软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(2):1028−1047 [doi: 10.13328/j.cnki.jos.006984]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

面向 GPU 并行编程的线程同步综述*

高 岚[1], 赵雨晨[1], 张伟功[1], 王 晶[2], 钱德沛[3]

[1](首都师范大学 信息工程学院, 北京 100048)

[2](中国人民大学 信息学院, 北京 100872)

[3](北京航空航天大学 计算机学院, 北京 100191)

通信作者: 张伟功, E-mail: zwg771@cnu.edu.cn

摘 要: 并行计算已成为主流趋势. 在并行计算系统中, 同步是关键设计之一, 对硬件性能的充分利用至关重要. 近年来, GPU (graphic processing unit, 图形处理器) 作为应用最为广加速器得到了快速发展, 众多应用也对 GPU 线程同步提出更高要求. 然而, 现有 GPU 系统却难以高效地支持真实应用中复杂的线程同步. 研究者虽然提出了很多支持 GPU 线程同步的方法并取得了较大进展, 但 GPU 独特的体系结构及并行模式导致 GPU 线程同步的研究仍然面临很多挑战. 根据不同的线程同步目的和粒度对 GPU 并行编程中的线程同步进行分类. 在此基础上, 围绕 GPU 线程同步的表达和执行, 首先分析总结 GPU 线程同步存在的难以高效表达、错误频发、执行效率低的关键问题及挑战; 而后依据不同的 GPU 线程同步粒度, 从线程同步表达方法和性能优化方法两个方面入手, 介绍近年来学术界和产业界对 GPU 线程竞争同步及合作同步的研究, 对现有研究方法进行分析与总结. 最后, 指出 GPU 线程同步未来的研究趋势和发展前景, 并给出可能的研究思路, 从而为该领域的研究人员提供参考.

关键词: 通用图形处理器 (GPGPU); 并行编程; 线程同步; 性能优化

中图法分类号: TP301

中文引用格式: 高岚, 赵雨晨, 张伟功, 王晶, 钱德沛. 面向GPU并行编程的线程同步综述. 软件学报, 2024, 35(2): 1028–1047. http://www.jos.org.cn/1000-9825/6984.htm

英文引用格式: Gao L, Zhao YC, Zhang WG, Wang J, Qian DP. Survey on Thread Synchronization in GPU Parallel Programming. Ruan Jian Xue Bao/Journal of Software, 2024, 35(2): 1028–1047 (in Chinese). http://www.jos.org.cn/1000-9825/6984.htm

Survey on Thread Synchronization in GPU Parallel Programming

GAO Lan[1], ZHAO Yu-Chen[1], ZHANG Wei-Gong[1], WANG Jing[2], QIAN De-Pei[3]

[1](Information Engineering College, Capital Normal University, Beijing 100048, China)

[2](School of Information, Renmin University of China, Beijing 100872, China)

[3](School of Computer Science and Engineering, Beihang University, Beijing 100191, China)

Abstract: Parallel computing has become the mainstream. Among all the parallel computing systems, synchronization is one of the critical designs and is imperative to fully utilize the hardware performance. In recent years, GPU, as the most widely used accelerator, has developed rapidly, and many applications have placed greater demands on GPU thread synchronization. However, current GPUs cannot support thread synchronization efficiently in many real-world applications. Although many approaches have been proposed to support GPU thread synchronization and much progress has been made, the unique architecture and parallel pattern of GPUs still lead to many challenges in GPU thread synchronization research. In this study, thread synchronization in GPU parallel programming is divided into different categories according to different synchronization purposes and granularity. Around the synchronization expression and execution, the key problems and challenges of synchronization on GPUs are firstly analyzed, i.e., being difficult to express efficiently, incurring frequent concurrency bugs, and low execution efficiency. Secondly, the study introduces the research on synchronization for thread

* 基金项目: 国家自然科学基金 (62202317, 62076168); 北京市自然科学基金 (4214063); 北京市教委科技一般项目 (KM202110028011)
收稿时间: 2022-09-23; 修改时间: 2023-03-12; 采用时间: 2023-06-07; jos 在线出版时间: 2023-10-18
CNKI 网络首发时间: 2023-10-19

contention and synchronization for thread cooperation on GPUs in academia and industry in recent years from two aspects of thread synchronization expression method and performance optimization method based on different GPU thread synchronization granularity. Then the existing research methods are analyzed. On this basis, the study points out the future research trends and development prospects of GPU thread synchronization and feasible research methods, providing a reference for researchers in this field.

Key words: general-purpose computing on graphics processing unit (GPGPU); parallel programming; thread synchronization; performance optimization

在并行计算系统中, 同步是关键设计之一, 直接影响系统中各计算单元的协同, 对硬件性能的充分利用至关重要. 近年来, 受到功耗、复杂性、可靠性等方面的限制, 传统多核/众核处理器性能已达到上限. 而数据密集型和计算密集型应用日益广泛, 巨大的计算量以及天然的并行性促进了各种计算加速处理器的出现. 在众多并行计算处理器中, GPU (graphic processing unit, 图形处理器) 并行规模大、数值计算能力强, 是目前应用最为广泛的通用计算加速器, 并在科学计算、大数据处理与分析、深度学习等领域取得了令人瞩目的应用加速效果.

图形处理应用具有计算量大、并行度高、数据访问规则等特征, 因此在设计之初, GPU 并未过多考虑对线程同步的支持. 例如, 为有效提高系统吞吐 (throughput), GPU 采用单指令多线程 (single instruction multiple threads, SIMT) 的执行模式, 并通过大规模 (数以万计) 线程的并行执行掩盖访存时延. 另外, GPU 还采用弱一致性存储模型 (weak memory consistency), 以进一步降低访存时延. 上述 GPU 体系结构的特点对并行度高、数据访问较规则的应用非常友好, 然而却导致并行线程间的同步存在如下问题.

(1) 难以高效表达. GPU 线程组织结构复杂, 各层线程组织单元的执行和调度方式不同, 给不同粒度线程同步的表达带来难度. 另外, GPU 的弱一致性存储模型增加了原子操作 (线程同步重要原语之一) 的访存开销, 进一步导致线程同步难以高效表达.

(2) 错误频发. 传统 CPU 多核处理器并行编程中的线程同步即存在错误频发的问题, 而 GPU 采用 SIMT 的执行模式和非抢占式的线程块 (thread block) 调度方式, 使得线程同步更易引入与并行执行相关的错误, 例如死锁 (deadlock)、活锁 (livelock) 等.

(3) 执行效率低. 线程同步在 GPU 中极易引入分支执行 (branch divergence), 并造成不同线程组织单元的执行进度不同. GPU 并行规模巨大, 使得上述因素给线程同步的性能带来极大影响, 并造成硬件资源的严重浪费.

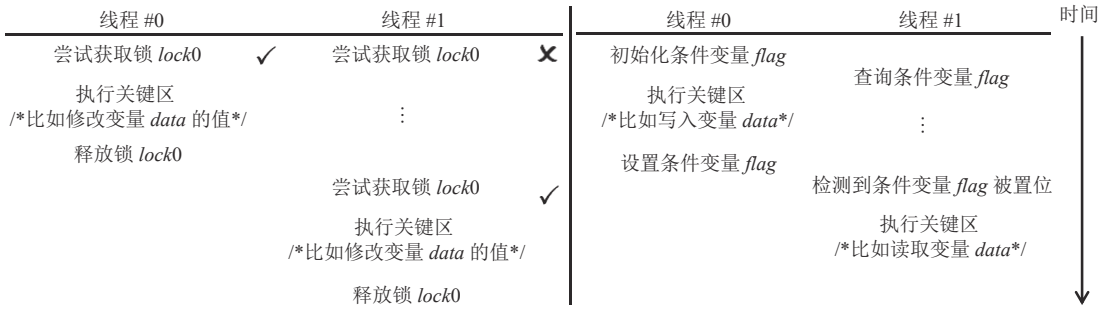
GPU 线程同步存在的上述问题极大限制了 GPU 应用范围的扩展和应用性能的进一步提升. 在 GPU 通用计算发展初期, 简单的线程同步原语尚可满足大多数应用的线程同步需求. 然而, GPU 通用计算发展到现在, 应用范围日益扩大, 越来越多的应用需要进行频繁的线程同步, 且同步模式多变而复杂, 极其需要高效的线程同步机制的支持, 例如基于哈希表/树/图的应用、在线事务处理 (online transaction processing) 等数据库应用. 另外, 虽然很多应用已经在 GPU 通用计算中取得了较好的加速效果, 但 GPU 同步机制的不完善却严重制约这些应用性能的进一步提高, 例如应用于广告推荐、语音识别、图像识别、语言翻译、药物研发、自动驾驶等领域的深度学习应用.

