DimZ]=DimZ; dims[DimY]=DimY;
dims[DimX]=DimX; break;
case 5: dims[DimZ]=DimZ; dims[DimY]=DimX;
dims[DimX]=DimY; break;
```

```
case 6: dims[DimZ]=DimY; dims[DimY]=DimX;
dims[DimX]=DimZ; break;
}

__int64 _3rd_dim_groups =
r.virtual_group_grid[dims[DimX]];
if (_3rd_dim_groups < MaxGroupsInDim)
{
__int64 _2d_leftover =
r.virtual_group_grid[dims[DimZ]] *
r.virtual_group_grid[dims[DimY]];
__int64 _2d_extent =
__int64(ceil(pow(double(_2d_leftover),
double(0.5))));
if (_2d_extent < MaxGroupsInDim)
{
r.physical_group_grid[0] =
int(_2d_extent);
r.physical_group_grid[1] =
int(div_ceil(_2d_leftover, _2d_extent));
r.physical_group_grid[2] =
int(_3rd_dim_groups);
```

[0414]

```
r.dispatch_mode = disp;
return;
}
}
}
}
}

//在这里，N>3，否则需要某种展形
__int64 x = __int64(ceil(pow(double(required_groups),
double(1)/double(3))));
__int64 _2d_leftover = div_ceil(required_groups, x);
__int64 y = __int64(ceil(pow(double(_2d_leftover),
double(1)/double(2))));
__int64 z = div_ceil(_2d_leftover, y);

if (max(x,max(y,z)) >= MaxGroupsInDim)
throw NULL;

r.physical_group_grid[0] = int(z);
r.physical_group_grid[1] = int(y);
r.physical_group_grid[2] = int(x);
```

[0415]
```
        r.dispatch_mode = 7;
}
```

[0416] 伪代码41

[0417] 下面是存根代码。在下面的代码中，一些数据被示出为被当作参数而被传递，而实际上，那些参数将被分解为将允许更有效的代码生成的常数。尤其，THREAD_GROUP_EXTENTS和DISPATCH_MODE在大多数情况将被“焦化(SCORCHED)”为二进制代码。

[0418]
```
template <int N>
bool calculate_blocked_logical_index(
const extents<N>& g,
const extents<N>& thread_group_extents,
const physical_dispatch_info& r,
const physical_thread_info& th,
index<N>& logical_global_idx,
index<N>& logical_group_idx,
index<N>& logical_local_idx
)
```

```
{
//注意，基于编译时常数N的值和分派模式，下面的代码包含一些失效
代码。
//编译器将相应地消除下面的代码。
//为完整起见，我们以这种冗余的形式给出这些代码。
int dispatch_mode = r.dispatch_mode;

//直接映射
if (dispatch_mode >=1 && dispatch_mode <= 3)
{
//断言(dispatch_mode == N);
for (int i=0; i<N; i++)
{
logical_global_idx[i] = th.global_id[3-N+i];
logical_group_idx[i] = th.group_id[3-N+i];
logical_local_idx[i] = th.local_id[3-N+i];
}
//断言(g.within_boundaries(logical_idx));
return true;
}
```

[0419]

```
//部分直接映射(N=3)
if (dispatch_mode >= 4 && dispatch_mode <= 6)
{
size_t yz_group_offset =
r.physical_group_grid[1]*th.group_id[DimZ] +
th.group_id[DimY];
extents<2> yz_group_grid(r.virtual_group_grid[DimZ],
r.virtual_group_grid[DimY]);
index<2> yz_group_id = raise<2>(yz_group_offset,
yz_group_grid);
index<3> xyz_group_id = index<3>(yz_group_id[DimZ],
yz_group_id[DimY], th.group_id[DimX]);

//重新排序xyz_group，使得我们可以
//恢复我们放置在存根中的逻辑线程组的初始次序。
int dims[3]; //为各维度设置的置换

switch (dispatch_mode) {
//逆映射情况4: dims[DimZ]=DimZ;
dims[DimY]=DimY; dims[DimX]=DimX; break;
case 4: dims[DimZ]=DimZ; dims[DimY]=DimY;
dims[DimX]=DimX; break;
```

```
//逆映射情况5: dims[DimZ]=DimZ;
dims[DimY]=DimX; dims[DimX]=DimY; break;
case 5: dims[DimZ]=DimZ; dims[DimY]=DimX;
dims[DimX]=DimY; break;

//逆映射情况6:映射dims[DimZ]=DimY;
dims[DimY]=DimX; dims[DimX]=DimZ
case 6: dims[DimZ]=DimX; dims[DimY]=DimZ;
dims[DimX]=DimY; break;
}

xyz_group_id = index<3>( xyz_group_id[dims[DimZ]],
xyz_group_id[dims[DimY]], xyz_group_id[dims[DimX]]);

//过滤（对各组均匀）
if
(!r.virtual_group_grid.within_boundaries(xyz_group_
id))
return false;

//生成全局ID和局部ID
index<3> xyz_logical_idx;
```

[0420]

```
xyz_logical_idx[0] = xyz_group_id[0] *
thread_group_extents[0] + th.local_id[0];
xyz_logical_idx[1] = xyz_group_id[1] *
thread_group_extents[1] + th.local_id[1];
xyz_logical_idx[2] = xyz_group_id[2] *
thread_group_extents[2] + th.local_id[2];

//取悦编译器，事实上N = 3，因此下面是空操作
logical_global_idx = xyz_logical_idx.extend<N>(0);
logical_group_idx = xyz_group_id.extend<N>(0);
logical_local_idx = th.local_id.extend<N>(0);

