(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 103917959 B
(45)授权公告日 2017.11.14

(21)申请号 201280053875.7
(22)申请日 2012.10.31
(65)同一申请的已公布的文献号
申请公布号 CN 103917959 A
(43)申请公布日 2014.07.09
(30)优先权数据
13/288,833 2011.11.03 US
(85)PCT国际申请进入国家阶段日
2014.04.30
(86)PCT国际申请的申请数据
PCT/US2012/062768 2012.10.31
(87)PCT国际申请的公布数据
WO2013/066988 EN 2013.05.10
(73)专利权人 超威半导体公司
地址 美国加利福尼亚州
(72)发明人 李·W·豪斯
本尼迪克特·R·盖斯特
迈克尔·C·休斯顿
迈克尔·曼特 马克·莱瑟
诺曼·拉宾 布赖恩·D·恩柏林
(74)专利代理机构 上海胜康律师事务所 31263
代理人 李献忠
(51)Int.Cl.
*G06F 9/52*(2006.01)
(56)对比文件
CN 101925881 A,2010.12.22,
US 2009037707 A1,2009.02.05,
CN 101739381 A,2010.06.16,
Shivali Agarwah, Saurabh Joshi, Rudrapatna K. Shyamasundar.Distributed Generalized Dynamic Barrier Synchronization.《Distributed Computing and Networking》.2011,
审查员 李雨晴

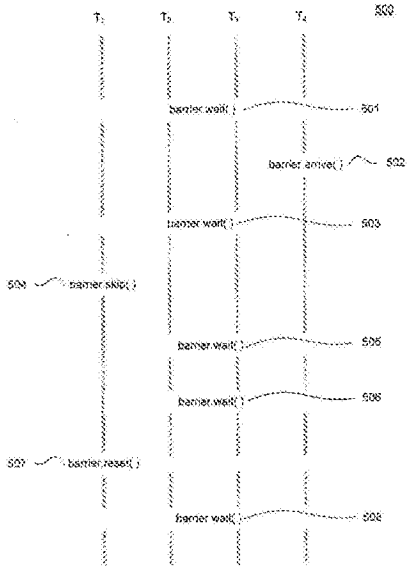
权利要求书2页 说明书10页 附图9页

(54)发明名称

用于工作项同步的方法和系统

(57)摘要

公开了用于使一个或多个处理器上的工作项同步的方法、系统和计算机程序产品实施方案。实施方案包括:通过来自所述组的第一工作项执行屏障跳越指令;以及响应于所执行的屏障跳越指令,重新配置屏障以使来自所述组的其它工作项在序列中的多个点上同步,而不要求第一工作项在多个点中的任一个上到达屏障。




CN 103917959 B

1.一种使一个或多个处理器上的一组工作项同步的方法，其包括：

通过来自所述组的第一工作项执行屏障跳越指令；以及

响应于所述执行的屏障跳越指令，重新配置屏障以使来自所述组的其它工作项在序列中的多个点上同步，而不要求所述第一工作项在所述多个点中的任一个上到达所述屏障，同时允许所述第一工作项前进而不必在所述多个点中的任一个上在所述屏障同步。

2.如权利要求1所述的方法，其还包括：

配置所述屏障以在一序列同步点处使所述组同步，其中所述序列包括所述多个点。

3.如权利要求1所述的方法，其中重新配置所述屏障以使其它工作项同步包括：

使与所述屏障相关的跳越计数递增，其中所述跳越计数在确定来自所述组的所有工作项是否到达所述屏障时被使用。

4.如权利要求1所述的方法，其还包括：

使所述其它工作项在所述多个点中的第一点和第二点处同步，其中所述第一工作项在所述第一点和所述第二点处不到达所述屏障。

5.如权利要求4所述的方法，其中使所述其它工作项同步包括：

对于到达所述屏障的所述其它工作项中的每个，确定它是否是所述其它工作项中的最后一个；以及

当所述其它工作项中的所述最后一个到达所述屏障时，将所有所述其它工作项解锁以重新开始处理。

6.如权利要求5所述的方法，其中确定它是否是所述其它工作项中的最后一个包括：

比较跳越计数和访问计数的和与释放阈值，其中当所述跳越指令被执行时所述跳越计数递增，其中当工作项到达所述屏障时所述访问计数递增，且其中所述释放阈值对应于所述组中的工作项的数量。

7.如权利要求4所述的方法，其中使所述其它工作项同步包括：

在所述第一点处使所述其它工作项同步；

部分重置所述屏障以仅使所述其它工作项同步；以及

在部分重置所述屏障以仅使所述其它工作项同步之后在所述第二点处使所述其它工作项同步。

8.如权利要求7所述的方法，其中部分重置所述屏障以仅使所述其它工作项同步包括：

将与所述屏障相关的访问计数设置到指示没有工作项到达所述屏障的初始值，其中所述访问计数代表到达所述屏障的工作项的数量；以及

使跳越计数的值持久，其中当工作项发布屏障跳越指令时，所述跳越计数递增。

9.如权利要求1所述的方法，其还包括：

通过来自所述组的任何工作项执行屏障重置指令；以及

响应于所述屏障重置指令，进一步重新配置所述屏障以要求来自所述组的所有工作项到达所述屏障，以便使所述组同步。

10.如权利要求9所述的方法，其中进一步重新配置所述屏障以要求来自所述组的所有工作项到达所述屏障包括：

将与所述屏障相关的访问计数设置到指示没有工作项到达所述屏障的初始值；以及

将与所述屏障相关的跳越计数设置到指示没有工作项发布与所述屏障相关的屏障跳

越指令的初始值。

11.如权利要求1所述的方法，其中所述组的工作项是工作组，且其中所述工作组在图形处理单元的处理元件中执行。

12.如权利要求1所述的方法，其中所述组包括来自在图形处理单元中执行的波前的两个或多个工作组的工作项。

13.如权利要求1所述的方法，其中所述组包括在CPU上执行的工作项。

14.如权利要求1所述的方法，其中配置所述屏障以在一序列同步点处使所述组同步响应于包括在库函数中的指令而被执行。

15.一种使一个或多个处理器上的一组工作项同步的系统，其包括：

一个或多个处理器；

在所述一个或多个处理器上执行的一组工作项；以及

屏障同步模块，其被配置成当被所述一个或多个处理器执行时使所述一个或多个处理器：

通过来自所述组的第一工作项执行屏障跳越指令；以及

响应于所述执行的屏障跳越指令，重新配置屏障以使来自所述组的其它工作项在序列中的多个点上同步，而不要求所述第一工作项在所述多个点中的任一个上到达所述屏障，同时允许所述第一工作项前进而不必在所述多个点中的任一个上在所述屏障同步。

16.如权利要求15所述的使一个或多个处理器上的一组工作项同步的系统，其中所述屏障同步模块还被配置成当被所述一个或多个处理器执行时使所述一个或多个处理器：

配置屏障以在一序列同步点处使所述组同步，其中所述序列包括所述多个点。

17.如权利要求15所述的使一个或多个处理器上的一组工作项同步的系统，其中所述一个或多个处理器是异构计算系统的部分。

18.如权利要求15所述的使一个或多个处理器上的一组工作项同步的系统，其中所述屏障同步模块还被配置成当被所述一个或多个处理器执行时使所述一个或多个处理器：

通过来自所述组的任何工作项执行屏障重置指令；以及

响应于所述屏障重置指令，进一步重新配置所述屏障以要求来自所述组的所有工作项到达所述屏障，以便使所述组同步。

19.一种存储命令的非临时性计算机可读介质，其中所述命令如果被执行则使同步一组工作项的方法包括：

通过来自所述组的第一工作项执行屏障跳越指令；以及

响应于所述执行的屏障跳越指令，重新配置屏障以使来自所述组的其它工作项在序列中的多个点上同步，而不要求所述第一工作项在所述多个点中的任一个上到达所述屏障，同时允许所述第一工作项前进而不必在所述多个点中的任一个上在所述屏障同步。