基于以上原因, 面向 GPU 并行编程的线程同步研究受到国内外学术界和工业界的广泛关注. 国内外学术界和工业界提出了一些改善 GPU 线程同步的方法和技术, 例如简化 GPU 线程同步表达的编译技术[1]、GPU 动态并行技术 (dynamic parallelism, 允许 GPU 函数嵌套启动和执行)[2−5]、GPU 互斥锁机制[6−12]等. 这些方法虽然从某些角度或局部改善了 GPU 线程同步存在的问题, 然而从总体上看, GPU 线程同步仍然有很多问题亟待解决, 而且随着 GPU 通用计算的发展和普及, 这些问题日益严峻. 本文对 GPU 并行编程中的线程同步进行了分类; 围绕线程同步的表达和执行, 首先分析总结 GPU 线程同步面临的问题和挑战, 而后依据不同的 GPU 线程同步粒度, 对现有 GPU 线程同步的表达方法和性能优化方法进行详细介绍, 剖析国内外 GPU 并行编程中线程同步的研究现状; 基于这些讨论, 总结展望 GPU 线程同步未来的研究趋势和发展前景.

本文的第 1 节对 GPU 并行编程中的线程同步进行分类. 第 2 节对 GPU 线程同步所面临的问题和挑战进行详细阐释和分析. 第 3 节和第 4 节分别对当前国内外 GPU 线程同步的表达和执行的研究现状进行综述. 在此基础上, 第 5 节对 GPU 并行编程中线程同步的研究趋势进行展望.

1 GPU 并行编程中线程同步分类

根据不同的线程同步目的, 可将线程同步分为线程竞争同步和线程合作同步两类. 如图 1 所示, 线程竞争同步保证多个线程互斥地访问共享数据, 从而保证多线程程序的访存一致性, 强调访问共享数据时的互斥性; 而线程合作同步在相互协同的多个线程之间保证对共享数据先写后读的顺序, 强调访问共享数据时的顺序性. 在并行编程中, 线程竞争同步主要通过原子操作及基于原子操作的互斥锁 (mutex) 实现; 而线程合作同步一般通过条件变量 (conditional variable)、信号量 (semaphore)、栅栏 (barrier) 实现. 图 1 及后文相关图中对号表示锁获取成功, 叉号表示锁获取失败.



(a) 线程竞争同步示例图 (b) 线程合作同步示例图

图 1 线程同步分类

与传统并行编程不同, GPU 编程模型将线程划分为 4 个层次: 最底层为 GPU 线程 (thread); 较高一层为线程束 (warp/wavefront), 由一定数量的 GPU 线程组成 (通常为 32 个); 多个线程束进一步组成线程块; 多个线程块进一步组成网格 (grid). 其中, 各层线程组织单元的执行和调度方式各异, 对线程竞争/合作同步的支持也不尽相同. 因此, 本文将 GPU 并行编程中的线程同步划分为线程、线程束、线程块级别的同步.

综上所述, 本文将 GPU 并行编程中的线程同步分为 6 类, 所分类别及详细说明如表 1 所示. 在竞争同步中, GPU 的线程层次指共享数据的互斥访问粒度: 线程级别的竞争同步是以线程为粒度对共享数据进行互斥访问; 而线程束和线程块级别的竞争同步则分别以线程束和线程块为粒度对共享数据进行互斥访问, 属于线程级别竞争同步的特例. 在合作同步中, GPU 的线程层次则表明合作同步主体的范围: 线程级别合作同步的主体为线程, 协同范围为线程束; 线程束级别合作同步的主体为线程束 (包括线程束内所有线程), 协同范围为线程块; 线程块级别合作同步的主体为线程块 (包括线程块内的所有线程), 协同范围为 GPU 函数, 因而也可称为全局合作同步.

表 1 线程同步分类说明

级别	竞争同步	合作同步
线程级别	以线程为粒度互斥地访问共享数据	线程束内的线程间进行协同
线程束级别	以线程束为粒度互斥地访问共享数据	线程块内的线程束间进行协同
线程块级别	以线程块为粒度互斥地访问共享数据	线程块间进行协同

2 GPU 线程同步面临的问题和挑战

GPU 的体系结构特点导致线程同步存在难以高效表达、错误频发、执行效率低的问题. 本节根据 GPU 线程同步的分类, 围绕线程同步的表达和执行, 进一步详细分析并总结了 GPU 并行编程中线程同步所面临的问题和挑战, 如表 2 所示.

具体而言, GPU 并行编程中线程同步的表达主要面临两个问题和挑战: 1) 线程级别竞争同步及线程块级别合作同步的表达所产生的同步开销大; 2) 线程同步的表达极易出现与并行执行相关的错误, 包括线程级别竞争同步

导致的死锁、活锁问题, 以及线程块级别合作同步导致的死锁问题. 在线程同步的执行方面, 竞争同步在执行过程中频繁出现同步失败, 线程束/线程块级别合作同步在执行过程中存在同步主体的长时间等待, 这些严重降低线程同步在 GPU 中的执行效率.

表 2　GPU 并行编程中线程同步面临的问题和挑战

<table>
<tr><th rowspan="2">级别</th><th colspan="2">竞争同步</th><th colspan="2">合作同步</th></tr>
<tr><th>表达</th><th>执行</th><th>表达</th><th>执行</th></tr>
<tr><td>线程级别</td><td rowspan="3">死锁、活锁问题
(SIMT)
同步开销大
(原子操作开销大)</td><td rowspan="3">同步失败频繁
(GPU并行规模大、
分支执行多)</td><td>无开销
(SIMT)</td><td>执行效率高
(SIMT)</td></tr>
<tr><td>线程束级别</td><td rowspan="2">线程级别竞争同步的特例</td><td>开销小
(GPU硬件提供轻量级栅栏)</td><td rowspan="2">同步等待时间长
(调度机制简单)</td></tr>
<tr><td>线程块级别</td><td>死锁
(线程块非抢占式调度)
同步开销大
(无硬件支持、原子操作开销大)</td></tr>
</table>

2.1　线程同步的表达在 GPU 中的问题和挑战

(1) 线程同步开销巨大

原子操作是线程同步的重要原语之一, 通过实现共享数据访问的原子性, 可保证多个线程对共享数据操作的互斥性, 并完成多线程间的同步通信, 在 GPU 竞争同步、线程块级别合作同步的表达中频繁使用. 现有 GPU 在硬件和运行时层面提供了很多原子操作, 比如, NVDIA CUDA 编程模型[13]提供了全面的原子操作: atomicAdd、atomicSub、atomicExch、atomicMin、atomicMax、atomicInc、atomicDec、atomicCAS、atomicAnd、atomicOr、atomicXor. 这些原子操作对单独的 32 位或 64 位数据以原子性的方式进行操作, 可以实现简单的线程间同步通信. 然而, 与大多数 CPU 体系架构不同, GPU 采用弱一致性存储模型, 使得原子操作的执行引入较大访存开销, 导致 GPU 中线程同步开销增大. 而 GPU 巨大的并行规模又会极大增加原子操作的数量, 进一步加剧了 GPU 线程同步的开销.

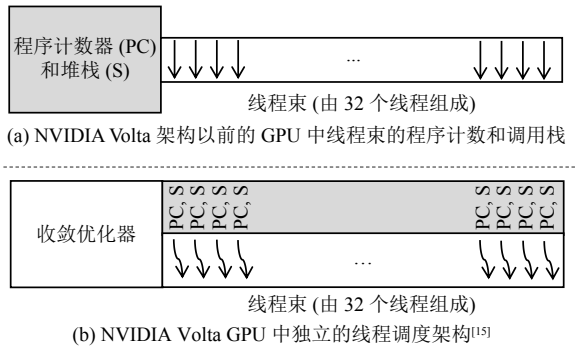
因而, 降低 GPU 中原子操作的访存开销或减少线程同步表达中原子操作的使用, 从而降低线程同步的开销, 是 GPU 线程同步面临的一个挑战.

(2) 死锁与活锁问题频繁

在 GPU 并行编程中, 线程级别竞争同步极易导致死锁、活锁问题, 而线程块级别合作同步极易出现死锁问题[8,9,14].

● 线程级别竞争同步的死锁及活锁问题

目前, 绝大多数 GPU (比如, NVIDIA 所有 Volta[15]架构以前的 GPU) 采用 SIMT 的执行方式, 以支持程序的大规模并行. 如图 2(a) 所示, 在这些 GPU 中, 线程束内的线程共享程序计数器和堆栈等调度资源, 并以锁步的方式执行程序. 这种执行方式导致线程级别竞争同步极易出现死锁及活锁问题.