return g.within_boundaries(logical_global_idx);
}

//间接映射(dispatch_mode == 7)
__int64 group_offset = flatten<3>(th.group_id,
r.physical_group_grid);
//对各组均匀过滤
if (group_offset >=
r.virtual_group_grid.total_elements())
return false;
logical_group_idx = raise<N>(group_offset,
```

[0421]
```
            r.virtual_group_grid);
            logical_global_idx = logical_group_idx *
            thread_group_extents + th.local_id;
            logical_local_idx = th.local_id;
            return true;
        }
```

[0422] 伪代码42

[0423] 示例过程

[0424] 图2-5是示出实现在此描述的技术的示例过程200、300、400和500的流程图。这些过程的讨论将包括参考图1的计算机组件。这些过程中的每一个被示出为逻辑流程图中的框的集合，其表示可以在硬件、软件、固件或其组合中实现的操作的序列。在软件的上下文中，框表示被存储在由这样的计算机的一个或多个处理器执行时执行所叙述的操作的一个或多个计算机可读存储介质上的计算机指令。注意，所描述的过程的次序不旨在被解释成限制，且可以以任何次序组合任何数量的所描述的过程框以实现过程或替换过程。另外，可以在不偏离在此描述的本主题的精神和范围的前提下从过程删除单个框。

[0425] 图2示出促进产生能够在支持DP的硬件上执行的程序的示例过程200。该产生可以例如通过C++编程语言编译器来执行。过程200至少部分地由包括例如图1的计算设备102的计算设备或系统来执行。计算设备或系统被配置为促进一个或多个DP可执行程序的产生。计算设备或系统可以是不支持DP的或支持DP的。如此配置的计算设备或系统取得特定的机器或装置的资格。

[0426] 如在这里所示出，过程200以操作202开始，其中计算设备获得程序的源代码。此源代码是以一些人类可读的计算机编程语言(例如，C++)编写的文本的语句或声明的集合。可以从被存储在诸如存储系统106等的辅助存储系统中的一个或多个文件获得源代码。

[0427] 对于此示例过程200，所获得的源代码包括对DP调用点的调用的文本表示。文本表示包括与对DP调用点的调用相关联的自变量的指示符。来自上面的伪代码列表8-11的函数调用是在这里所期望的文本表示的类型的示例。尤其，那些列表的forall、scan、reduce和sort函数调用以及它们的自变量是示例文本表示。当然，也期望函数调用和自变量的文本表示的其他格式。

[0428] 在操作204，计算设备预处理源代码。在被编译时，预处理可以包括源代码的词汇分析和语法分析。在编译器的编程语言的上下文内，预处理检验各种字、数字和符号的含义以及它们与编程规则或结构的一致性。而且，源代码可以被转换为中间格式，其中文本的内容以对象或令牌方式来表示。此中间格式可以将内容重新排列为树结构。对于此示例过程200，代替使用对于DP调用点函数(带有其自变量)的调用的文本表示，可以以中间格式来表示DP调用点函数调用(带有其自变量)。

[0429] 在操作206，计算设备处理源代码。在被编译时，源代码处理将源代码(或源代码的中间格式)转换为可执行指令。

[0430] 在操作208，当计算设备处理源代码(以其本机格式或中间格式)时，计算设备解释函数调用(带有其自变量)的每一个表示。

[0431] 在操作210，计算设备确定函数调用的经解析的表示是否需要DP计算。如果函数调用的经解析的需要DP计算，示例过程200移动到操作212。否则示例过程200移动到操作214。

在操作212或214处生成适当的可执行指令之后，示例过程返回到操作208，直到所有的源代码已经被处理。

[0432] 在操作212，计算设备在支持DP的硬件(例如，DP计算引擎120)上生成用于DP计算的可执行指令。所生成的DP可执行指令包括基于对带有其所关联的自变量的DP调用点函数的调用的那些指令。创建那些DP调用点函数指令，以便在具体目标支持DP的硬件(例如，DP计算引擎120)上执行。另外，在那些DP函数指令被执行时，基于自变量来定义数据集，且该数据集被存储在是支持DP的硬件的一部分的存储器(例如，节点存储器138)中。此外，在那些DP函数指令被执行时，一旦该数据集被存储在支持DP的存储器中，就执行DP调用点函数。

[0433] 在操作214，计算设备在不支持DP的硬件(例如，非DP主机110)上生成用于非DP计算的可执行指令。

[0434] 在处理之后，或作为处理的一部分，计算设备链接所生成的代码并将它与其他已编译的模块和/或运行时程序库组合以产生最终的可执行文件或镜像。

[0435] 图3示出促进在支持DP的硬件中执行DP可执行程序的示例过程300。过程300至少部分地由包括例如图1的计算设备102的计算设备或系统来执行。计算设备或系统被配置为在不支持DP的硬件(例如，非DP主机110)和支持DP的硬件(例如，DP计算引擎120)两者上执行指令。实际上，用执行操作和/或是操作的对象的适当的硬件(例如，非DP主机110和/或计算引擎120)来示出操作。如此配置的计算设备或系统取得特定的机器或装置的资格。

[0436] 如在这里所示出，过程300以操作302开始，其中计算设备选择将被用于DP计算的数据集。更具体地，计算设备的不支持DP的硬件(例如，非DP主机110)选择被存储在存储器(例如，主存储器114)中的数据集，所述存储器(例如，主存储器114)不是计算设备(例如，计算设备102)中的一个或多个的支持DP的硬件的一部分。

[0437] 在操作304，计算设备将所选择的数据集的数据从非DP存储器(例如，主存储器114)传递到DP存储器(例如，节点存储器128)。在一些实施方式中，主机110和DP计算引擎120可以共享普通存储器系统。在那些实施方式中，将对数据的权限或控制从主机传递到计算机引擎。或者，计算引擎获得存储器中的数据的共享控制。对于这样的实施方式，在此所传递的数据的讨论意指，DP计算引擎拥有对数据的控制而不是数据已经被从一个存储器移动到另一个存储器。

[0438] 在操作306，计算设备的支持DP的硬件将数据集的所传递的数据定义为字段。当数据集被存储在支持DP的存储器(例如，节点存储器138)上时，字段定义数据集的逻辑排列。DP调用点函数调用的自变量定义字段的参数。那些参数可以包括数据集的秩(即，维数)和数据集中的每一个元素的数据类型。索引和计算域是影响字段的定义的其他参数。这些参数可以帮助定义字段的处理的形状。在存在准确的类型匹配时，则它只是普通的自变量传递，但可以存在投影或局部投影。

[0439] 在操作308，计算设备的支持DP的硬件准备将由多个数据并行线程执行的DP内核。DP内核是在数据集的一部分上执行的基本迭代DP活动。DP内核的每一个实例是相同的DP任务。特定的DP任务可以由程序员在编程DP内核时指定。多个处理元素(例如，元素140-146)表示每一个DP内核实例。

[0440] 在操作310，作为计算设备的支持DP的硬件的一部分运行的DP内核的每一个实例将来自字段的数据的一部分接收为输入。DP内核的每一个实例在数据集的不同部分(如由

字段所定义)上操作,这是数据并行性的本质。因此,每一个实例将其自己的数据集的一部分接收为输入。

[0441] 在操作312,计算设备的支持DP的硬件并行调用支持DP的硬件中的DP内核的多个实例。对于由先前的操作适当地设置的所有动作,在操作312执行实际的数据并行计算。

[0442] 在操作314,计算设备的支持DP的硬件获得从DP内核的所调用的多个实例得到的输出,所得到的输出被存储在支持DP的存储器。至少初始地,将来自DP内核实例的执行的输出收集和存储在本地的支持DP的存储器(例如,节点存储器128)。

[0443] 在操作316,计算设备将所得到的输出从支持DP的存储器传递到不支持DP的存储器。当然,如果存储器被主机和计算引擎共享,则仅需要传递控制或授权而不是数据本身。一旦收集和存储来自DP内核实例的所有输出,就从DP计算引擎120将所收集的输出移动回到非DP主机110。

[0444] 操作318表示执行一个或多个非DP计算并同时地与DP内核的多个实例的并行调用(操作312)这样做的计算设备的不支持DP的硬件。也可以同时地与诸如操作306、308、310和314的DP计算等的其他DP计算一起执行这些非DP计算。此外,可以同时地与诸如操作304和316的数据传输等的在非DP存储器和DP存储器之间的其他数据传输一起执行这些非DP计算。

[0445] 被示出为操作316的一部分的输出的返回传输与调用程序异步。也就是说,发起DP调用点函数的程序(例如,程序118)不需要等待DP调用点的结果。相反,程序可以继续执行其他非DP活动。输出的实际的返回传输是同步点。

[0446] 在操作320,计算设备继续正常执行一个或多个非DP计算。

[0447] 图4示出促进至少一个DP函数的DP调用存根的产生的示例过程400。可以例如通过诸如编译器116等的C++编程语言编译器来执行存根产生。过程400至少部分地由包括例如图1的计算设备102的计算设备或系统来执行。计算设备或系统被配置为促进一个或多个DP调用存根的产生。计算设备或系统可以是不支持DP的或支持DP的。如此配置的计算设备或系统取得特定的机器或装置的资格。

[0448] 对于过程400的此描述,出于示出的目的,引用以下示例源代码伪代码:

[0449]