20.如权利要求19所述的非临时性计算机可读介质，其中所述命令如果被执行则使方法还包括：

配置屏障以在一序列同步点处使所述组同步，其中所述序列包括所述多个点。

# 用于工作项同步的方法和系统

## 技术领域

[0001] 本发明大致涉及工作项同步。

## 背景技术

[0002] 图形处理单元(GPU)通常包括理想地适合于在并行数据流上执行相同的指令的多个处理元件，如在单指令多数据(SIMD)设备中的情况或在数据并行处理中的情况。在很多计算模型中，中央处理单元(CPU)起主机或控制处理器的作用，并不干涉其它处理器(例如GPU)的专用函数，例如图形处理。

[0003] 多核CPU(其中每个CPU具有多个处理核心)为与在GPU上可用的那些函数类似的专用函数(例如图形处理)提供处理能力。多核CPU或GPU的一个或多个计算核心可以是同一裸片(例如AMD Fusion™)的部分，或可选地在不同的裸片(例如具有NVIDIA GPU的Intel Xeon™)中。最近，具有CPU和GPU(例如CellSPE™、Intel Larrabee™)两者的特征的混合核心被提出，用于通用GPU(GPGPU)型计算。GPGPU型的计算主张使用CPU来主要执行控制代码并将性能关键数据-并行代码卸载到GPU。GPU主要用作加速器。多核CPU和GPGPU计算模型的组合包括CPU核心和作为加速器目标的GPU核心两者。很多多核CPU核心具有在很多领域中与GPU可比较的性能。

[0004] 为具有CPU和GPU的异构计算平台开发了几个框架。这些框架包括斯坦福大学的BrookGPU、NVIDIA的计算统一设备架构(CUDA)和被称为科纳斯组织的行业协会的OpenCL。OpenCL框架提供用户可创建用于GPU的应用的C型开发环境。OpenCL使用户能够例如指定用于将一些计算(例如数据-并行计算)卸载到GPU 的指令。OpenCL也提供编译器和运行时间环境，其中代码可在异构或其它计算系统内被编译并执行。

[0005] 由OpenCL、CUDA和很多低级GPU中间语言体现的计算模型有时被称为单指令多线程(“SIMT”)。在SIMT模型的频繁使用的实施中，使用在矢量上的硬件掩模组的SIMD执行用于模拟比在硬件中可获得的更精细粒度的线程。

[0006] 为了有效地利用计算模型(其中CPU和GPU都可用于很多类型的代码的执行)，很多灵活的线程同步模型是需要的。当与不是传统图形处理任务的工作项一起使用时，在OpenCL中可用的常规工作项同步可能效率较低。例如，尽管在图形处理任务中的各自并发性工作项可能常常没有歧异，但是在CPU相关的应用中，执行的歧异可能相对较高。在OpenCL中提供的同步模型单独地不足以处理工作项的这样的动态行为。

[0007] 在OpenCL中，在工作组内的工作项可被同步。在工作组内，可发布屏障(barrier)指令，其具有下列语义：在工作组中的所有工作项必须在任何工作项可前进而经过屏障之前到达屏障。“屏障”阻挡到达它的所有进程，直到上面提到的语义被满足。它接着释放被阻挡的进程以重新开始它们各自的处理。然而，在OpenCL和其它类似的常规框架中，在控制流中的屏障使用被严重限制。

[0008] 图1A示出使用单个工作项(在图1A中被称为内核(kernel))以将值加载到组共享存储器空间中，其它工作项可从该存储器空间得到加载的值。加载值的工作项以及工作组

中的其它工作项被阻止越过屏障前进，直到组中的所有工作项都到达屏障。

[0009] 将值加载到组共享存储器中的上述操作也可从库函数内完成，从内核代码调用库函数。图1B示出包括屏障指令的库函数。图1C是调用库函数的内核。图1C中的代码示出阻挡调用相应库的所有工作项直到指定的工作项将数据加载到共享区域的操作。

[0010] 图1D示出一个示例，其中将对屏障的调用放置在库内部可导致不正确的操作。例如，调用包括来自具有条件(其中条件之一没有对库函数的调用)的内核的屏障指令的库函数可导致死锁。这是因为屏障只在一组的所有工作项都到达它时才释放，然而一个或多个工作项(其条件未被满足)将根本不到达屏障。

[0011] 而且，在OpenCL的工作项同步框架中，执行内核的工作组中的工作项必须在任何工作项被允许继续执行之前执行屏障指令。屏障指令必须被执行内核的工作组中的所有工作项遇到(即，在指令流中到达)。如果屏障指令在条件语句内部，则所有工作项必须进入条件即任何工作项进入条件语句并执行屏障。如果屏障指令在循环内部，则所有工作项必须在任何工作项被允许越过屏障继续执行之前针对循环的每次迭代执行屏障指令。这些限制可限制系统和编程员最佳地利用处理资源的能力。

[0012] 因此所需要的是实现工作项同步的更灵活和有效的使用的方法和系统。

## 发明内容

[0013] 公开了用于工作项的更有效和灵活的调度的方法和系统。公开了使工作项指示它永久地离开同步组使得在该同步组中的工作项的执行中出现的随后屏障不等待已宣布从工作组离开的工作项的一种技术。公开了另一技术，通过该技术，工作项可再加入同步组，以便继续与该组的其它工作项同步。所公开的技术可在各种情况下编程时在提高的处理效率和灵活性方面产生明显的优点。

[0014] 所公开的方法、系统和计算机程序产品实施方案包括由来自组的第一工作项执行屏障跳越指令并响应于所执行的屏障跳越指令而重新配置屏障以使来自该组的其它工作项在序列中的多个点上同步，而不要求第一工作项在多个点中的任一个上到达屏障。

[0015] 下面参考附图详细描述本发明的另外的实施方案、特征和优点以及本发明的各种实施方案的结构和操作。

## 附图说明

[0016] 合并在说明书中并构成说明书的部分的附图示出本发明的实施方案，并连同上面给出的一般描述和下面给出的实施方案的详细描述一起用来解释本发明的原理。在附图中：

[0017] 图1A-图1D示出呈伪码形式的常规屏障同步示例。

[0018] 图2A示出根据本发明的实施方案的屏障跳越指令(以伪码形式)。

[0019] 图2B示出根据实施方案的具有使用库调用的屏障跳越指令的内核(以伪码形式)。

[0020] 图3示出根据实施方案的屏障重置指令(以伪码形式)。

[0021] 图4A-图4C示出根据实施方案的屏障同步的示例性使用情况的伪码范例。

[0022] 图5示出根据实施方案的几个工作项随着时间的示例性流。

[0023] 图6示出根据实施方案的用于工作项同步的方法。

[0024] 图7示出根据实施方案的用于工作项同步的系统的方框图。

[0025] 图8示出根据实施方案的工作项同步模块的方框图。

## 具体实施方式

[0026] 虽然在本文使用特定应用的说明性实施方案描述了本发明，但是应理解，本发明不限于此。可以获得本文提供的教导的本 领域中的技术人员将认识到在本发明范围和本发明将具有重要效用的另外领域内的另外的修改、应用和实施方案。

[0027] 可在任何计算机系统、计算设备、娱乐系统、媒体系统、游戏系统、通信设备、个人数字助理或使用一个或多个处理器的任何系统中使用本发明的实施方案。在系统包括异构计算系统的场合，本发明可能是特别有用的。如在本文使用的术语“异构计算系统”是多种类型的处理器是可用的计算系统。