(a) NVIDIA Volta 架构以前的 GPU 中线程束的程序计数和调用栈

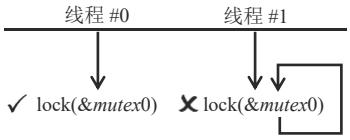
(b) NVIDIA Volta GPU 中独立的线程调度架构[15]

图 2　线程级别竞争同步的死锁及活锁问题

本文以竞争同步通用的表达方法互斥锁为例, 阐述 SIMT 执行方式如何导致线程级别竞争同步出现死锁及活锁. 图 3(a) 采用自旋锁[16,17]的方式在 GPU 上实现互斥锁. 如图 3(b) 所示, 同一线程束内的线程 #0 和 #1 竞争同一个锁 *mutex*0, 其中线程 #0 成功获取锁. 由于同一线程束内的线程采用锁步的方式执行程序, 线程 #0 在成功获取锁之后, 需要等待线程 #1 成功获取 *mutex*0, 才能汇合并执行后续代码. 此时, 线程 #0 因无法继续执行导致无法释放 *mutex*0, 而这使得线程 #1 永远无法获得该锁, 并进一步导致线程 #0 永远无法继续执行, 从而出现死锁.

```
bool lock(addr){
    if (atomicCAS(addr, 0, 1) == 1) {return false;}
    else {return true;}
}
void unlock(addr){
    atomicExch(addr, 0);
}
// 自旋锁
while(!lock(addr)){
    critical section
    unlock(addr);
}
```

(a) 自旋锁[16,17]



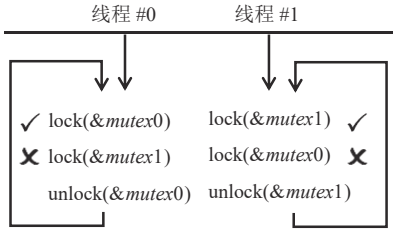
(b) 死锁示例图

图 3 死锁

为避免死锁, 可以使用非阻塞锁[16,17]. 如图 4(a) 所示, 在非阻塞锁中, 当一个线程无法获取某个锁时, 它不会轮询, 而是首先释放掉已经成功获取的锁, 等待其他线程执行完关键区域并释放锁之后, 再尝试获取该锁. 这种方式虽然可以避免死锁, 但当同一线程束内的不同线程以相反的顺序获取相同系列锁的时候, 则会由于在线程束内循环获取锁而导致活锁的出现. 以图 4(b) 为例, 同一线程束内的线程 #0 和 #1 以相反的顺序竞争锁 *mutex*0 和 *mutex*1. 开始时, 两个线程分别成功获取 *mutex*0 和 *mutex*1. 而后, 因为所需的锁已被对方获取, 线程 #0 和 #1 均无法成功获取锁. 此时, 二者会释放已经获得的锁 (线程 #0 释放 *mutex*0, 线程 #1 释放 *mutex*1) 并重新进行尝试. 由于同一线程束内的线程采用锁步的方式执行程序, 线程 #0 和 #1 陷入无限循环, 导致活锁. 在实际应用中, 由非阻塞锁引起的活锁问题非常普遍[8,9].

```
bool lock(addr){
    if (atomicCAS(addr, 0, 1) == 1) {return  false;}
    else {return true;}
}
void unlock(addr){
    atomicExch(addr, 0);
}
// 非阻塞锁
done = false;
while (done == false){
    if (lock(addr0)){
        if(lock(addr1)){
            critical section
            unlock(addr0);
            unlock(addr1) ;
            done = true;
        }else{unlock(addr0);}
    }
}
```

(a) 非阻塞锁[16,17]



(b) 活锁示例图

图 4 活锁

与死锁导致线程永远等待的情况不同, 当出现活锁时, 线程仍会不停地执行指令, 但却无法向前执行. 目前, 部分 GPU 为线程束内的每个线程设计了独立的程序计数器和堆栈等调度资源, 比如图 2(b) 中 NVIDIA 的 Volta GPU[15].

然而, 为高效支持大规模线程的并发执行, 在这些 GPU 中, 线程束内的线程在绝大多数情况下仍然采用 SIMT 的执行方式 (比如, 只在出现分支执行时, 这些 GPU 才会令线程束内的线程独立执行指令). 因此, 对于这些 GPU, 虽然能够避免死锁问题 (如图 3(b) 所示, 死锁发生时会出现分支执行), 但仍然会出现活锁 (如图 4(b) 所示, 活锁发生时线程 #0 和 #1 均会进行锁的释放, 而不会出现分支执行).

● 线程块级别合作同步的死锁问题

GPU 采取非抢占式的线程块调度方式, 线程块只有当执行完后, 才会释放硬件资源 (流多处理器, streaming multiprocessors, SM). 因而, 当线程块规模超过 GPU 硬件资源容量时, 这种调度方式会导致线程块间合作同步出现死锁.

如图 5 所示, 线程块 #0 到 #24 在同步点 *barrier*0 进行合作同步, 即执行到同步点 *barrier*0 时需互相等待, 直到所有线程块到达 *barrier*0 后才能继续执行. 由于 GPU 硬件资源容量的限制, 线程块 #0 到 #23 被调入到 GPU 中执行, 而线程块 #24 则需等待部分已调入的线程块执行完并释放硬件资源后, 才能调入到 GPU 中开始执行. 此时, 线程块 #0 到 #23 执行到同步点 *barrier*0 时会进行等待 (等待线程块 #24 执行到同步点 *barrier*0), 无法释放计算资源; 而线程块 #24 则等待计算资源空闲而无法执行, 二者相互等待, 出现死锁.

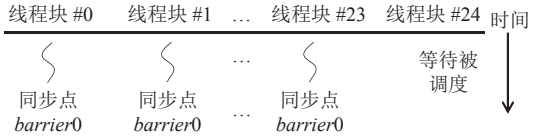


图 5 线程块级别合作同步导致的死锁示例图

综上所述, 线程级别竞争同步引入的死锁和活锁问题, 以及线程块级别合作同步引入的死锁问题极大地增加了 GPU 并行编程中线程同步表达的难度, 如何避免这些问题以保证线程同步的正确表达, 是 GPU 线程同步面临的又一个挑战.

2.2 线程同步的执行在 GPU 中的问题和挑战

线程同步在 GPU 中执行效率低. 线程同步的执行效率主要体现在程序性能和对硬件的使用效率两个方面. 线程同步对实际执行过程中的并行规模、共享数据冲突程度、线程调度等因素影响很大, 易导致程序执行效率低 (程序性能差、硬件利用率低). 而 GPU 线程组织结构复杂、并行规模巨大、调度机制及调度策略简单, 会进一步恶化线程同步的执行效率. 具体而言, GPU 并行编程中造成线程同步执行效率低的因素主要包括以下两点.

(1) 竞争同步中频繁的同步失败

为提升应用性能, 程序多采用忙等待 (busy-waiting) 的竞争同步方式 (比如前文中图 3(a) 和图 4(a)). 然而, 这种同步方式在程序并行规模过大时, 会导致较高的同步失败频率[18,19]. 在 GPU 并行编程中, 数以万计的线程规模更易导致频繁的同步失败. 而同步失败会增加大量无用的内存访问, 引入额外的执行开销. 在 GPU 并行编程中, 为充分利用 GPU 的大规模并行度, 应用往往采用细粒度线程同步对共享数据进行互斥访问, 比如, 为每个共享数据都分配一个互斥锁. 这种同步方式导致与传统的多核/众核 CPU 并行系统相比, GPU 中线程同步的失败引入的执行开销更大, 对执行效率的影响也更明显.

首先, 细粒度的线程同步方式导致线程同步失败时部分工作可能需要重新执行, 从而引入额外开销. 以 BarnesHut 程序[20] (一种典型的 *n*-body 算法) 为例, 在其构建树 (TreeBuilding 函数) 的过程中需要加锁. 而当锁获取失败时, 必须先重新对树进行遍历并确定数据需要插入的树分支, 才能再次尝试进行加锁. 如图 6 所示, 这部分需要重新执行的代码会引入额外的开销.

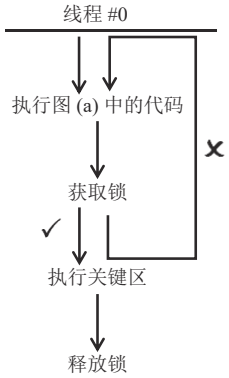
其次, 细粒度的线程同步方式导致在进入关键区域之前, 需要执行更多的同步操作, 这使得线程同步的失败引入更多同步操作的执行, 也影响应用的实际并行度, 从而引入更大的开销, 严重影响线程同步执行效率. 以典型的在线事务处理应用测试程序 SmallBank[21]为例, 图 7(a) 为 SmallBank 应用中联合 (amalgamate) 事务的代码. 在该事务中, 关键区域由 4 个嵌套锁保护. 当其中任一锁获取失败时, 其他成功获取的锁则不再需要, 且应被释放掉 (比

如, 图 7(b) 中, 线程 #0 获取的 *user*#0 储蓄账户和支票账户对应的互斥锁). 另外, 上述这些锁在被释放之前, 还会导致其他想要获取这些锁的线程无法成功获得该锁, 从而影响应用的实际并行度. 比如, 如图 7(b) 所示, *user*#0 储蓄账户对应的互斥锁对线程 #0 来说没有用处, 还导致线程 #1 同步的失败.