```
__declspec(vector) void foo(index<2> cIdx, float & c,
read_only<field<2, float>> mA, read_only<field<2, float>>
mB, int n);

grid<2> gA(rv, M, W), gB(rv, W, N), gC(rv, M, N);
field<2, float> mA(gA), mB(gB), mC(gC);
…
 forall(mC.get_grid(), foo, mC, mA, mB, 5);
```

[0450] 伪代码43

[0451] 如在这里示出,过程400以操作402开始,其中计算设备标识当前并行活动在计算域中的位置。这基于目标相关计算单元身份(例如,GPU六维线程部署模型中的线程索引)来进行。换言之,计算设备标识目标支持DP的硬件(例如,计算引擎120)中的物理位置,其中对于所指派的计算域进行数据并行活动。

[0452] 而且，在操作402，计算设备选择线程部署策略，并在调用存根中基于目标相关计算单元身份(这在运行时是可用的，对编译器来说不可访问)生成标识(例如，映射)当前并行活动在计算域中的位置的相应索引映射代码。在这里，计算设备也生成入口函数头部的声明，这是因为在头部中所声明的内容取决于线程部署决定。

[0453] 此操作可以包括检查诸如forall等的DP函数的自变量；生成入口函数头部的声明和调用存根中的目标相关资源的声明；以及将DP硬件线程位置映射到逻辑并行活动在计算域中的位置。

[0454] 为了进一步详细描述，作为操作402的一部分，计算设备检查诸如forall等的DP函数的自变量。计算设备确定所需要的目标缓冲器的数量和类型，且然后在调用存根中声明它们。基于上面的伪代码14中给出的示例，在forall调用中存在三个字段类型自变量，且因而需要三个设备缓冲器绑定。

[0455] 每一字段实例对应于目标缓冲器绑定变量。然而，字段可以被包含在另一数据结构中。因此，执行每一个DP函数自变量的深遍历(deep traversal)以便标识所有的字段实例。计算设备及物地(transitively)确定由自变量列表包含的每一字段实例的可写性，以便定义用于调用存根的中间代码中的目标缓冲器绑定的具体类型。一旦收集所有信息，为存根声明缓冲器绑定的列表。

[0456] 也确定要使用的缓冲器绑定变量的种类。如果存根是以例如HLSL，则HLSL具有许多种类的缓冲器绑定类型(例如，ByteAddrressBuffer、StructuredBuffer等等)。StructuredBuffer<T>可以被用来表示field<N,T>的设备缓冲器绑定。

[0457] 在一些实现中，类似于在HLSL中的实现，将缓冲器绑定类型分成两个类别：只读或可写。StructuredBuffer<T>是只读，且RWStructuredBuffer<T>是可写。至少一个实现使用内核参数类型的“constness”来确定给定的函数的自变量是否是可写的。另一实现使用专用类型read_only<T>来断言只读用法，以使得编译器可以将其映射到只读缓冲器绑定。

[0458] 基于来自上面的伪代码19的示例，这是至今所创建的调用存根的一部分：

[0459]
```
////////////////////////////////////////////////////////
RWStructuredBuffer<float> gB1 : register(u0);
StructuredBuffer<float> gB2   : register(t0);
StructuredBuffer<float> gB3   : register(t1);

////////////////////////////////////////////////////////
```

[0460] 伪代码43

[0461] 而且，声明常量缓冲器。常量缓冲器被用来将所有必要的数据从非DP主机110传递到不是字段的数据集的一部分的DP计算引擎120。换言之，经由常量缓冲器将DP函数的自变量列表中的所有其他非字段数据传递给目标。常量缓冲器可以包含(作为示例而非限制)：

[0462] ●字段自变量的形状信息(例如，范围、乘法器、偏移)；

[0463] ●包括计算域参数在内的任何非字段函数自变量(例如，标量自变量)；以及

[0464] ●着色器分派形状信息(例如，组维度乘法器)。

[0465] 基于来自上面的伪代码19的示例，常量缓冲器声明可以类似于此：

[0466]

```
//////////////////////////////////////////////////////////////
cbuffer CB_PARAMS {
  __group_dims_mult cb_group_dims_mult;
  __grid_2 cb_compute_grid;
  __field_2_base cb_arg1;
  __field_2_base cb_arg2;
  __field_2_base cb_arg3;
  int cb_arg4;
};
//////////////////////////////////////////////////////////////
```

[0467] 伪代码44

[0468] 在这里，_grid_2是grid<2>的HLSL类，且_field_2_base是对应于field<2,T>的共享信息的HLSL类。注意，特殊索引参数不能被传递且不经由常量缓冲器而被传递。相反，在调用存根中生成它们。

[0469] 另一实现可以包括生成原始的非类型化数据，且使用适当的偏移来从最初类型化的数据访问数据。在该情况中，常量缓冲器类似于：

```
cbuffer CB_PARAMS {
uint m1;
```

[0470]