[0028] 在GPU中，被分配到处理元件的工作项被称为“工作组”。被发布用于并行执行的两个或多个工作项是“波前”。工作组可包括一个或多个波前。虽然主要关于使工作组的工作项同步描述了实施方案，但是本公开的教导可适用于使遍及任一个或多个处理器和/或访问共享存储器的进程组的工作项同步。如本文使用的术语“内核”指作为一个或多个工作项并行地执行的、具有相同的编码基数的程序和/或处理逻辑。应注意，在一些实施方案中，术语“工作项”和“线程”是可互换的。在本公开中，“工作项”和“线程”的可互换性说明例如体现在实施方案中的模型中的工作项执行的灵活模拟的或真实的独立性。

[0029] 本发明的实施方案可通过实现并发工作项之间的更有效和更灵活的同步来明显提高系统的性能。在GPU、多核CPU或例如使用SIMD或SIMT框架执行非常大数量的并发工作项的其它处理器中，实施方案通过使一些工作项能够离开和/或再加入一组工作项来提高效率，这组工作项使其在指令流中的几个点处的执行同步。例如，当特定的工作项不需要与同步组的其余部分的进一步同步时，它可发布屏障跳越指令以从同步组永久地移除本身。

[0030] 如果特定的工作项必须再次包括在同步工作组中，则屏障重置指令可随后被发布。跳越指令实际上声明相应的工作项将不在屏障被重置之前的任何点到达屏障。重置指令允许同一屏障被重新使用。

[0031] 通过使工作项能够永久地跳越屏障，各种性能提高将实现。例如，编程灵活性明显提高(如在图2A-图2B中示出的)，因而允许更有效的代码的创建。循环行为(特别是在具有大量并发工作项的系统中)也得到改进。

[0032] 例如，在常规系统(包括包含在屏障上的到达和等待操作的系统)中，所有工作项都被包括屏障的循环阻挡，直到最长的迭代工作项完成其迭代。相反，在本发明的实施方案中，不需要与组的进一步同步的工作项可退出循环，而不引起系统中的死锁。

[0033] 处理速度的提高以及功率效率的提高被实现。

[0034] 图2A示出一种函数(以伪码形式)，其中根据本发明的实施方案使用屏障跳越指令(例如b.skip())。已命名的屏障b被声明，且跳越指令在离开条件循环之后被发布。在循环内，工作项等待b，直到规定的条件被满足。

[0035] 对应于“theKernel”函数(在图2A中示出)的所有工作项在循环中时可以不重复相同或相似数量的迭代。当每个工作项离开循环时，跳越指令被发布。现有工作项对跳越指令的发布向还没有离开循环的其它工作项指示，现有工作项将不再到达屏障。因此，屏障可被

重新配置以避免在当前和随后的示例中等待现有工作项，直到至少被称为“屏障重置”的另一指令被发布为止。例如，屏障可被重新配置，使得到达屏障所需的工作项的数量可减小以解释现有的工作项。

[0036] 在常规系统中，包括在循环内的屏障函数(例如barrier()或barrier().wait)的工作项不能被有效地处理。例如，为了防止死锁，必须迫使组中的所有工作项迭代相同的次数，使得在每次迭代中，所有工作项都到达(或访问)屏障。这样的方法在工作项可能具有不同的执行路径的环境中将显然是浪费的。

[0037] 另一常规同步指令是屏障到达。然而，到达指令只从当前屏障释放调用程序。例如，如果在图2A中，跳越指令用到达指令代替，则屏障将仍然需要现有工作项在随后的屏障实例中到达屏障。因此，到达指令不提供机制，工作项可通过该机制永久地从同步组移除本身。

[0038] 图2B是根据本发明的允许选定的工作项将数据拷贝到具有其它工作项的共享空间的、以伪码形式的库函数“loadFunction”的图示。库函数包括对屏障等待指令(例如b.wait())的两个调用。这两个屏障等待指令将由调用库函数的任何工作项发布。

[0039] 在图2B中，当规定的条件被满足时，内核函数“theKernel”调用laodFunction。否则，它发布跳越指令。

[0040] 在条件(在图2B中)的else部分中的跳越指令确保不满足条件的所有工作项调用跳越指令。因此，未能满足条件的所有工作项将从屏障的两个随后实例免除，且屏障实例将不等待这些工作项。在指令流中的屏障的实例可被称为同步点。两个同步点对应于对库函数中的屏障等待(b.wait())的两个调用。死锁将不出现，因为每个工作项调用loadFunction并从而在两个同步点处到达屏障，或调用跳越指令，其使本身免于必须到达屏障的任何随后实例。

[0041] 相反，如果在图2B中使用常规到达指令而不是跳越指令，则死锁将出现。例如，虽然不满足条件的每个工作项执行到达指令，但是到达指令只从紧接的下一屏障免除发布工作项。发布工作项将被预期在第二同步点到达屏障，且因此死锁将出现。

[0042] 图3是根据实施方案的屏障重置指令(例如b.reset)的使用的以伪码形式的图示。当规定的条件被满足时，内核函数“theKernel”调用loadFunction，如图2B所示。否则，它发布跳越指令，后面是重置指令。

[0043] 如上面关于图2B描述的，跳越指令的发布确保未能满足条件的所有工作项将从两个随后的同步点免除。在同步点处的屏障将不等待这些工作项。屏障重置指令将屏障重置到其原始配置。例如，虽然跳越指令实际上减小针对该屏障的同步组的尺寸，重置指令却反转由组中的工作项发布的任何前面的跳越指令的效应。重置指令的实施要求任何未决的屏障在重置完成之前被同步。根据一些实施方案，屏障重置被实施为自同步指令，其使遍及所有工作项的同步点涉及到重置被应用到的屏障实例中。在其它实施方案中，用户可包括一个或多个同步点以确保在重置指令处的同步。图4A是根据实施方案的具有独立于其组尺寸的限定尺寸的屏障的使用的以伪码形式的图示。例如，具有原始同步组尺寸16的屏障b可被创建，而不考虑多少工作项在工作组中，从该工作组调用屏障。创建具有限定尺寸16的屏障b使在屏障b上调用等待指令的任何进程能够与在屏障b上同步的该组中的任何15个工作项同步。

[0044] 具有限定尺寸的屏障可被创建用于一种应用，其中已知只有特定数量的工作项将满足限定的条件。以这种方式，如图4A所示，满足条件的工作项可在屏障b上同步，而其它工作项采用“else”路径。因为屏障只等待满足条件的限定数量的工作项，所以没有死锁条件出现，且不满足条件的工作项不需要发布跳越指令。

[0045] 在图4B的示例性图示中，屏障用于使两组工作项同步。一组满足规定的条件。可使用屏障b1和b2来单独地同步不满足该条件的另一组。工作项属于b1同步组或b2同步组，并可向它不属于的组发布跳越指令。

[0046] 在图4C的示例性图示中，使用以伪码形式的分级屏障。满足第一规定条件的工作项在屏障b1之后的等待指令上同步，而其它工作项跳越屏障b1。满足第一规定条件的那些工作项遇到另一屏障b2。在遇到第二屏障b2的工作项中，满足第二条件的那些工作项等待b2，其它工作项跳越b2。如在伪码中所示的，基于屏障b1的当前状态来创建屏障b2。

[0047] 更具体地，屏障b2的尺寸(即，b2强制同步的工作项的数量)由在b1之后不发布跳越指令的工作项的数量表示。在图4C中，这个数字是满足第一条件的工作项的数量。当b1被跳越时，其它实施方案可能需要屏障更新b2的基础实施。而且，为了避免可能在b2的更新中的一些情况中出现的竞争条件，可通过在b1跳越操作之前有同步点来保护b1跳越操作。