竞争同步失败导致的需要重新执行的代码片段

```
// 重新遍历树结构
ch = childd[n* 8+j];
while (ch >= nbodiesd ) {
   n = ch;
   depth++ ;
   r *= 0.5f;
   dx = dy = dz = −r;
   j = 0;
   // 重新判断树的具体分支
   if (x < p.x) {j = 1; dx = r;}
   if (y < p.y) {j = 2; dy = r;}
   if (z < p.z) {j = 4; dz = r;}
   x += dx;
   y += dy;
   z += dz;
   ch = childd[n*8+j];
}
```

(a) BarnesHut 中锁获取失败时需要重新执行的代码



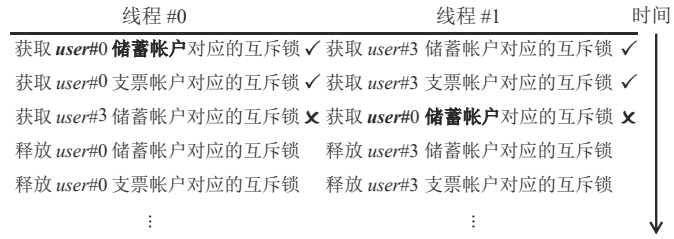
(b) 代码重新执行示例图

图 6 线程同步失败引入的额外执行开销

SmallBank 中联合事务的代码片段

```
done=false;
while(!done){
  // 执行联合事务前，需要获取 4 个锁
  if(lock (acc_0_s)){
    if(lock (acc_0_c)){
      if(lock (acc_1_s)){
        if(lock (acc_1_c)){
          // 执行联合事务
          total = s_0 + c_1;
          c_0 = 0;
          s_1 += total;
          done=true;
          /*当 1.事务执行完;
            或 2. 有锁获取失败时, 释放掉已获取的锁*/
          release_lock(acc_0_s, acc_0_c, acc_1_s, acc_1_c);
        }else{release_lock(acc_0_s, acc_0_c, acc_1_s);}
      }else{release_lock(acc_0_s, acc_0_c); }
    }else{release_lock(acc_0_s);}
  }
}
```

(a) SmallBank 中联合事务的代码片段



(b) 线程同步失败影响应用实际并行度示例图

图 7 线程同步失败导致的应用实际并行度降低

另外, 在 GPU 并行编程中, 线程级别竞争同步的失败还会引入频繁的分支执行, 进一步严重影响程序性能, 并造成硬件资源浪费.

(2) 合作同步中同步主体 (线程束/线程块) 的长时间等待

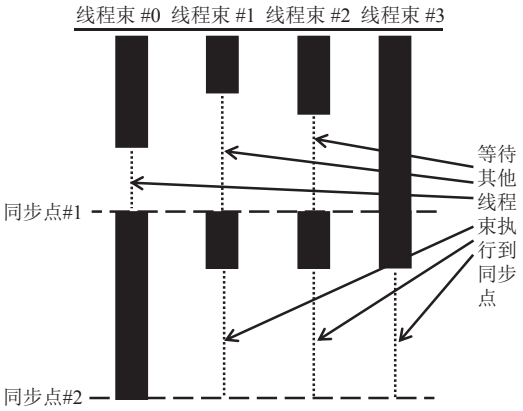
在 GPU 并行编程中, 线程束和线程块级别的合作同步在执行过程中, 进行合作的同步主体执行进度不一致, 会导致较早执行到同步点的主体长时间等待, 致使没有足够的并行线程掩盖访存时延, 影响程序性能并造成硬件资源浪费. 以著名的 Rodinia 测试程序集[22]中的 B+Tree 程序为例, 如图 8(a) 所示, 线程束在同步点#1 和同步点#2 会通过 CUDA 编程模型提供的同步原语__syncthreads() 进行合作同步. 此时, 若各线程束的执行进度不一致, 则会导致先到达同步点的线程束等待执行进度慢的线程束, 比如图 8(b) 中线程束#0 到#2 在执行到同步点#1 时需要等

待线程束#3, 线程束#1 到#3 在执行到同步点#2 时需要等待线程束#0, 影响程序的并行度, 并造成硬件资源的浪费. 与 CPU 并行系统不同, GPU 的硬件线程调度逻辑相对简单, 且无操作系统等软件的支持, 上述同步主体长时间等待的情况非常普遍[23], 对应用性能的影响也更明显.

B+Tree 中线程合作同步代码片段

```
int thid = threadIdx.x, bid = blockIdx.x, i;
for(i = 0; i < height; i++){
  // 要查找的键值位于两个键中间
  if((knodesD[currKnodeD[bid]].keys[thid]) <= keysD[bid] &&
   (knodesD[currKnodeD[bid]].keys[thid+1] > keysD[bid])){
    if(knodesD[offsetD[bid]].indices[thid] < knodes_elem ){
      offsetD[bid] = knodesD[offsetD[bid]].indices[thid ];
    }
  }
  __syncthreads(); // 同步点#1
  if(thid == 0){
    currKnodeD[bid] = offsetD[bid];
  }
  __syncthreads(); // 同步点#2
}
```

(a) B+Tree[22]中线程合作同步代码片段



(b) 线程同步中同步主体等待示例图

图 8 B+Tree 程序示例

综上所述, 竞争同步中频繁的同步失败以及合作同步中同步主体的长时间等待严重影响 GPU 线程同步的执行效率, 是 GPU 线程同步面临的挑战之一.

3 GPU 线程同步表达方法研究

在分析总结 GPU 线程同步中存在的问题和挑战的基础上, 本节围绕线程同步的表达, 分别对 GPU 并行编程中竞争同步和合作同步的表达方法及其研究进程进行综述分析.

3.1 竞争同步的表达

如表 3 所示, 根据表达竞争同步所采用技术的不同, 本文将现有 GPU 竞争同步表达方法可以划分为原子操作、互斥锁、事务内存、并行算法, 以及代码生成 5 种不同类别.

表 3 GPU 竞争同步表达方法汇总

竞争同步表达方法		应用范围	额外硬件支持	时间开销	实现复杂性
原子操作[13,24]		小	否	中	低
互斥锁	自旋锁及非阻塞锁[16,17]	小	否	中	中
	串行化自旋锁[25]	大	否	高	中
	硬件锁[12]	大	是	低	中
	其他加锁策略[7−11]	大	否	中	中
事务内存	硬件事务内存[17,26−31]	大	是	低	低
	软件事务内存[32−36]	大	否	高	低
并行算法[37−40]		小	否	低	高
代码生成[1]		大	是	低	中

(1) 原子操作

原子操作是最简单的竞争同步实现方式. 比如, 两个并发线程对共享数据 a 进行操作: 假设 a 的初始值为 0, 两个线程分别对其进行加 1 操作. 该操作包括 1) 从内存读取 a 的原始值; 2) 对读取的值进行加 1 操作; 3) 将加 1 之后的值写回内存 3 个步骤. 如果两个线程没有进行互斥访问, 则可能出现两个线程同时从内存读取 a 的原始值

0, 并分别进行后续两个步骤的情况. 此时, a 的最终结果为 1, 而并非期望得到的结果 2. 对于这种情况, 两个线程可以通过原子操作的方式实现对共享数据 a 的互斥访问.

如本文第 2.1 节所述, GPU 在硬件和运行时层面提供了很多原子操作[13,24], 但原子操作开销较大. 针对该问题, 文献 [41−45] 对 GPU 的存储模型进行探索, 通过提出新的缓存一致性协议或内存模型, 减少原子操作的访存开销. 除此之外, NVIDIA 的大多数 GPU 在每个 SM 中分配一定数量的共享内存, 并在其较新的 Hopper[46]架构 GPU 中进一步设计线程块集群 (thread block cluster), 在多个 SM 间分配共享内存, 以支持 SM 内部以及线程块集群内部 SM 间原子操作的快速执行, 减少原子操作引入的访存开销.

另外, 原子操作中操作对象的数据类型受限, 操作类型单一, 极大地局限了其应用范围 (只能保证线程对单个共享地址的操作的原子性), 使得原子操作无法直接实现复杂的竞争同步. 比如, 当每个线程需要访问多个共享地址, 并且需要保证这些地址上多个操作的原子性时, 无法使用原子操作.

(2) 互斥锁

互斥锁是一种通用的竞争同步表达方式, 它通过原子操作构造, 可以支持更加复杂的、对共享数据的互斥访问. 在互斥锁中, 共享数据或者共享区域由互斥锁保护, 只有当线程成功获得该互斥锁时才能对共享数据或者共享区域进行访问. 互斥锁可以由程序员显式实现 (如图 7(a) 所示), 也可以在编程语言中实现. 由于互斥锁具有实现简单、程序效率高等优点, 因而在 CPU 程序中应用广泛.

然而, 如本文第 2.2 节所述, GPU 并行编程中, 线程级别互斥锁极易出现死锁、活锁问题 (比如表 3 的自旋锁及非阻塞锁). 为避免上述错误, 文献 [25] 设计在相同线程束内, 线程以串行方式获取自旋锁. 这种方式虽然可以避免死锁及活锁, 但线程束内线程串行获取锁将导致 GPU 执行单元的利用率大幅下降, 影响程序性能并造成硬件资源浪费.

针对上述问题, Yilmazer 等人[12]提出为 GPU 体系结构上的互斥锁提供硬件支持, 称为 HQL. HQL 利用 GPU 中线程快速切换的特点以及缓存的结构特征, 在非一致性的存储结构中设计并实现了一个多层次队列, 以在 GPU 中支持实现互斥锁. HQL 可以避免死锁、活锁的出现, 但对 GPU 硬件的修改限制了其使用范围. 除此之外, 由于硬件开销的限制, HQL 中锁条目的数量有限, 使得程序并发度受限, 对程序性能造成影响.