```
unit m2;
...
}
```

[0471] 在这一情况中，将不生成HLSL类。

[0472] 操作402也可以包括入口函数头部的声明。这实际上是调用存根的声明。这涉及声明“numthread”属性和声明系统变量，对调用存根来说可能是有用的。

[0473] “numthread”属性指定框内的线程的形状。numthread指示如何以多个维度部署在线程组中的线程。对于至少一个实现，对于非tiled情况，计算设备生成作为普通的线程部署的“numthreads[256,1,1]”。对于tiled情况，如果tile的秩小于或等于三，则如果可能的话计算设备使用“numthreads”属性中的tile的形状。否则，计算设备尽可能高效地将tile的维度分布到3维物理线程组，或如果tile不能适合物理线程组，则报告错误。

[0474] 操作402也可以包括将DP硬件的线程位置映射到逻辑并行活动在计算域中的位置。

[0475] 计算域的概念有助于描述数据并行函数的并行性。这允许程序员在部署并行活动时以逻辑级而不是计算设备级(即，硬件级)思考。例如，对于向量加法，计算域是向量的形状，也就是说，每一元素可以通过并行活动独立地计算。然而，在使用在此描述的一种或多种实现时，这些逻辑并行活动应如何被映射到底层DP硬件不需要是程序员的关心的部分。

[0476] 然而，每一种类的DP硬件具有其自己的部署并行计算的方式。在一些DP硬件中，以六维模型分派GPU线程。对于那些GPU线程，使用三个维度来描述线程框(或组)的部署，并使用三个维度来描述在框(组)内的线程的形状。从过程400得到的调用存根可以将DP硬件的线程位置(例如，六维域中的点)映射到逻辑并行活动在计算域中的位置。

[0477] 完成此映射的一个方式包括在相同的分派中的所有线程当中获得线性的线程标识(“ltid”)(这在此被称为“变换索引展平”或简单地称为“展平”)且然后，将线性线程标识

映射回到计算域中的索引(这在此被称为“索引提高”或简单地称为“提高”)。对于tiled部署,使用相似的展平和提高算法可以将三维线程组域和三维组内域分别独立地映射到tile索引和本地索引。

[0478] 例如,存在带有范围E0、E1、E2(最显著的到最不显著的)的三维域。对于在此域中的给定的点(I0、I1、I2),

[0479] ltid=I0*E1*E2+I1*E2+I2

[0480] 注意,并行活动的总数小于由“numthreads”预定义的组内的线程维数(例如,对于非tiled情况是256)是可能的。在该情况中,将存在不应执行内核的一些线程,这是由于它们不是计算域的一部分。为防止这些线程进行不被支持进行的工作,在计算线性线程标识的代码之后,编译器将调用存根的剩余部分封装在条件中,以确保映射到计算域的线程将执行内核,但其他线程不执行内核。

[0481] 下一个步骤是将ltid提高到计算域的索引。以下的函数在grid<2>计算域中进行提高:

[0482]
```
__index_2 map_index(__grid_2 g, uint flat_index)
{
__index_2 index;
uint extent = g.get_size();
extent /= g.m_extents[0];
index.m_values[0] = flat_index / extent;
flat_index = flat_index % extent;
index.m_values[1] = flat_index;
return index;
}
```

[0483] 伪代码45

[0484] 例如,对于非平铺情况,进行展平和提高调用存根的代码可以类似于此:

[0485]
```
////////////////////////////////////////////////////////////
unit ltid = (Gid.z * cb_group_dims_mult.x_mult_y +
             Gid.y * cb_group_dims_mult.x +