[0048] 图5是根据本发明的实施方案的几个工作项T1、T2、T3和T4随着时间的示例性流程500的图示。如所示，T1、T2、T3和T4并发性地或实质上并发性地开始。在每个工作项中的第一屏障等待指令使它在同步点501同步。在同步点501的同步涉及工作项T1、T2、T3和T4等待它们当中的最后一个工作项到达501，并接着并发性地或实质上并发性地重新开始执行。

[0049] 在点502，工作项T4发布屏障到达指令。屏障到达指令通知屏障的下一实例不等待T4，且T4前进，而不必在屏障的第二实例处同步。

[0050] 在同步点503，工作项T1、T2和T3基于指令流中的第二屏障等待指令在屏障的第二实例上同步。当所有工作项T1、T2和T3到达同步点503时，它们并发性地或实质上并发性地重新开始执行。

[0051] 在点504，T1发布屏障跳越指令。屏障跳越指令通知屏障的任何随后的实例不等待T1，且T1前进而不必在屏障的随后实例处同步。

[0052] 在同步点505，工作项T2、T3和T4同步。注意，虽然T4以前发布屏障到达指令(在502)，这个以前的发布只从下一出现的屏障(即，在同步点503)免除T4。在同步点505的同步之后，T2、T3和T4前进以再次在同步点506同步。当在同步点506同步时，T2和T3每个遇到了在它们的各自指令流中的屏障等待指令的四个实例。此时，T4只遇到了三个实例。在同步点505和506的屏 障不阻挡T1，因为T1在同步点505和506之前在504发布了屏障跳越指令。

[0053] 在时间上的点507处，在506出现之后，T1发布屏障重置指令。这将屏障重置到其原始配置。原始屏障配置成在所有四个工作项T1、T2、T3和T4上同步。重置使T1在同步点507与T2、T3和T4同步。可通过实施重置作为自同步指令或通过与重置相关的用户指定的同步指令在点507处实现同步。

[0054] 在点507处的同步之后，T1-T4基于在每个工作项中的等待指令在同步点508处继续进行同步。

[0055] 图6是根据实施方案的工作项同步的示例性方法600的图示。

[0056] 在操作602处，开始一组工作项。工作项可以是相同代码的多个工作项。在相同代

码的各自工作项中执行的实际指令序列可以是相同的或不同的，取决于条件估计等。根据另一实施方案，工作项不都是相同的代码，并可包括彼此共享一个或多个同步点的任何工作项。

[0057] 多个工作项可并发性地或非并发性地开始。可在CPU、GPU上、在两个或多个GPU上、在CPU的两个或多个核心上或一个或多个GPU和一个或多个CPU核心的任何组合上执行工作项。根据实施方案，多个工作项是在GPU的一个处理元件上执行的工作组。

[0058] 在操作604处，创建屏障b。可通过在声明屏障b的工作项上执行指令来创建或例示屏障b。根据实施方案，当系统遇到屏障b的声明的第一实例时，屏障的实例在存储器中被创建。随后声明屏障b的工作项接收到对已经创建的屏障b的引用。屏障的基础实施在各自的框架和/或系统中可以是不同的。计数信号量是示例性机制，具有上述语义的屏障可通过该机制来实施。

[0059] 存储器中的屏障b对象的创建包括初始化动态存储器和/或硬件中的一个或多个存储器位置和/或寄存器。例如，关于屏障b，需要如下所述维持几个计数。可在写和读到那些存储器位置时使用适当的并发控制机制在动态存储器中维持屏障对象以及所有计数。根据另一实施方案，虽然对应于屏障b的对象在动态存储器中被例示，但是对应的计数被维持在特定的硬件寄存器中。

[0060] 在操作606，初始化屏障b所需的各种计数被确定，且它们的初始值被设置。访问工作项的数量(“访问计数”)限定已到达屏障b的工作项的数量。当工作项发布屏障等待指令或等效指令时，工作项已“到达”屏障。访问计数可被初始化为0。在屏障b上执行屏障跳越指令的工作项的数量可在跳越的计数(“跳越计数”)中被跟踪。屏障释放阈值(“释放阈值”)是屏障正等待的工作项的数量。根据实施方案，屏障b被初始化具有等于在操作602开始的组中的工作项的数量的释放阈值。根据另一实施方案，创建(在操作604)具有限定尺寸的屏障b，而不考虑在操作602开始的组的尺寸。因此，释放阈值被初始化为限定尺寸。

[0061] 在操作608，在执行来自指令流的任何数量的其它指令之后，工作项x到达同步指令。同步指令可以是但不限于下列项之一：屏障等待、屏障到达、屏障跳越和屏障重置。

[0062] 在操作610，做出同步指令是否是屏障等待指令的决定，且如果是，方法600继续进行到操作612。

[0063] 在操作612，访问计数被更新。根据实施方案，访问计数递增一以指示工作项x到达屏障。

[0064] 在操作614，将更新的访问计数和跳越计数的和与释放阈值进行比较。如果该和等于释放阈值，则工作项x是到达屏障的最后一个工作项，且屏障在操作618释放。根据实施方案，释放屏障使一个或多个计数值被重置且阻挡的工作项重新开始执行。

[0065] 屏障的释放在操作620引起访问计数的重置且在操作622引起在屏障上被阻挡的所有工作项的重新开始。在操作620，根据实施方案，访问计数被重置为0。注意，重置访问计数清除早些时候出现的屏障到达指令的任何影响。然而，重置访问计数并不清除早些时候出现的任何屏障跳越指令的影响。如在来自块的屏障释放的情况中完成的，仅仅访问计数的重置可被考虑为在屏障上的“部分重置”操作。如在本文使用的术语“部分重置”传达应用于不发布跳越指令的工作项的屏障部分被重置。

[0066] 在操作622，所有阻挡的工作项重新开始执行。根据实施方案，阻挡的工作项可等

待信号量，且信号量被重置，使得阻挡的工作项可重新开始执行。例如，在硬件或软件中实施的计数信号量可用于阻挡工作项。在完成操作622之后，方法600继续进行到操作608。

[0067] 如果在操作614确定访问计数和跳越计数的和不等于释放阈值，则在操作616，工作项x被阻挡。根据实施方案，可通过使工作项等待信号量来执行工作项x的阻挡。当操作618-622针对屏障出现时，工作项x将随后被释放。操作619代表在例如通过操作618-622释放屏障之后工作项x的执行的继续。

[0068] 如果在操作610决定指令不是屏障等待指令，则在操作624决定指令是否是屏障到达。如果是，则在操作626更新访问计数。根据实施方案，访问计数递增一以指示工作项x已到达屏障。在更新访问计数之后，在操作628，工作项x继续指令流的执行。随后，当下一同步指令被遇到时，处理继续进行到操作608。

[0069] 如果在操作624决定指令不是屏障到达指令，则在操作630决定指令是否是屏障跳越。如果是，则在操作632更新跳越计数。根据实施方案，跳越计数递增1。然后在操作636，工作项x继续执行并当下一同步指令被遇到时继续进行到操作608。

[0070] 如果在操作630决定指令不是屏障跳越指令，则在操作638确定指令是否是屏障重置。如果是，则在操作640，屏障b被重置。根据实施方案，响应于屏障重置指令而重置屏障包括将访问计数和跳越计数设置为0。因此，在屏障重置在屏障b上被执行之后，屏障b被重置到其原始状态。在一些情况下，如果组中的所有工作项都未在屏障处被同步，则竞争条件可出现。因此，在一些实施方案中，屏障重置指令可被实施为自同步指令，并可引起遍及所有工作项的同步点。在其它实施方案中，用户可包括一个或多个同步点以确保在重置指令处的同步。在操作640之后，当在指令流中遇到下一同步指令时，处理可继续到操作608。