与硬件锁的设计思路不同, 文献 [8,9] 在非阻塞锁的基础上, 设计锁窃取算法, 采用纯软件的方式避免活锁. 他们重新设计了锁的数据结构, 除记录锁是否被占用外, 还对占用锁的线程进行记录. 在此基础上, 他们分析活锁可能出现的情况, 设计令可能导致活锁的线程去窃取其他线程占用的锁, 并总结锁窃取的 3 个必要条件, 确保能够通过锁的窃取避免活锁, 且不会引入新的活锁问题. 锁窃取能够保证在商用 GPU 中正确的表达互斥锁, 但该研究仍存在一些问题, 比如活锁的检测和锁的窃取会引入额外的原子操作; 活锁的检测存在错检的可能, 导致不必要的锁窃取, 这些都会造成锁获取开销的增大.

以上研究专注于 GPU 线程级别竞争同步的正确表达, 却难以避免其中的原子操作在 GPU 中引入的较大访存开销. 针对该问题, Wang 等人[7]提出采用分布式方法进行锁的获取和释放, 设计线程块间通信方法, 将所有锁获取及释放操作交与部分线程块执行. 通过这种方法, 他们将以往存储在速度较慢的全局内存中的锁迁移到快速的共享内存中, 从而避免互斥锁中原子操作引入的较大访存开销. 他们的研究能够减少互斥锁的开销, 然而, 这种分布式的编程方法通用性较差, 编程难度也较高.

(3) 事务内存

事务内存编程方法通过事务 (transaction) 的操作方式实现对共享数据的互斥访问, 具有编程难度低、程序错误少、编程效率高等特点. 如图 9 所示, 程序员在利用事务内存系统编写并行程序时, 只需在线程中使用事务内存将需要并行执行的临界区指定为事务, 临界区的正确并行执行由事务内存底层机制保证, 而不再需要程序员关心. 在事务内存系统中, 每个事务的所有指令在程序执行过程中表现出原子性: 如果事务所读的共享数据在事务执行过程中没有被其他任何并发事务更新, 事务提交 (commit), 提交完成以后事务对共享数据的更新对其他事务可见; 否则事务与其他并发事务冲突, 事务撤销, 事务不对共享数据进行任何更新.

事务内存系统最初在通用 CPU 处理器体系结构中被广泛研究并应用. 受到 CPU 事务内存系统的启发, 研究

者近年来尝试在 GPU 上实现事务内存系统, 以在 GPU 中实现竞争同步. 然而, GPU 体系结构与 CPU 体系结构差别较大, 传统的事务内存系统设计或无法直接应用于 GPU 体系结构, 或实现开销大[47]. 因此, 文献 [32−36] 在 GPU 中提出软件事务内存系统. 其中, Holey 等人[33]通过适当放开冲突验证的精度, 从而减少冲突检测的开销, 并在 GPU 中提出了一个轻量级的事务内存方法. 然而, 他们提出的设计没有解决在事务内存系统中使用互斥锁时可能遇到的死锁、活锁问题, 也不能保证事务执行的正确性, 且无法满足不透明性 (opacity), 这可能导致事务在执行过程中陷入无限循环、非法地址访问或其他运行时错误. Xu 等人[35]为支持 GPU 体系结构中的大规模并发线程, 提出了一个多层次事务验证机制, 并提出在事务执行过程中对锁进行排序, 以避免活锁.

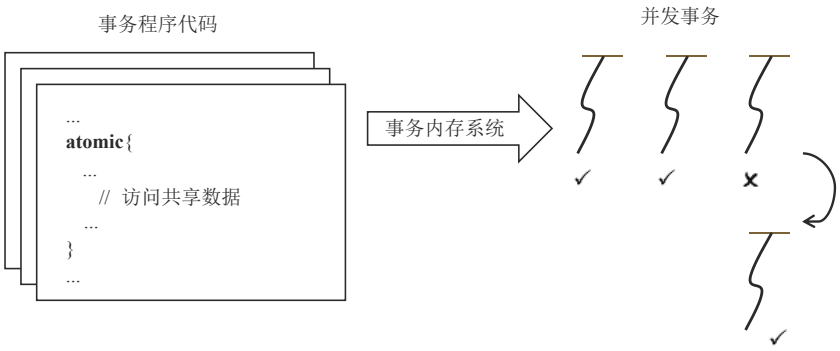


图 9 事务内存编程的原理

考虑到 GPU 软件事务内存会引入较大开销, 文献 [17,26−31] 在 GPU 中设计实现硬件事务内存系统. Fung 等人[17,30]提出 GPU 硬件事务内存系统 KILO TM, 将事务日志记录在片外全局存储器 (global memory) 中, 并使用高速缓存降低日志的访问延迟. 与传统的利用缓存一致性进行冲突检测的 CPU 硬件事务内存不同, 由于 GPU 没有缓存一致性的支持, KILO TM 使用基于值的冲突检测 (value-based conflict detection) 方法, 在 GPU 现有硬件中添加若干提交单元 (commit unit) 进行冲突检测. 为保证冲突检测和内存更新操作的正确性, 访问同一个全局存储器分区的所有事务需要串行进行冲突检测和提交, 因此事务验证和提交时的并发度较低, 影响程序性能. Villegas 等人[31]也提出了一种 GPU 硬件事务内存系统. 与 KILO TM 不同的是该事务内存系统只保证并发事务访问共享存储器中 (而不是全局存储器) 共享数据的互斥性. 在额外硬件的支持下, 硬件事务内存开销较小. 然而, 对硬件进行修改限制了其使用范围.

(4) 并行算法

考虑到在 GPU 中实现竞争同步存在的诸多问题和挑战 (详见第 2.1 节和第 2.2 节), 部分研究者针对特定应用设计并行算法[37−40], 以期能够避免在 GPU 中对共享数据进行互斥访问. 这些并行算法有一个共同模式: 首先识别没有冲突的工作集, 然后将识别出的无冲突的工作集利用 GPU 并发线程并行处理.

Narse 等人[39]针对 Delaunay 网格细化 (Delaunay mesh refinement) 应用, 提出了一种冲突避免方法. 在三角细化过程中每个线程需要处理一个不满足特定要求的坏三角, 如果两个线程处理的坏三角相邻, 则这两个线程之间有数据竞争, 导致程序不能产生正确的结果. 在 Narse 等人提出的三阶段冲突避免方法中, 每个 GPU 线程先标记自己需要处理的三角形及其相邻三角形, 然后检查所标记的三角形是否与其他线程标记的三角形重合, 如果不重合则这个线程可以继续三角细化执行, 否则不能执行; He 等人[40]针对在线事务处理应用, 通过对事务执行顺序进行排序来构建依赖图, 以标记事务间的冲突关系, 并在程序执行过程中通过查询依赖图避免冲突访问.

通过设计并行算法可以在 GPU 中实现复杂的竞争同步, 对特定算法会产生较好的加速效果, 具有执行效率高的优点. 然而, 这些并行算法不具有通用性, 对每个特定的应用需要设计不同的并行算法以识别出无冲突的、能够并行执行的操作, 编程难度极高. 另外, 这种方法还有另外一个缺点: 在识别和处理无冲突的工作集合两个阶段之间, 需要使用一个额外的全局同步栅栏[48], 以确保在所有 GPU 线程中, 处理阶段都在识别阶段之后进行. 而同步栅栏的使用会降低 GPU 线程的并发度, 影响应用性能.

(5) 代码生成

为了能够更便捷的在 GPU 上进行扩展编程, ElTantawy 等人[1]从代码生成的角度入手, 提出通过静态监测程序控制流图 (control flow graph, CFG) 的方法检测因 SIMT 执行方式导致的死锁及活锁问题, 并设计 CFG 转换算法予以避免. 他们首先对程序常见的死锁、活锁相关代码进行分析总结, 而后在编译层面对程序的 CFG 进行监测识别, 找出可能导致死锁及活锁的部分; 在此基础上, 进一步分析避免死锁及活锁的方法, 即延迟分支代码汇合执行, 并据此对 CFG 的相应部分进行转换, 从而实现程序的正确执行.

通过代码生成的方法避免死锁及活锁问题无需对程序代码进行修改, 能够节省大量的调试工作, 无任何编程难度, 且具有很强的通用性. 然而, 编译过程的限制使得这种方式存在错检的可能, 对程序性能影响较大. 因此, 上述研究工作在编译层面生成代码的基础上, 进一步对 GPU 的硬件进行了修改, 支持 CFG 转换部分的快速执行, 从而降低错检对程序性能造成的影响. 然而, 对 GPU 硬件的修改限制了其使用范围.

综上所述, 已有研究能够确保 GPU 中竞争同步表达的正确性. 然而, 如表 3 所示, 部分方法通用性较低、灵活性较差, 特别是会引入较大的同步开销. 比如, 大多数 GPU 互斥锁和事务内存的研究都以较大的执行开销为代价来避免死锁、活锁.

3.2 合作同步的表达

如表 4 所示, 根据表达合作同步所采用技术的不同, 本文将已有 GPU 合作同步表达方法可划分为栅栏、线程同步原语、编程模型、并行算法 4 种不同类别. 除此以外, 本文对线程块级别合作同步中避免死锁的相关研究进行综述分析.