             Gid.x) * 256
          + GTid.x;
__index_2 compute_idx = map_index(cb_compute_grid, ltid);
////////////////////////////////////////////////////////////
```

[0486] 伪代码46

[0487] 注意,此展平和然后提高方法是不取决于线程组形状或计算域形状的形式的最通用的算法。但是涉及此两步映射的计算可以是一些应用程序所关心的。一种优化可以是如果可能的话使线程组形状匹配计算域形状。

[0488] 此展平-提高方法处理在目标线程模型和计算域之间的所有种类的映射。为改善性能,对于某些专用的情况,可以部署较好的执行计划以使目标线程模型与计算域部分地或完全地匹配,使得目标线程索引可以直接地被用作计算域的索引。对于这些情况,编译器将稍微不同地生成代码以利用这样的专门化。

[0489] 在操作404,计算设备用目标相关资源设置DP函数的字段对象。

[0490] 在操作406，计算设备准备内核期望的适当的参数。此操作在图5的稍后讨论中详细说明。

[0491] 在操作408，计算设备产生将调用具有所准备的参数的内核的代码。

[0492] 在操作410，计算设备将结果存储回到正确的位置。在投影被执行时，这一操作将结果恢复回到其中发生到标量的投影的缓冲器。对于发生到标量的投影的那些可写的参数，计算结果被写回到原始的缓冲器的适当的存储槽。

[0493] 在操作412，计算设备输出诸如存根118的那些等的调用存根。调用存根可以被存储到例如存储器114和/或存储子系统106。

[0494] 此外，计算设备可以生成附加代码以设置适当的环境以开始DP计算。例如，HLSL中的调用存根可以包括以下(作为示例而非限制)：

[0495] • 定义将要使用的所有种类的缓冲器，例如，结构化缓冲器(可写或不可写)、纹理、常量缓冲器等等；

[0496] • 声明适当的入口点函数头部，入口点函数头部包括numthreads属性和SV变量(SV变量是DP计算设备提供的专用的物理线程索引，包括SV_GroupIndex、SV_DispathThreadId、SV_GroupThreadID、SV_GroupID)。

[0497] 图5示出准备内核期望的适当的参数的示例过程500。过程500至少部分地通过包括例如图1的计算设备102的计算设备或系统来执行。计算设备或系统被配置为在不支持DP的硬件(例如，非DP主机110)和支持DP的硬件(例如，DP计算引擎120)两者上执行指令。实际上，借助于执行操作和/或是操作的对象的适当的硬件(例如，非DP主机110和/或计算引擎120)来示出各操作。如此配置的计算设备或系统取得特定的机器或装置的资格。

[0498] 过程500是来自图4的操作406的扩展。如在这里所示出，过程500在操作404之后拾取。在操作502，计算设备确定内核函数的参数是否对应于特殊索引，即，它标识计算域中的并行活动。如果是，则过程继续到操作504。如果不是，则过程进行到操作506。

[0499] 在操作504，在专用的令牌(例如，(_index，_tile_index，_local_index))被用作DP函数自变量的一部分时，计算设备创建适当的索引实例。如果使用了专用的令牌，则程序员指定对非元素的内核的映射。在这之后，过程500完成，且过程400在操作408拾取。另外，此过程可以对内核函数参数一个一个地迭代。

[0500] 如果没有使用特殊索引令牌，则过程500沿着经由操作506的路径向下进行。作为这一操作的一部分，计算设备确定形式参数是否匹配实际参数。如果是，则过程继续到操作508。如果不是，则该过程进行到操作510。

[0501] 在操作508，计算设备将DP函数的自变量直接地广播给内核。如果参数是广播标量($R_a=R_f=0$)，则来自常量缓冲器的值被内核利用。如果参数是广播字段($R_a=R_f>0$)，则适当的字段类型的实例被创建并用经由常量缓冲器和被定义为过程400的一部分的缓冲器所传送的字段的形状信息来初始化。在操作508之后，过程500是完整的，且过程400在操作408加速。