[0071] 如果在步骤638确定指令不是屏障重置指令，则在操作639，工作项x可继续执行。在操作639之后，当在指令流中遇到下一同步指令时，处理可继续到操作608。

[0072] 根据另一实施方案，不是维持单独的跳越计数和屏障的释放阈值，可只维持释放阈值。例如，释放阈值可递减以反映工作项已执行屏障跳越指令。然后，当屏障重置指令被发布时，计数的重置包括将释放阈值重置到其原始尺寸。实际上，根据该方法，执行屏障跳越的一定数量的工作项可被认为离开同步组。

[0073] 图7是根据实施方案的工作项同步的系统的方框图图示。在图7中，示例性异构计算系统700可包括一个或多个CPU(例如CPU 701)以及一个或多个GPU(例如GPU 702)。异构计算系统700也可包括系统存储器703、持久存储设备704、系统总线705、输入/输出设备706和屏障同步器709。

[0074] CPU 701可包括市场上可买到的控制处理器或定制控制处理器。CPU 701例如执行控制异构计算系统700的操作的控制逻辑。CPU 701可以是多核CPU，例如具有两个CPU核心741和742的多核CPU。除了任何控制电路以外，CPU 701还包括分别是CPU核心741和742的CPU高速缓存存储器743和744。CPU高速缓存 存储器743和744可用于在CPU核心741和742上的应用的执行期间分别临时存储指令和/或参数值。

[0075] 例如，CPU高速缓存存储器743可用于在CPU核心741上的控制逻辑指令的执行期间临时存储来自系统存储器703的一个或多个控制逻辑指令、变量的值或恒定参数的值。CPU 701也可包括专用矢量指令处理单元。例如，CPU核心742可包括可有效地处理矢量化指令的流SIMD扩展(SSE)单元。本领域中的技术人员将理解，CPU 701可包括比选定示例中的CPU核

心更多或更少的CPU核心，并且也可没有高速缓存存储器或具有更复杂的高速缓存存储器分级结构。

[0076] GPU 702可包括市场上可买到的图形处理器或定制设计的图形处理器。GPU 702例如可执行用于选定功能的专用代码。通常，GPU 702可用于执行图形功能，例如图形流水线计算和在显示器上渲染图像。

[0077] GPU 702包括GPU全局高速缓存存储器710及一个或多个计算单元712和713。图形存储器707可包括在GPU 702中或耦合到GPU 702。每个计算单元712和713分别与GPU本地存储器714和715相关。每个计算单元包括一个或多个GPU处理元件(PE)。例如，计算单元712包括GPU处理元件721和722，且计算单元713包括GPU PE 723和724。

[0078] 每个GPU处理元件721、722、723和724分别与至少一个专用存储器(PM)731、732、733和734相关。每个GPU PE可包括一个或多个标量和矢量浮点单元。GPU PE也可包括专用单元例如平方根倒数单元和正弦/余弦单元。GPU全局高速缓存存储器710可耦合到系统存储器(例如系统存储器703)和/或图形存储器(例如图形存储器707)。

[0079] 系统存储器703可包括至少一个非持久存储器，例如动态随机存取存储器(DRAM)。系统存储器703可在应用或其它处理 逻辑的部分的执行期间存储处理逻辑指令、恒定值和变量值。例如，屏障同步器709的控制逻辑和/或其它处理逻辑可在通过CPU701执行屏障同步器709期间存在于系统存储器703内。如在本文使用的术语“处理逻辑”指控制流指令、用于执行计算的指令和用于对资源的相关访问的指令。

[0080] 持久存储器704包括能够存储数字数据的一个或多个存储设备，例如磁盘、光盘或闪存。持久存储器704可例如存储屏障同步器709的指令逻辑的至少部分。在异构计算系统700启动时，操作系统和其它应用软件可从持久存储器704加载到系统存储器703中。

[0081] 系统总线705可包括外围部件互连(PCI)总线、工业标准架构(ISA)总线或这样的设备。系统总线705也可包括网络(例如局域网(LAN))连同耦合部件(包括异构计算系统700的部件)的功能。

[0082] 输入/输出接口706包括连接用户输入/输出设备(例如键盘、鼠标、显示器和/或触摸屏)的一个或多个接口。例如，可通过键盘和鼠标连接的用户接口706向异构计算系统700提供用户输入。异构计算系统700的输出可通过用户接口706输出到显示器。

[0083] 图形存储器707耦合到系统总线705和GPU 702。图形存储器707通常用于存储从系统存储器703传输的数据用于由GPU快速访问。例如，在GPU 702和图形存储器707之间的接口可以比系统总线接口705快几倍。

[0084] 屏障同步器709包括在GPU 702和CPU 701中的任一或两个上的使功能同步的逻辑和处理逻辑。屏障同步器709可配置成使全局地遍及计算机中的处理器组、在每个单独的处理器中和/或在处理器的每个处理元件内的工作项同步。下面还关于图8描述了屏障同步器709。本领域中的技术人员将理解，可使用软件、固件、硬件或其任何组合来实施屏障同步器。当在软件中实施时，例如屏障 同步器709可以是用C或OpenCL写的计算机程序，其当被编译和执行时存在于系统存储器703中。在源代码形式和/或编译可执行形式中，屏障同步器709可存储在持久存储器704中。在一个实施方案中，用硬件描述语言(例如Verilog、RTL、网表)规定了屏障同步器709的一些或全部功能，以能够通过掩模作品/光掩模的产生来最终配置制造过程以产生体现本文描述的发明的方面的硬件设备。

[0085] 本领域中的技术人员将理解，异构计算系统700可包括比图7所示的更多或更少的部件。例如，异构计算系统700可包括一个或多个网络接口和/或软件应用例如OpenCL框架。

[0086] 图8是根据实施方案的屏障同步器800的图示。屏障同步器800包括工作项阻挡模块802、屏障释放模块804、屏障工作项组模块806、屏障跳越模块808和屏障重置模块810。而且，屏障同步器800可包括屏障寄存器812。根据实施方案，屏障同步器800包括在屏障同步器709中。

[0087] 工作项阻挡模块802操作来在屏障上阻挡一个或多个工作项。可使用信号量(例如计数信号量)和寄存器来实施屏障。工作项阻挡可通过使阻挡的工作项等待信号量来实施。信号量可在硬件或软件中实施。当遇到屏障等待指令时，可阻挡工作项。工作项阻挡模块可例如包括处理逻辑以实施操作，包括方法600的操作616。

[0088] 屏障释放模块804操作来在足够数量的工作项到达屏障时释放屏障。如上所述，屏障可使用信号量来实施，且释放屏障可包括释放信号量。当遇到屏障等待指令且工作项证明是最后一个工作项以完成一定数量的工作项到达屏障的要求时，可释放工作项。屏障释放模块可例如包括处理器逻辑以实施操作，包括方法600的操作618-622。

[0089] 屏障工作项组模块806操作来掌握在各种执行工作项当中的同步组的线索。屏障工作项组模块806也可操作来根据组构造来 创建和初始化屏障。根据实施方案，屏障工作项组模块806可例如包括处理逻辑以实施操作，包括方法600的操作602-606。

[0090] 屏障跳越模块808操作来实施屏障跳越指令。例如，屏障跳越指令将使相应屏障的跳越计数被更新。根据实施方案，屏障跳越模块808可包括处理逻辑以实施操作，包括方法600的操作630-636。

[0091] 屏障重置模块810操作来实施屏障重置指令。例如，屏障重置指令将使与屏障有关的访问计数和跳越计数被重置，和/或释放阈值被调整。屏障重置模块810可例如包括处理逻辑以实施操作，包括方法600的操作638-640。