表 4 GPU 合作同步表达方法汇总

合作同步表达方法		同步级别	应用范围	额外硬件支持	时间开销	灵活性
栅栏	软件栅栏[48−50]	线程束/线程块	大	否	中	高
	硬件栅栏[51]		大	是	低	高
线程同步原语	__syncthreads()[13]	线程束	大	否	低	中
	cudaLaunchCooperativeKernel[13]	线程块	大	否	高	低
	Cooperative Groups[52]	线程/线程束/线程块	大	否	中	中
编程模型[2−5,53−55]		线程块	小	否	低	低
并行算法[56−59]			小	否	低	中

(1) 栅栏

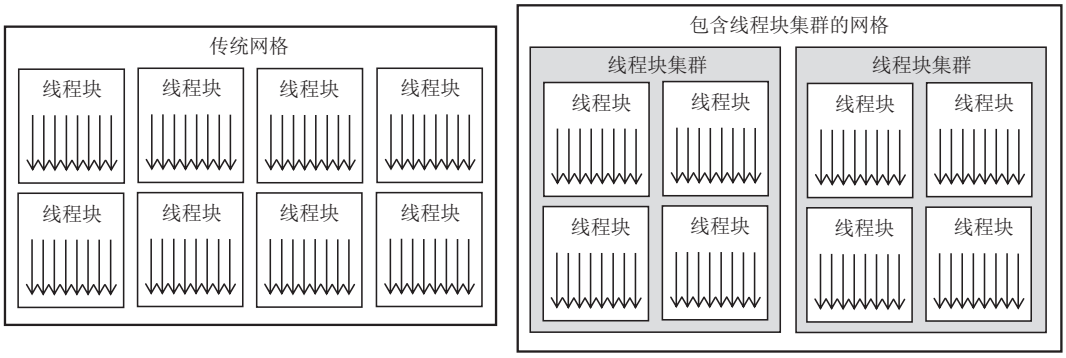
栅栏是一种常用的合作同步实现方法, 主要包括同步到达和同步等待两类语句. 两者均表明同步主体已到达该同步点. 其中, 同步到达语句不会阻塞同步主体的执行; 而同步等待语句则不同, 需要等到所有同步主体阻塞或到达该同步点时, 才会令同步主体继续执行. 通过同步到达和同步等待语句的使用, 栅栏能够实现线程间的合作同步. 如图 10 所示, 程序员可以利用原子操作显式的实现栅栏.

```
1: int synct_counter; //统计到达同步点的同步主体数量
2: void sync_arrive() //同步到达
3:     atomicAdd(&synct_counter, 1);
4: void sync_wait() //同步等待
5:     atomicAdd(&synct_counter, 1);
6:     while(atomicCAS(&synct_counter, SYN_NUM, 0) != SYN_NUM)
7:     //SYN_NUM 为进行合作同步的同步主体数量
```

图 10 利用原子操作实现栅栏

为支持高效的线程合作同步, 通用 CPU 处理器提供了若干硬件栅栏. 基于相同目的, 近年来 GPU 也逐渐在其硬件中增加硬件栅栏. 以 NVIDIA GPU 为例, 其近年来为每个线程块提供了有限数量的硬件栅栏 (16 个), 以高效

的支持线程束级别合作同步, 并在 PTX (parallel thread execution) 指令中提供了若干同步原语. 其中, 最主要的同步原语为 bar.sync 和 bar.arrive[60]. 前者属于同步等待语句, 实现了阻塞操作, 使得某一线程束执行到 bar.sync 处开始等待, 直到一定条件满足以后才会继续执行 (比如, 只有当 n 个线程束等待在 bar.sync 处或执行到 bar.arrive 时, 当前线程束才会继续执行). 后者属于同步到达语句, 是一个非阻塞操作, 用于指示当前线程束已经执行到 bar.arrive 处. 利用这两个同步原语, 文献 [18,61,62] 在 GPU 线程块内的多个线程束之间实现了生产者-消费者的数据处理模式. 除此以外, 如图 11 所示, NVIDIA 在其较新的 Hopper[46]架构 GPU 中设计了线程块集群 (thread block cluster), 并在每个线程块集群中增加硬件栅栏, 以高效地支持线程块集群内部线程块间的合作同步.



(a) 传统 GPU 网格示例图 (b) NVIDIA Hopper 架构 GPU 包含线程块集群的网格示例图[46]

图 11 网格示例

在 GPU 并行编程中, 硬件栅栏可以有效实现线程束级别合作同步, 然而, 目前 GPU 提供的硬件栅栏数量有限, 在一定程度上会限制线程束级别合作同步的灵活表达. 另外, GPU 硬件目前仅针对线程块集群内部的线程块提供硬件栅栏, 只能用来支持线程块级别合作同步中部分线程块的合作同步, 而非同一线程块集群内的线程块进行合作同步时, 则通常需要 GPU 函数的重复启动来实现, 开销巨大. 因而, 部分研究[48,49,51]聚焦于 GPU 线程块级别合作同步的表达. 在这其中, Xiao 等人[48]在软件层面实现了两种 GPU 全局同步栅栏: 基于原子操作的全局同步栅栏, 以及不使用原子操作的全局同步栅栏. Stuart 等人[49]使用原子操作在 GPU 上支持更灵活的线程块级别合作同步, 即任意两个线程块之间的合作同步. 除此以外, Li 等人[50]针对原子操作访存开销大而导致合作同步开销大的问题, 研究了 GPU 原子操作的微指令, 将这些微指令重新组合在线程块内多个线程之间, 以较低的访存开销在 GPU 中实现了生产者-消费者数据处理模式.

(2) 线程同步原语

现有 GPU 编程模型提供了一系列线程同步原语, 支持合作同步的表达. 以 CUDA 编程模型[13]为例, 它提供了支持线程束级别合作同步的原语__syncthreads() (如前文图 8(a) 所示), 线程块级别合作同步的原语 cudaLaunchCooperativeKernel(), 以及线程/线程束/线程块级别合作同步的原语 Cooperative Groups[52]等. 其中, 线程级别合作同步的原语主要针对不再严格按照 SIMT 方式执行的 GPU (比如, NVIDIA Volta 及以后的 GPU, 如前文图 2(b) 所示); 支持线程束级别合作同步及线程块集群内部线程块级别合作同步的原语充分利用了 GPU 提供的硬件栅栏, 同步开销较小.

线程同步原语通用性强. 然而, GPU 提供的上述线程同步原语灵活性较差, 特别是对线程块内部分线程束和部分线程块的合作同步的表达较为复杂, 且限制较多. 另外, 由于 NVIDIA 只在 Hopper 架构以后的 GPU 中, 针对线程块集群内的线程块 (如图 11 所示) 提供硬件栅栏, Cooperative Groups[52]在 Hopper 以前的 GPU 中表达线程块级别合作同步以及 Hopper 以后的 GPU 中表达线程块集群间的线程块合作同步时, 均会引入较大同步开销[63].

(3) 编程模型

针对 GPU 线程块级别合作同步存在的同步开销大的问题, 文献 [2−5,53−55] 探索新的 GPU 编程模型, 以减少

在 CPU 端重复启动 GPU 函数的次数, 从而降低线程块级别合作同步开销. 现有编程模型主要包括动态并行[2−5]和驻留线程 (persistent threads)[54,55]两类. 其中, 动态并行编程模型从 GPU 端动态启动函数, 以避免在 CPU 端进行 GPU 函数的启动. 在这其中, 文献 [5] 提出了一种动态生成轻量级线程块的方法, 以进一步减少 GPU 函数的启动开销, 并提高 GPU 流多处理器的占用率; 文献 [3] 将线程动态产生的 GPU 函数合并成不同粒度, 从而减少 GPU 函数的启动次数; 文献 [53] 令父线程 (parent threads) 始终保持活跃, 替代子函数 (subkernel) 的创建, 从而避免子函数启动引入的开销. 动态并行编程模型虽然能够减少线程块级别合作同步的开销, 但从 GPU 端启动函数极易导致负载不均衡.

驻留线程 (persistent threads) 编程模型[54,55]将函数驻留在 GPU 中, 直到所有任务完成后才结束 GPU 函数, 从而避免了 GPU 函数的重复启动. 与传统的并行编程模型不同, 在驻留线程编程模型中, 每个线程所要执行的任务是动态分配的, 而驻留线程的维度和规模、任务在驻留线程中的均衡分配均依靠编程人员的编程经验. 在这其中, 驻留线程维度和规模的设置直接影响 GPU 的硬件利用率, 而过大的线程规模还有可能导致死锁[64] (线程块级别同步中可能出现的死锁问题, 详见第 2.1 节), 因此需要根据实际 GPU 的硬件资源谨慎设置, 难度较大, 还会限制代码的可迁移性.