[0502] 如果形式参数不匹配实际参数，则该过程沿着经由操作510的路径向下进行。这意味着该参数涉及投影($R_a=R_f+R_c$)。作为这一操作的一部分，使用作为过程400的一部分计算的计算索引，秩R_a处的字段被投影到秩R_f处的字段。如果R_f为零，则将元素从投影的字段加载到标量本地的变量。在操作508之后，过程500完成，且过程400在操作408拾取。

[0503] 替代过程500，可以允许内核函数访问计算域中的并行活动的身份。代替用特殊令牌，可以隐含地允许它们的功能。以这一方式，例如，如果内核函数具有一个比由forall调用提供的更多参数，且第一参数是在具有与计算秩相同的秩的类型index_group，则编译器可以准备特殊索引并将其作为其第一参数而传递给内核函数。在这里，做出以下判决：是否应从特殊索引或来自forall调用的用户提供的数据映射给定的内核函数参数，且对于后面的情况，是否应使用投影或广播。

[0504] 实现细节

[0505] 在一种或多种实现中，编译器116独立地从调用存根的代码生成执行内核的生成。调用存根不是内核的定制。相反，编译器116将内核看作是调用函数且仅要求内核的签名。这允许更灵活的编译模型和内核的所生成的代码的重用，但是如果内核体是可用的，则不排除优化机会。

[0506] 尤其，将在此描述的本发明概念实现到C++编程语言，可以涉及使用模板语法来表达大多数概念和避免核语言的扩展。模板语法可以包括可变参数模板，它是采取可变数量的自变量的模板。模板是允许函数和类与泛型类型一起操作的C++编程语言的特征。也就是说，函数或类无需必须针对每一数据类型重写就可以作用于许多不同的数据类型。泛型类型允许将数据提高到类型系统中。这允许定制在编译时通过标准兼容的C++编译器来检查的域专用的语义。适当的编译器(例如，编译器116)可以具有准确的错误消息并强加某种类型的限制。

[0507] 类index的实例表示任何N维索引或N维范围。N是编译时常量，它是索引模板类的参数。或者，索引可以从范围分开。而且，作为替换方式的一部分，为避免绑定到模板，可以在编译时将所有秩指定为该类型的静态的常量成员。在二维整数空间中，例如，此空间中的点通常被标记为“<x,y>坐标”。对于在此描述的一种或多种实现，这样的坐标可以用二维空间来表示为类index<2>的实例。对于类型index<2>的给定的实例idx，使用idx[0]和idx[1]来获得“x”和“y”。程序员拥有是否使得“x”与idx[0]相等且“y”与idx[1]相等的选择，或用相反的方式。这一决定时常由用于索引对象且用来索引到其中的数据的物理布局来指示。

[0508] 索引可以从数组初始化或直接地通过将索引值传送到构造器来初始化。这是可以达成的伪代码的示例：

[0509] size_t extents[3]={3,7,0};

[0510] index<3>idx1(extents);

[0511] index<3>idx2(3,7,0);

[0512] 伪代码47

[0513] 上面的伪代码将初始化三维索引，其中idx1[0]等于3，idx1[1]等于7，且idx1[2]等于0。换言之，将索引的元素列出为从较低的索引维度到较高的索引维度。

[0514] 索引支持许多有用的操作。可以复制和比较它们。乘法、加法和减法操作符可以以逐点的方式被应用到两个索引对象(具有相同的维度)。可以计算两个索引的点乘(像向量那样处理它们)。索引对象是轻量级的，且通常传值来传递，且被分配在堆栈上。

[0515] DP调用点函数调用的自变量被用来定义DP调用点函数将在其上操作的字段的参数。换言之，自变量帮助定义字段所定义的数据集的逻辑排列。

[0516] 除了关于解释字段的自变量的规则除外，还有在一种或多种实现中可以被应用到

DP调用点函数的其他规则:将相同的标量值传递给调用,且避免定义评估次序。