[0092] 屏障寄存器812包括与屏障有关的硬件和/或软件寄存器。屏障寄存器812可包括屏障记录814，其包括多个寄存器和/或存储器位置。示例性屏障记录814包括屏障标识符822、锁824、阻挡工作项计数826、到达工作项计数828、跳越工作项计数830和阈值832。屏障标识符822可以是指针、索引或唯一地识别屏障的存储器位置或寄存器的其它标识符。

[0093] 锁824可以是对信号量或其它实体的指针或引用，进程在所述实体上被阻挡。阻挡工作项计数826是在屏障上被阻挡的工作项的数量。到达工作项计数828是发布屏障到达指令的工作项的数量。跳越工作项计数830是发布屏障跳越指令的工作项的数量。阈值832是释放阈值或同步组的组尺寸。

[0094] 结论

[0095] 概述和摘要章节可阐述如发明人设想的本发明的一个或多个但不是所有示例性实施方案，且因此并不用来以任何方式限制本发明和所附权利要求。

[0096] 上面在示出指定功能的实施及其关系的功能构造块的帮助下描述了本发明。为了描述的方便，这些功能构造块的边界在本文 被任意界定。可界定可选的边界，只要指定功能及其关系被适当地执行。

[0097] 特定的实施方案的前述描述充分揭露本发明的一般性质，使得其他人可通过应用在本领域的技术范围内的知识来容易修改和/或适应这样的特定实施方案的各种应用而没

有过度的实验，而不偏离本发明的一般概念。因此，基于本文提出的教导和指导，这样的适应和修改被规定为在所公开的实施方案的等效形式的意义和范围内。应理解，本文的用语或术语是为了描述而不是限制的目的，使得本说明书的术语或用语由技术人员按照教导和指导来解释。

[0098] 本发明的广度和范围不应被上述示例性实施方案中的任一个限制，而应只根据下面的权利要求及其等效形式来被限定。

```
Kernel loadKernel(global float *data, local float *sharedData)
{
  // get workitem 0 in the group load the group's constant from data
  // and place it in the shared array
  if( get_localId(0) == 0) {
    sharedData[0] = data[get_group_id(0)];
  }
  // Ensure all workitems wait for the operation to complete
  barrier(CLK_GLOBAL_MEM_FENCE);
   ... // do something using the shared value
}
```

图1A

```
float loadFunction (global float *data, local float *sharedData)
{
  // get workitem 0 in the group load the group's constant from data
  // and place it in the shared array
  if( get_localId(0) == 0) {
     sharedData[0] = data[get_group_id(0)];
   }

// Ensure all workitems wait for the operation to complete
  barrier(CLK_GLOBAL_MEM_FENCE);
  return sharedData[0];
}
```

图1B

```
kernel theKernel(global  float *data, local float *sharedData)
{
  float val= loadFunction(data, sharedData);
  // do something using val
}
```

图1C

```
kernel theKernel(global float *data, local float *sharedData)
{
  If ( someCondition ) {
    float val= loadFunction(data, sharedData);
    // do something using val
  } else {
    // we know we don't need to do that load here
  }
}
```

图1D

```
kernel theKernel(global float *data, local float *sharedData)
{
  barrier b;
  while( someCondition ) {
    b. wait();
    // do something
  }
  b.skip();
}
```

图2A

```
kernel theKernel(global float *data, local float *sharedData)
{
  barrier b;
  if( someCondition ) {
    // pass the barrier object down to the called function
    float val= loadFunction(data, sharedData, b);
    // do something using val
  } else {
    // instead of arrive, skip here. That way loadFunction can use the barrier
    //  more than once while maintaining correct behavior. If arrive were used
    // waiting a second time in loadFunction would
    // cause a wait on this workitem too, which is undesired behavior.
    b.skip();
    // we know we don't need to do that load here
  }
}

float loadFunction (global float *data, local float *sharedData, barrier b)
{
  // get workitem 0 in the group load the group's constant from data
  // and place it in the shared array
  if( get_localId(0) == 0) {
    sharedData[0] = data[get_group_id(0)];
  }
  // Wait specifically on the barrier we passed:
  b.wait();
  // Communicate in some way
  // Wait a second time
  b.wait();
  return sharedData[0];
}
```

图2B

```
kernel theKernel(global float *data, local float *sharedData)
{
  barrier b;
  if( someCondition ) {
    // pass the barrier object down to the called function
    float val= loadFunction(data, sharedData, b);
    // do something using val
  } else {
    // instead of arrive, skip here. That way loadFunction can use the barrier more than once while
    // maintaining correct behavior. If arrive were used waiting a second time in load Function would
    // cause a wait on this workitem too, which is undesired behavior.
    b.skip();
    // we know we don't need to do that load here
  }
  b.reset();
  //Do more
  b.wait();
  // Do more post synchronization of "all" workitems
}
```

图3

```
kernel theKernel(global float *data, local float *sharedData)
{
  // A barrier for 16 workitems
  barrier b(16);
  If ( someCondition that applies only to 16 workitems in the set) {
    // pass the barrier object down to the called function
    float val= loadFunction(data, sharedData, b);
    // do something using val
  } else {
    // We know that the first condition dealt with only 16 workitems. The rest will
follow this path
    // and hence do not need to notify the barrier of their wish to skip.
  }
}
```

图4A

```
kernel theKernel(global float *data, local float *sharedData)
{
  // create two barriers to seperately synchronize two groups of workitems
  barrier b1, b2;
  if ( someCondition that applies only to b1 workitems) {
        // do something
        b2.skip();
        b1.wait();
        // do something
  } else { // else applies only to b2 workitems
        // do something
        b1.skip();
        b2.wait();
        // do something
  }
}
```

图4B

```
kernel theKernel(global float *data, local float *sharedData)
{
  // create outer barrier b1
  barrier b1;
  if ( someCondition ) {
        // do something
        b1.wait();
        // do something
        // create inner barrier b2
        barrier b2(b1);
        if ( someCondition2 ) {
              b2.wait();
        }
        b2.skip();
        // do something
  }
  // do something
  b1.skip();
  // do something
}
```