(4) 并行算法

线程同步与程序的数据结构、数据访问方式、并行方式等密切关联. 因而, 部分研究学者结合应用的自身特征针对其线程合作同步设计并行算法, 从而实现应用性能的提升. 在这其中, 很多研究[56−59]针对当前热点应用神经网络, 提出新的并行算法, 以更好地进行线程合作同步. 比如, Khorasani 等人[56]借鉴驻留线程编程模型的思想, 用一个单独的函数实现整个循环神经网络 (recurrent neural network, RNN), 并令其驻留在 GPU 中完成所有的时间步骤 (timesteps). 该研究根据 RNN 计算图中的节点依赖关系, 令所有线程块一起合作, 以串行的方式依次执行 RNN 中的神经元, 从而可将 RNN 中所有层的权重值静态分布在每个线程块的片上寄存器中, 增加权重值的重用率, 减少长时延的片外访存.

与竞争同步中并行算法的表达方法相同, 通过设计并行算法表达线程合作同步对特定应用会产生较好的加速效果, 具有执行效率高的优点. 然而, 这些方法不具有通用性.

(5) 线程块级别合作同步的死锁避免方法

上述栅栏、线程同步原语、编程模型, 以及部分并行算法的研究主要集中于线程块级别合作同步灵活、低开销的表达方法, 然而却以线程块规模不能超过 GPU 硬件资源容量为前提, 以避免因 GPU 非抢占式的线程块调度方式而导致的死锁. 为此, CUDA 编程模型[13]提供了 Cooperative Kernel 编程接口, 通过在 GPU 函数启动前计算应用的硬件资源占用情况并控制启动的线程块数量, 以避免死锁. 另外, 文献 [64] 提出了占用发现协议 (occupancy discovery protocol), 令 GPU 函数的所有线程块在执行任务前, 首先运行该协议, 并返回一个数值以确认该线程块在运行中. 通过这种方式, 该协议可动态预估 GPU 函数在实际 GPU 中的硬件占用情况, 得到能够在该 GPU 上同时并行执行的最大线程块数量, 并据此限制线程块的启动数量, 从而避免死锁.

以上研究可以避免线程块级别合作同步中的死锁. 然而, 通过限制线程块规模避免死锁的方法会导致应用范围受限, 且无法充分发挥 GPU 硬件的性能. 因此, Liu 等人[14]提出在 GPU 中实现上下文切换, 从而实现 GPU 线程块的抢占执行, 避免死锁. 与其他针对多应用在 GPU 中进行上下文切换的研究[65−67]不同, 他们主要针对单一应用中线程块间的上下文切换. 受文献 [66] 提出的部分上下文切换策略启发, 他们针对线程块级别合作同步的特点, 进一步压缩上下文数据, 优化上下文数据存储方式, 以减少上下文信息存储规模, 加速线程块切换时上下文信息的存取速度. 除此之外, 他们还通过对应用线程块级别合作同步的分析, 设计上下文切换策略, 以避免不必要的上下文切换, 减少上下文切换次数. 该研究虽然能够避免死锁, 然而其线程块上下文切换方法仍会引入较大开销, 需继续进行探索.

综上所述, GPU 能够高效的支持线程和线程束级别合作同步. 然而, 由于 GPU 硬件不支持线程块级别合作同步 (NVDIA 从 Hopper GPU 才开始在硬件层面支持线程块集群内部的线程块间合作同步), 导致线程块级别合作同步开销巨大. 另外, GPU 采取非抢占式的线程块调度方式, 使得线程块间的合作同步存在死锁问题. 因此, 已有绝

大多数研究都聚焦于 GPU 线程块级别合作同步的研究. 这些研究虽然能比较有效地减少线程块级别合作同步的开销, 但仍无法高效地避免死锁问题.

4 GPU 线程同步性能优化方法研究

结合本文第 2.2 节分析总结的 GPU 线程同步的执行存在的问题和挑战, 本节围绕线程同步的执行, 分别对 GPU 竞争同步和合作同步的性能优化方法及其研究进程进行综述分析.

4.1 面向竞争同步的性能优化方法

如第 2.2 节所述, 频繁的同步失败导致 GPU 竞争同步执行效率低. 在通用 CPU 处理器体系结构中, 大量研究工作聚焦于减少竞争同步失败的次数. 他们针对不同的竞争同步表达方法 (比如互斥锁、事务内存等) 提出了多种方案, 可分为共享数据竞争管理 (shared data contention management) 和并行控制 (concurrency control) 两大类. 其中, 共享数据竞争管理是对共享数据的并行访问进行管理, 设计访问策略减少或避免并行线程对同一共享数据的同时访问, 从而减少竞争同步失败次数. 典型的共享数据竞争管理方法即为回退 (back-off) 机制[68], 而指数回退机制 (在指数回退机制中, 线程因同步失败而等待再次进行同步的时延随该线程同步失败次数的增加指数增长) 被认为是最有效减少竞争同步失败次数的回退机制. 与共享数据竞争管理不同, 并行控制机制将程序的执行过程划分为若干采样阶段和执行阶段. 采样阶段会对部分 CPU 核中竞争同步失败的次数进行监测、统计及分析, 并据此在执行阶段对应用的实际并行度进行调整, 从而达到减少同步失败次数的目的.

然而, 与通用 CPU 处理器不同, GPU 线程规模巨大, 线程组织结构复杂, 且缺少高效的运行时管理机制, 这使得上述通用 CPU 处理器中降低同步失败次数的研究无法适用于 GPU 中. 比如, 图 12 列出了在 GPU 中实现回退机制的经典方法. 该方法会导致线程束中成功进行同步的线程被迫进入等待状态. 另外, 通过重复统计时钟 (cycle) 信息以获取等待时延的方法还会引入较大的额外开销, 造成 GPU 硬件资源的浪费[18]. 因此, 在程序共享数据访问冲突不是很大的情况下, 该方法甚至会导致程序性能的下降[19].

```
clock_t start = clock();
clock_t now;
for(;;){
   now = clock();
   clock_t cycles = now > start ? now  - start : now + (0xffffffff – start);
   if(cycles >= DELAY_FACTOR * blockIdx.x ){
      // 等待时延大于某一阈值时, 才会停止等待,继续执行
      break;
   }
}
```

图 12 GPU 并行编程中回退机制的经典实现方法[18,19]

鉴于此, 文献 [12,18,19] 在 GPU 中探索减少同步失败次数的方法. 在这其中, 文献 [18,19] 仿照通用 CPU 处理器中的共享数据竞争管理方法, 根据 GPU 体系结构的特点设计了针对同步失败的回退机制. ElTantawy 等人[19]从硬件层面入手, 采用硬件检测的方法, 检测线程处于自旋状态的线程束, 根据自旋次数决定线程所在线程束的优先级, 并据此设计 GPU 线程束调度策略, 以降低线程自旋的次数, 从而减少同步失败次数.

Gao 等人[18]则提出在 GPU 中采用纯软件的方法减少同步失败次数. 他们设计了纯软件线程束调度框架 SWCF. 在 SWCF 中, 他们将同一线程块内的线程束分为一个控制者 (controller) 和若干工作者 (worker). 其中, 控制者控制工作者的执行; 工作者正常执行任务, 并检测自己的执行是否会对程序性能造成影响, 在可能对程序性能造成影响时暂停执行. 通过利用 GPU 提供的有限硬件栅栏, 他们在 GPU 线程块内设计并实现了生产者-消费者执行模式, 使得控制者 (生产者) 和工作者 (消费者) 均能采用基于反馈的方式运行, 降低了 SWCF 引入的额外开销. 在 SWCF 的基础上, 他们进一步根据 GPU 并行编程中竞争同步的执行特点, 设计共享数据冲突管理策略, 在避免严重影响应用实际并行度的前提下, 减少并发线程对共享数据的冲突访问, 从而减少同步失败次数.

考虑到 GPU 不同线程组织的执行方式差异巨大, 在 GPU 中对部分 SM 的执行情况采样较为复杂, 且会对应

用实际并行度造成较大影响, 因而目前尚未有相关研究在 GPU 中实现通用 CPU 处理器中的并行控制机制. 上述研究虽然对 GPU 竞争同步执行效率的提升有积极作用, 但仍存在一些不足. 其中, 文献 [12] 由于硬件开销的限制而导致锁条目数量有限, 对应用性能有一定影响. 而文献 [18,19] 均以线程束为单位进行调度, 通过并发控制降低同步失败的次数. 这种方法对于线程束间的共享数据冲突有效, 然而却无法缓解线程束内线程间的共享数据冲突. 整体而言, 目前 GPU 并行编程中面向竞争同步性能优化方法研究较少, 需要更多深入的探索.

4.2 面向合作同步的性能优化方法

如第 2.2 节所述, GPU 简单的调度机制及调度策略极易导致合作同步中同步主体执行进度不一致, 使得部分同步主体需要长时间等待, 影响程序性能并造成硬件资源浪费. 针对这个问题, 现有研究[23,69,70]大多采用在硬件层面调度线程束的方式对线程合作同步的性能进行优化.

这些研究中, Liu 等人[23]针对线程束长时间阻塞的问题, 提出令相同线程块内执行时间较长的线程束有最高执行优先级的调度策略, 以减少线程块内线程束的等待时间. 他们首先设计等待数量多者优先 (most-waiting-first, MWF) 策略, 令有最多线程束在同步点等待的线程块拥有最高优先级, 而后在此基础上, 进一步设计关键指令取指优先 (critical-fetch-first, CFF) 策略, 在进行下一个周期的取指时, 优先取根据 MWF 策略最有可能被调入执行的线程块中线程束将要执行的指令, 以消除线程束调度不匹配 (MWF 策略引入) 导致的取指和执行阶段间的阻塞.