[0517] 如果实际参数是标量值,则相应形式参数可以被限制为具有非引用类型或被限制为具有“const”属性。对于这一限制,标量被同等地传递给所有内核调用。这是基于在调用点从主机环境复制的标量参数化计算节点的机制。

[0518] 在DP内核调用内,字段可以被限制为与至多一个非常量引用或聚集形式相关联。在该情况中,如果字段与非常量引用或聚集形式相关联,则不可以以不同于非常量引用或聚集形式的任何方式来引用字段。这一限制避免了必须定义评估次序。它也防止危险的混叠且可以作为冒险检测的副作用而被强加。此外,通过将指派的目标同等地看作是元素指派函数的实际的、非常量的、非元素参数,这一限制强加写前读取(read-before-write)语义。

[0519] 对于至少一种实现,可以使用“_declspec”关键字来将内核定义为对C++编程语言的扩展,其中给定的类型的实例应被存储带有域专用的存储类属性。更具体地,“_declspec(vector)”被用来定义对C++语言的内核扩展。

[0520] “map_index”在范围为0至网格的大小减一的偏移和N维度索引之间映射。在偏移和N维度索引之间的映射假设idx[0]是最重要的且最后的索引维度是最不重要的。

[0521] 在将N维数据存储在一维存储器中时,改变最不重要的索引维度得到对邻近的存储器位置的引用,同时改变最重要的索引维度得到远离的存储器引用。下表示出如何将二维网格放置在一维存储器中。元组是<idx[0],idx[1]>的形式。

[0522] 这是3乘4的二维网格的逻辑视图:

[0523]

<0,0>	<0,1>	<0,2>	<0,3>
<1,0>	<1,1>	<1,2>	<1,3>
<2,0>	<2,1>	<2,2>	<2,3>

[0524] 且这是网格将如何被转换成平面索引地址:

[0525]

平面索引	0	1	2	3	4	5	6	7	8	9	10	11	12
网格索引	<0,0>	<0,1>	<0,2>	<0,3>	<1,0>	<1,1>	<1,2>	<1,3>	<1,4>	<2,0>	<2,1>	<2,2>	<2,3>

[0526] 用法示例:

[0527] grid<2>cubic_domain(rv,3,4);

[0528] index<2>idx=cubic_domain.map_index(7);//idx=

[0529] <1,3>

[0530] 可以从用户提供的输入数据(forall调用的自变量)直接地映射内核函数所期望的参数中的某一些(经由投影或广播),一些参数(例如索引、tile索引、局部索引)取决于线程分派决定,线程分派决定不具有在用户提供的输入数据中的相应副本。在此描述的实现给用户提供在内核函数中引用这样的专用的项的途径。

[0531] 特殊索引令牌是用于该目的的一种方法。它用作forall自变量列表中的标记(marker),forall自变量列表让编译器知道,内核函数中的相应的参数应被映射到从线程

分派决定而不是任何用户提供的输入数据获得的特殊索引。或者，代替特殊索引令牌，在程序员想要引用在他们的内核函数中的这样的特殊索引的情况中，允许用户在内核参数列表中具有比在forall调用点所提供的多一个的参数，且第一参数可以是他们期望的特殊索引的类型。这是令编译器知道内核函数的第一参数应被映射到特殊索引的另一途径。

[0532] 结论

[0533] 如在本申请中所使用的，术语“组件”、“模块”、“系统”、“接口”等一般旨在表示计算机相关的实体，其可以是硬件、硬件和软件的组合、软件、或者执行中的软件。例如，组件可以是，但不限于是，在处理器上运行的进程、处理器、对象、可执行码、执行的线程、程序和/或计算机。作为示例，运行在控制器上的应用程序和控制器都可以是组件。一个或多个组件可以驻留在进程和/或执行线程中，并且组件可以位于一个计算机内和/或分布在两个或更多的计算机之间。

[0534] 此外，所要求保护的主题可以使用产生控制计算机以实现所公开的主题的软件、固件、硬件或其任意组合的标准编程和/或工程技术而被实现为方法、装置或制品。