图4C

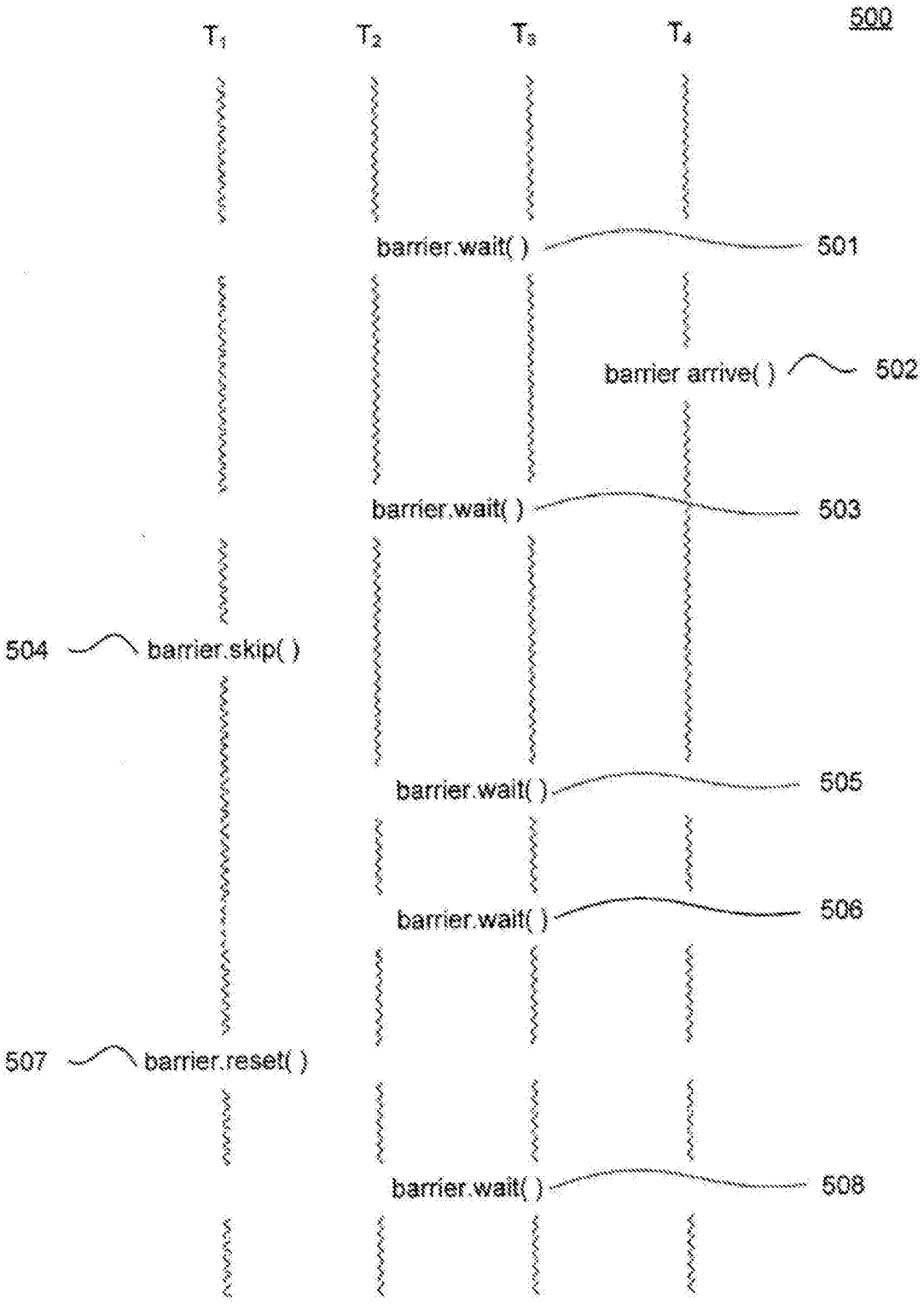

500
T1
T2
T3
T4
barrier.wait( )
501
barrier arrive( )
502
barrier.wait( )
503
504
barrier.skip( )
barrier.wait( )
505
barrier.wait( )
506
507
barrier.reset( )
barrier.wait( )
508

图5

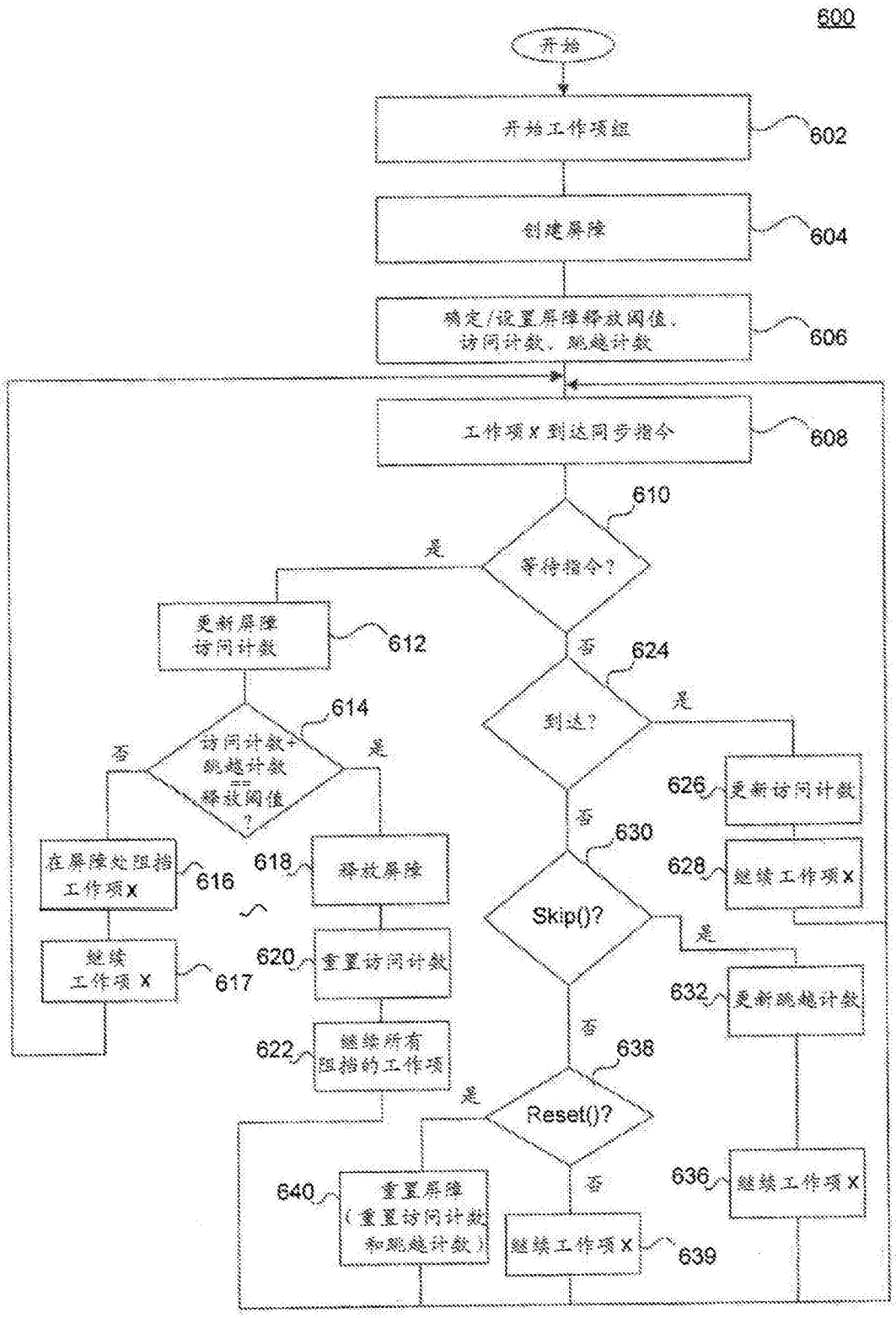

600
开始
开始工作项组
602
创建屏障
604
确定/设置屏障释放阈值、访问计数、跳越计数
606
工作项 x 到达同步指令
608
610
等待指令?
是
更新屏障访问计数
612
否
624
到达?
是
626
更新访问计数
否
614
访问计数+跳越计数==释放阈值?
否
是
在屏障处阻挡工作项x
616
618
释放屏障
630
Skip()?
628
继续工作项x
是
继续工作项 x
617
620
重置访问计数
632
更新跳越计数
否
622
继续所有阻挡的工作项
638
是
Reset()?
否
636
继续工作项x
640
重置屏障(重置访问计数和跳越计数)
继续工作项x
639