[0535] 所要求保护的主题的实现可以存储在某种形式的计算机可读介质上或通过某种形式的计算机可读介质传输。计算机可读介质可以是可由计算机访问的任何可用介质。作为示例，计算机可读介质可包括但不限于“计算机可读存储介质”和“通信介质”。

[0536] “计算机可读存储介质”包括以用于存储诸如计算机可读指令、计算机可执行指令、数据结构、程序模块或其他数据等信息的任何方法或技术实现的易失性和非易失性、可移动和不可移动介质。计算机可读存储介质包括但不限于，RAM、ROM、EEPROM、闪存或其他存储器技术、CD-ROM、数字多功能盘(DVD)或其他光盘存储、盒式磁带、磁带、磁盘存储或其他磁存储设备，或者任何其他可用于存储所需信息并可由计算机访问的介质。

[0537] “通信介质”通常用诸如载波或其他传输机制等已调制数据信号来体现计算机可读指令、计算机可执行指令、数据结构、程序模块或其他数据。通信介质也包括任意的信息传递介质。

[0538] 如本申请中所使用的，术语“或”意指包括性“或”而非互斥性“或”。即，除非另有指定或从上下文可以清楚，否则“X使用A或B”意指任何自然的包括性排列。即，如果X使用A，X使用B，或X使用A和B两者，则在任何以上情况下，都满足“X使用A或B”。另外，本申请中和所附权利要求书中所使用的冠词“一”和“一个”一般应被解释为是指“一个或多个”，除非另有指定或从上下文可以清楚指的是单数形式。

[0539] 尽管已经用结构特征和/或方法动作专用的语言描述了本主题，但要理解，所附权利要求书中定义的主题不必限于所描述的具体特征或动作。相反，这些具体特征和动作是作为实现权利要求的示例形式来公开的。

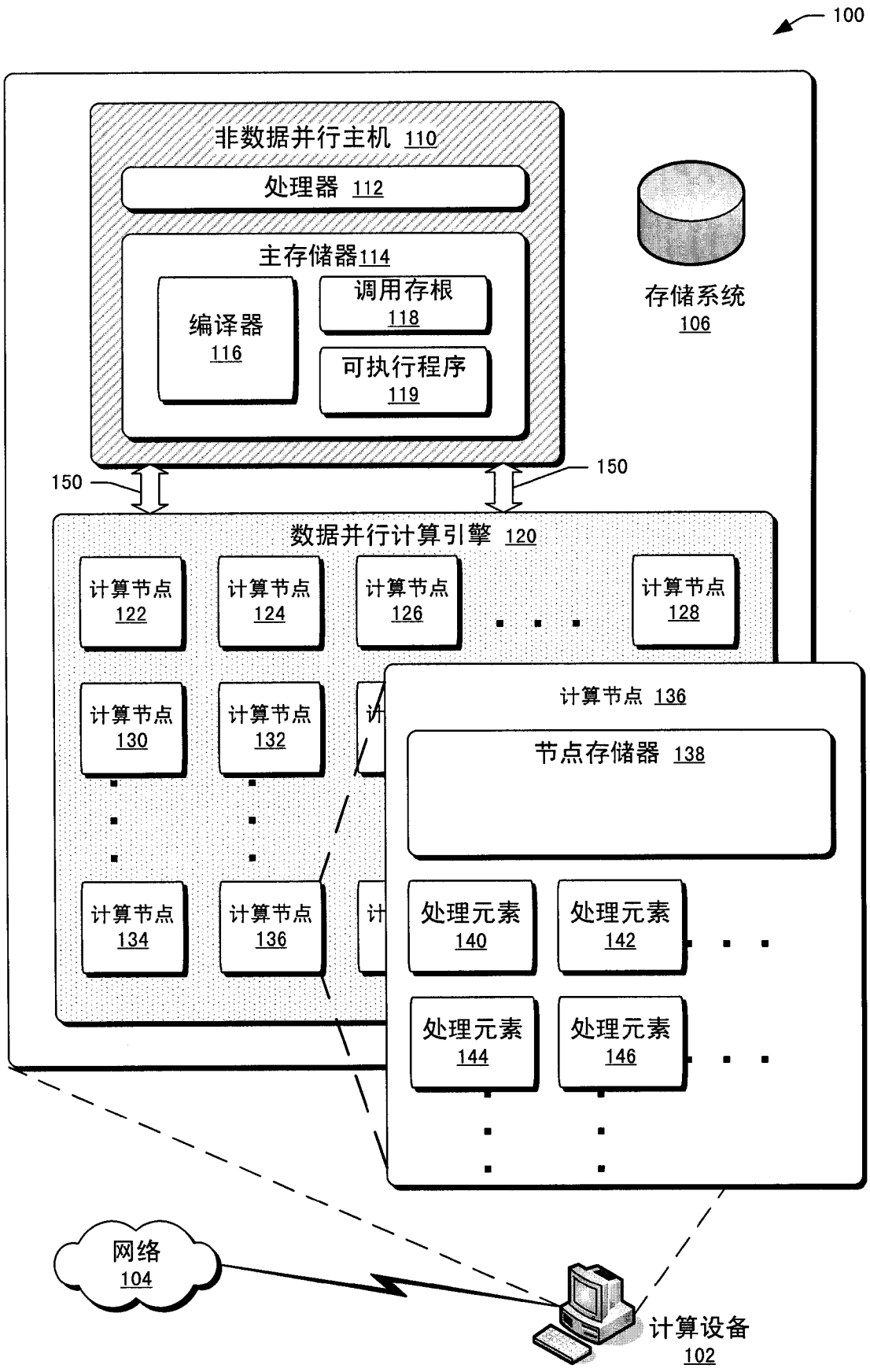

100
非数据并行主机 110
处理器 112
主存储器114
编译器 116
调用存根 118
可执行程序 119
存储系统 106
150
150
数据并行计算引擎 120
计算节点 122
计算节点 124
计算节点 126
计算节点 128
计算节点 130
计算节点 132
计算节点 134
计算节点 136
计算节点 136
节点存储器 138
处理元素 140
处理元素 142
处理元素 144
处理元素 146
网络 104
计算设备 102

图1

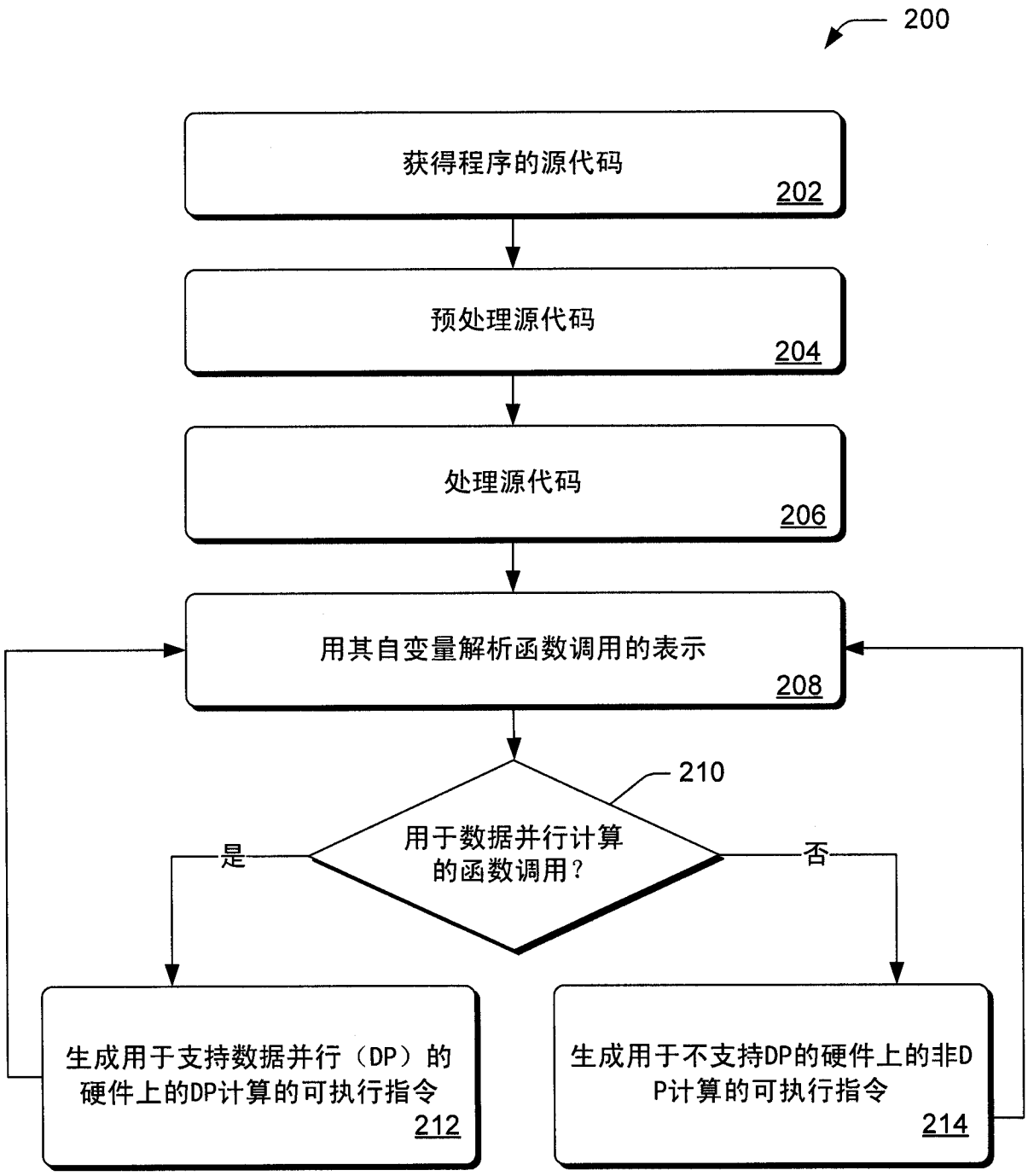

200
获得程序的源代码
202
预处理源代码
204
处理源代码
206
用其自变量解析函数调用的表示
208
210
用于数据并行计算的函数调用？
是
否
生成用于支持数据并行（DP）的硬件上的DP计算的可执行指令
212
生成用于不支持DP的硬件上的非DP计算的可执行指令
214

图2

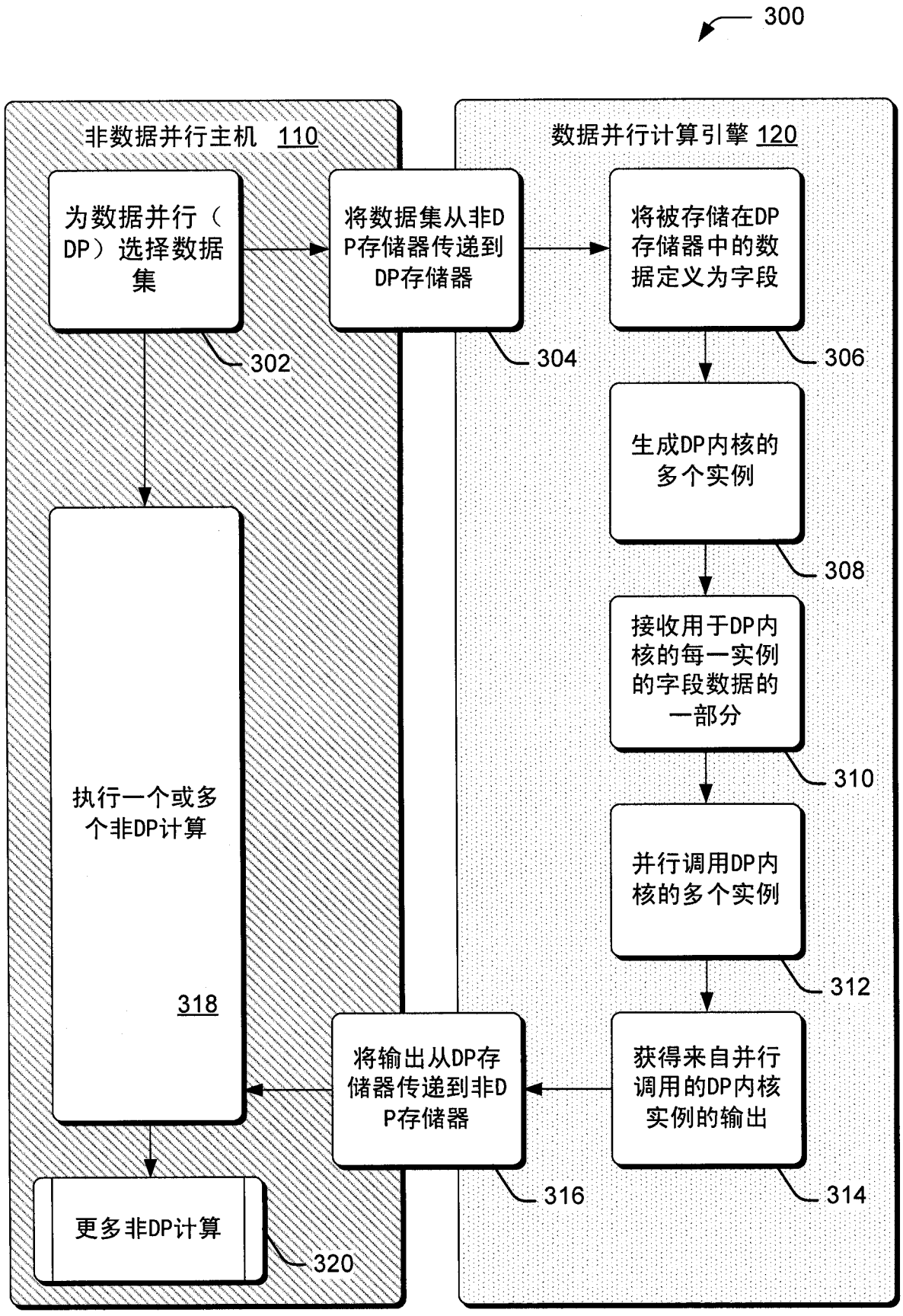

300
非数据并行主机 110
数据并行计算引擎 120
为数据并行（DP）选择数据集
302
将数据集从非DP存储器传递到DP存储器
304
将被存储在DP存储器中的数据定义为字段
306
生成DP内核的多个实例
308
接收用于DP内核的每一实例的字段数据的一部分
310
并行调用DP内核的多个实例
312
获得来自并行调用的DP内核实例的输出
314
将输出从DP存储器传递到非DP存储器
316
执行一个或多个非DP计算
318
更多非DP计算
320

图3

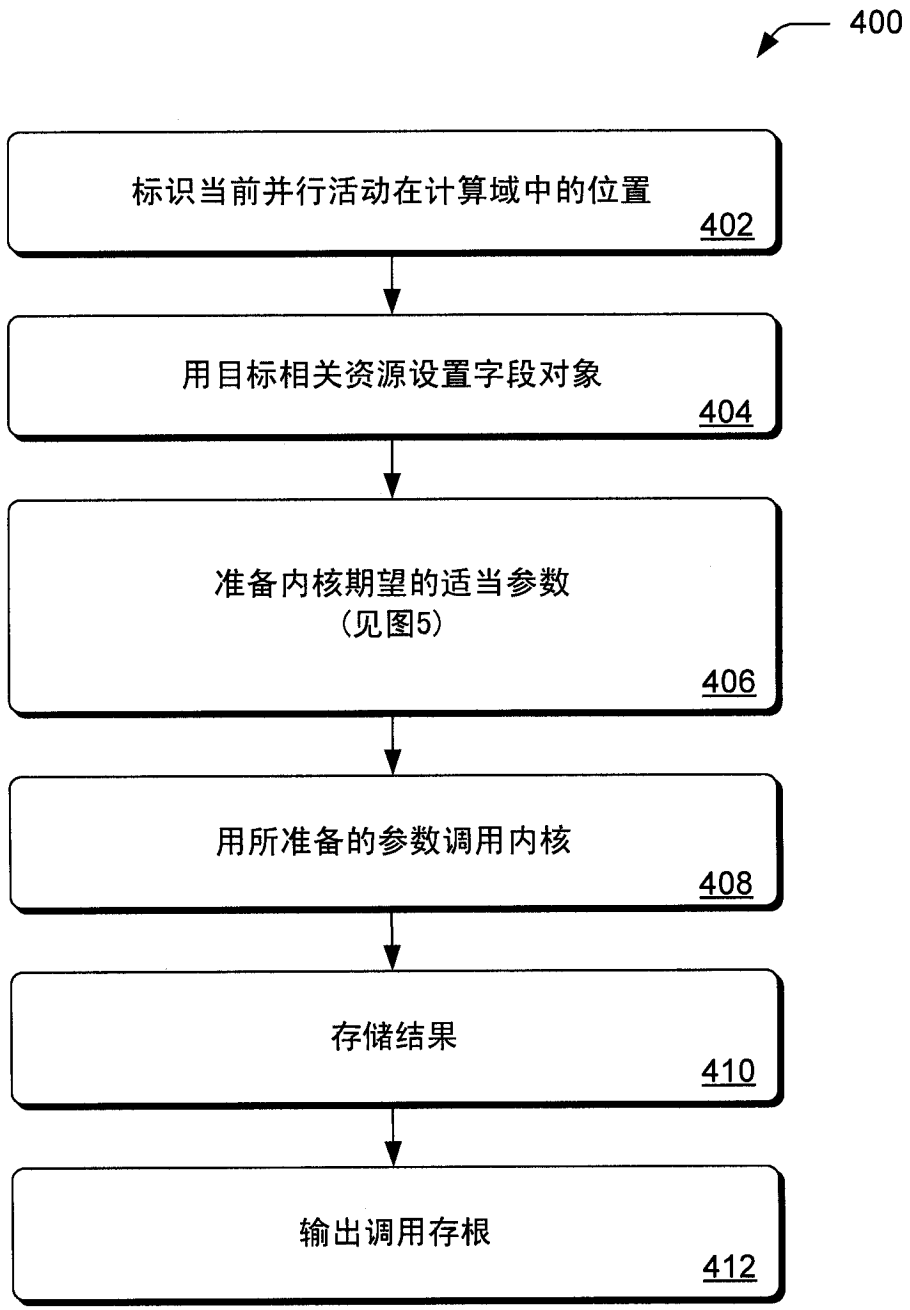

400
标识当前并行活动在计算域中的位置
402
用目标相关资源设置字段对象
404
准备内核期望的适当参数
(见图5)
406
用所准备的参数调用内核
408
存储结果
410
输出调用存根
412

图4

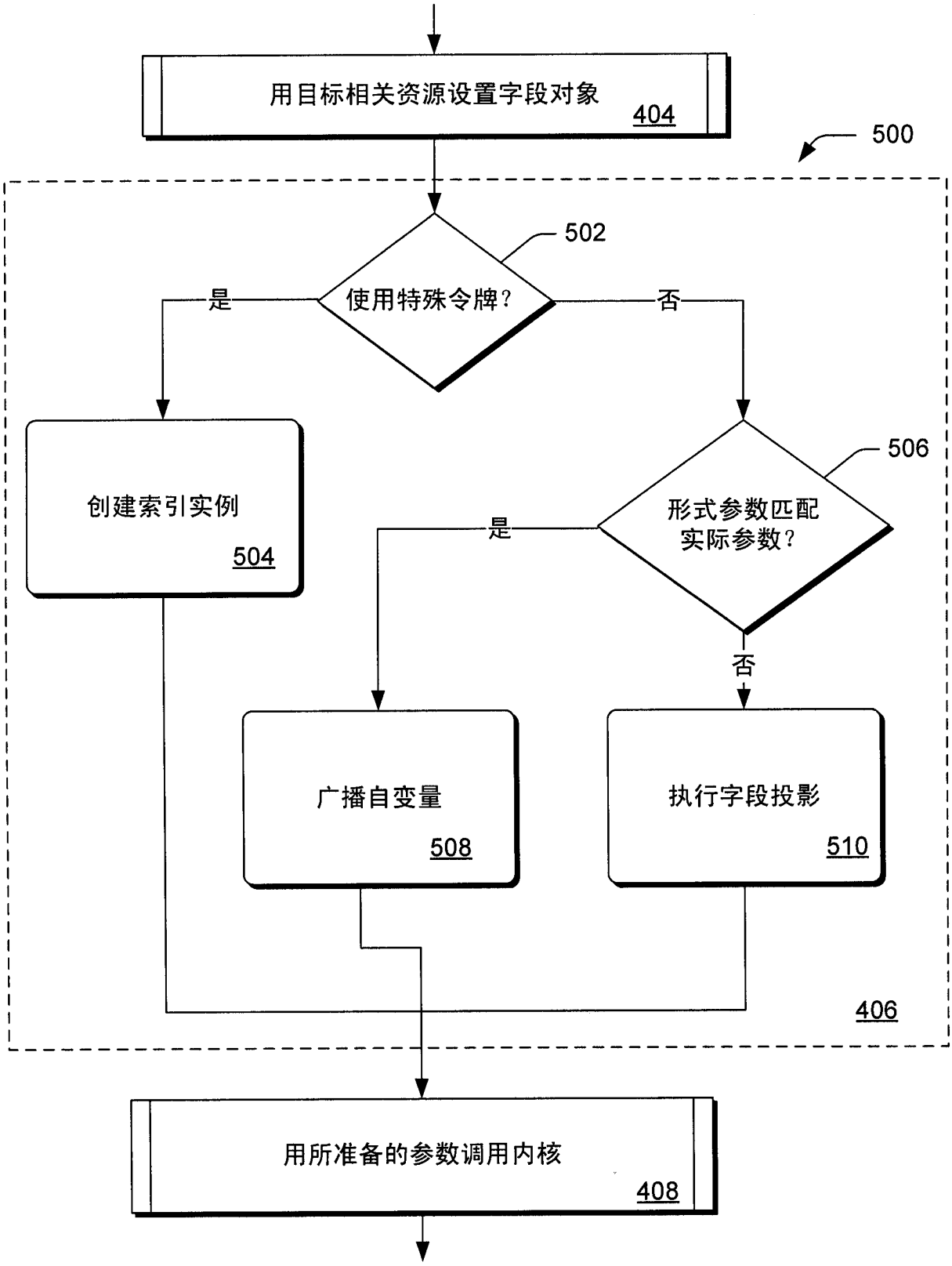

用目标相关资源设置字段对象
404
500
502
使用特殊令牌?
是
否
创建索引实例
504
506
形式参数匹配
实际参数?
是
否
广播自变量
508
执行字段投影
510
406
用所准备的参数调用内核
408

图5