图6

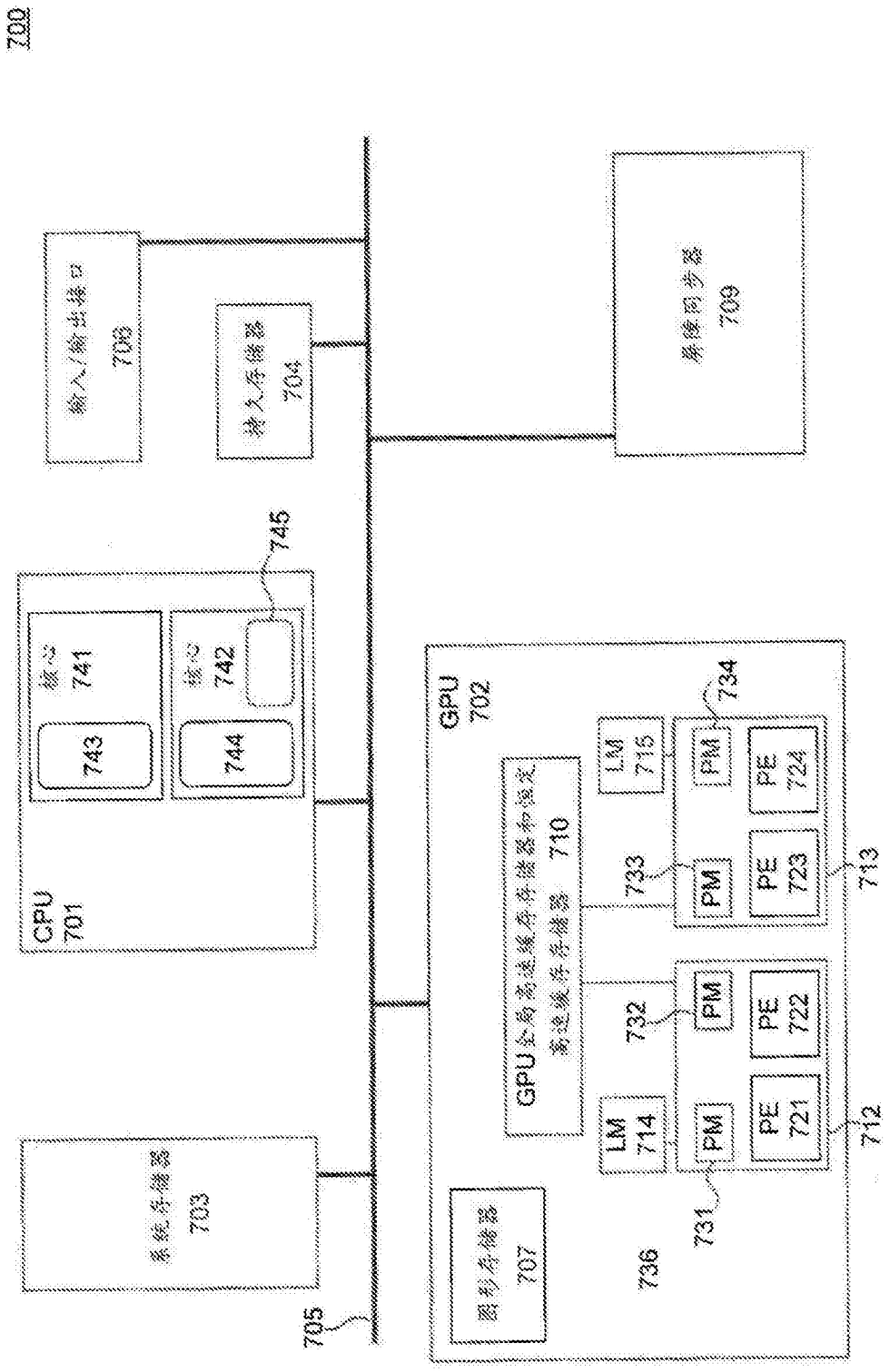

700
输入/输出接口
706
辅助存储器
704
易挥发存储器
709
745
核心
741
743
核心
742
744
CPU
701
GPU
702
GPU全局高速缓存储器和锁定高速缓存储器 710
LM
716
PM
734
PE
724
733
PM
PE
723
713
732
PM
PE
722
LM
714
PM
PE
721
712
731
736
图形存储器
707
系统存储器
703
705

图7

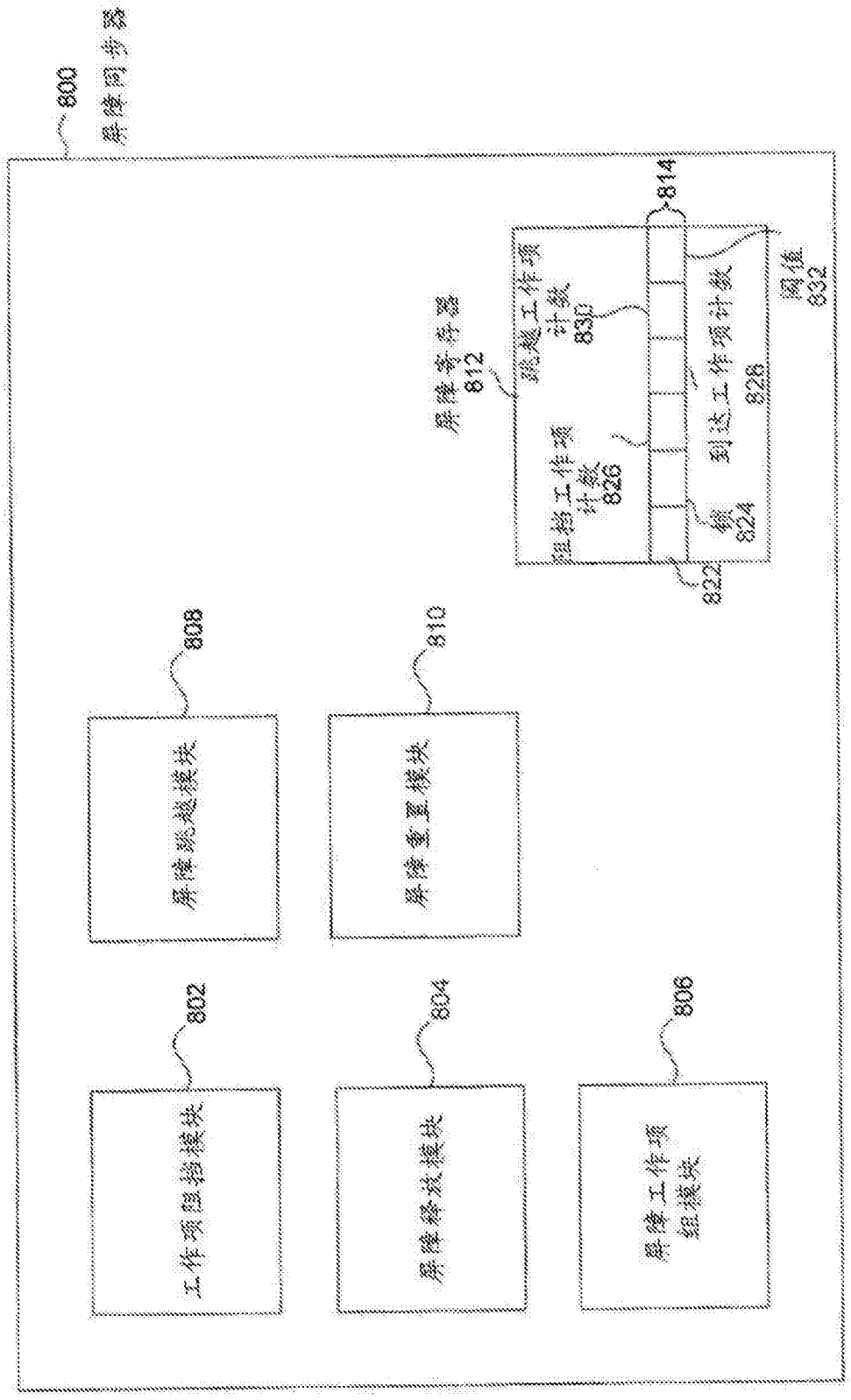

800
屏障同步器
屏障寄存器
812
跳越工作项计数
830
814
网值
832
阻挡工作项计数
826
锁
824
到达工作项计数
828
822
808
屏障跳越模块
810
屏障重置模块
802
工作项阻挡模块
804
屏障释放模块
806
屏障工作项组模块

图8