文献 [69] 则主要关注 GPU 流多处理器中因多个线程束调度策略的不同而导致的进行合作同步的同步主体执行进度差异巨大, 从而造成部分同步主体长时间等待的问题. 他们在 GPU 流多处理器中已有的多个线程束调度策略的基础上, 进一步提出线程合作同步已知的协作策略 SAWS, 分别针对线程束到达同步点以及线程束在同步点被阻塞两种情况, 设计不同的算法对 (不同线程束调度策略中) 进行合作同步的线程束的执行顺序进行调整, 以最大程度减少线程块内线程束因合作同步而导致的等待时间.

上述研究能够降低线程束级别合作同步中线程束的等待时间. 然而, 这些线程束调度方法只针对 GPU 编程模型提供的线程同步原语 (比如__syncthreads()) 有效, 而无法对通过其他方式表达的合作同步 (比如通过原子操作实现的栅栏) 的性能进行优化. 与他们的研究不同, Li 等人[70]从编程模型的角度入手, 设计令每个线程块只包含一个线程束, 从而避免了线程束级别合作同步. 这种方式虽然能够彻底解决线程束级别合作同步执行效率低的问题, 但不具有通用性, 应用范围较小. 另外, 已有工作主要针对线程束级别合作同步展开研究, 尚未考虑线程块级别合作同步中, 因线程块执行进度差异而导致的执行效率低的问题.

综上所述, 已有的 GPU 线程同步性能优化研究虽然能够在一定程度上提高 GPU 线程同步的执行效率, 但受 GPU 体系结构限制 (线程组织结构复杂、调度控制逻辑简单、且无运行时软件的支持), 大多数工作调度优化内容单一、优化策略简单, 效果不甚理想. 比如, 已有研究未能考虑线程级别竞争同步导致的分支执行. 另外, 目前绝大多数工作均重点关注在 GPU 中正确、高效地实现线程同步, 而对 GPU 线程同步性能优化的研究尚处于起步阶段, 仍有很大的探索及优化空间.

5 GPU 线程同步研究趋势和发展前景

基于上述国内外研究现状, 本文认为, 对 GPU 并行编程中线程同步的研究应从以下几个方面寻求突破, 并有如下的发展趋势.

5.1 系统化研究必不可少

现有研究虽然从不同角度提出了改善 GPU 线程同步的方法或技术, 但大多针对某一方面、或某一局部问题, 比如只针对单一粒度线程同步的表达, 或者只关注解决线程同步可能引入的错误. 然而, GPU 线程同步存在的难以高效表达、错误频发、执行效率低的问题既互相独立又互相关联. 另外, 这些问题源于 GPU 独特的体系结构和并行模式, 与编程模型和语言、运行时系统、体系结构多个层面均有关联. 因此, 需要针对 GPU 线程同步开展系统化的研究: 一方面使得软件能够更好地适配 GPU 硬件体系结构, 根据硬件约束优化算法设计; 另一方面软件需求驱动硬件体系结构升级, 持续丰富硬件的线程同步能力, 提升同步效率, 最终形成完善的 GPU 线程同步机制.

5.2 亟需更加灵活高效的线程同步表达方法

随着 GPU 可编程范围的不断扩展, GPU 应用的线程同步模式日益复杂. 然而, GPU 的体系结构特点、线程执行和调度方式却导致在表达线程同步时难以兼顾正确性、灵活性、高效性. 现有国内外研究工作重点解决线程同步在 GPU 中的正确表达, 灵活性较差, 且往往以增加线程同步的开销为代价. 比如, 现有线程合作同步表达方法难以满足神经网络应用复杂的线程同步通信需求; 对于线程级别竞争同步中同步失败导致的频繁分支执行, 已有研究尚未提出有效的解决方案. 因此, 本文认为, 在保证 GPU 线程同步表达正确性的基础上, 未来对更加灵活、高效的线程同步表达方法的研究是一个重要的研究方向.

在竞争同步的表达方面, 广泛使用的互斥锁及事务内存的实现均基于原子操作. 然而, 现有 GPU 内存模型会导致较大的原子操作开销. 另外, 线程级别竞争同步中原子操作的对象具有随机性, 会导致大量不连续的内存访问, 引入极大的访存开销. 因此, 在现有研究基础上, 进一步减少互斥锁/事务内存中原子操作的使用、降低原子操作开销、避免互斥锁/事务内存中原子操作对全局内存的访问是未来研究高效竞争同步表达方法需要解决的关键问题.

在合作同步的表达方面, 由于 GPU 流多处理器间同步通信硬件的缺失以及非抢占式的线程块调度方式, GPU 线程块级别合作同步的高效表达和灵活的死锁避免方法是未来需要关注的两个研究内容. 对于前者, NVIDIA 已在其 Hopper 架构中引入线程块集群 (由部分流多处理器组成的集群) 的概念, 并在其内部设计专用的同步通信硬件. 然而, 如何设计硬件结构满足所有流多处理器间的同步通信、如何基于同步通信硬件设计灵活高效的线程块合作同步表达方法仍是尚未解决的关键问题. 针对 GPU 线程块级别合作同步的死锁问题, 已有研究主要集中于线程块规模的限制, 使得应用范围受限, 且无法充分发挥 GPU 硬件的性能. 因此, 本文认为, 探索 GPU 线程块切换, 打破 GPU 线程块的非抢占式调度方式是最直接有效的研究方法.

5.3 灵活的运行时调度对线程同步执行效率至关重要

由于 GPU 强大的计算能力和巨大的并行规模, 仅在 GPU 中正确实现线程同步, 即可使应用获得比 CPU 高达数十倍的加速比[9]. 因此, 到目前为止的绝大多数研究尚未兼顾线程同步的执行效率. 鉴于此种现状, 同时伴随 GPU 线程同步表达方法的研究越来越完善, 未来 GPU 线程同步的研究会更加关注线程同步执行效率的提升. 如本文第 4 节所述, 目前对 GPU 线程同步性能优化的研究尚处于起步阶段, 已有研究往往将通用 CPU 处理器中线程同步的性能优化方法移植到 GPU 中, 且大多采用线程束调度方式进行优化, 尚未针对 GPU 体系结构特征以及 GPU 中线程同步行为特征进行有针对性的、高效的性能优化. 因此, 本文认为, GPU 线程同步的性能优化尚有巨大的探索空间, 需要进一步提升应用性能, 提高 GPU 硬件利用率.

线程同步对实际执行过程中的并行规模、共享数据冲突程度、线程调度等因素影响很大, 需要针对线程同步持续优化 GPU 任务调度. 与 GPU 硬件层面的任务调度相比, 在运行时层面进行任务调度更加灵活, 应用范围也更广. 不仅如此, 近年来 GPU 作为应用最为广泛的加速器, 越来越多的部署在各种数据中心及超级计算机中, 如何有效优化这些 GPU 中线程同步的性能以降低能源消耗也越来越重要. 基于上述原因, 本文认为, 在从编程模型和语言、体系结构层面入手展开研究的基础上, 在运行时系统层面针对线程同步持续优化 GPU 任务调度是未来线程同步性能优化的一个重要研究方向.

然而, GPU 体系结构中控制逻辑非常简单, 且无操作系统和运行时系统支持. 因此, 如何在 GPU 硬件约束下设计任务调度机制, 以对 GPU 线程同步执行效率进行优化是研究难点. 另外, 由于 GPU 以高并行度和高吞吐量为系统优化目标, 不适合的任务调度不仅无法提升应用性能, 反而极易导致程序并行度降低, 造成应用性能的下降. 如何结合 GPU 的并行特点设计线程同步任务调度策略也需进一步探索.

5.4 有必要与更多应用的自身特征进行深度融合

线程同步作为并行程序的重要特征之一, 与程序的数据结构、数据访问方式、并行方式等密切关联. 因而, 结合应用的自身特征对其线程同步进行优化, 对应用性能的提升具有重要意义. 目前, 现有研究已针对一些典型的算法和应用, 在 GPU 中提出了有效的线程同步优化方案, 并取得了较好的加速效果. 然而, 已有研究涉及的应用范围较小, 没有充分研究更广泛应用的线程同步特征. 比如, 由于缺少高效的线程同步的支持, 神经网络往往通过函数

的重复启动进行线程同步, 引入较大的同步开销, 还导致极低的片上空间数据复用率, 严重影响性能提升. 随着 GPU 并行计算可编程范围的不断扩大, 在完善 GPU 通用线程同步机制的基础上, 未来需要针对更多应用, 且与应用自身的特征进行更深层次的融合, 才能有效优化应用的线程同步, 获得应用性能的显著提升.

在众多应用中, 线程同步与典型神经网络应用的深度融合尤为重要. 在神经网络应用的实现中, 并行算法和线程同步二者互相掣肘: 基于数据流图 (data flow graph, DFG) 的并行算法导致同步通过高开销的 GPU 函数启动隐式完成; 另一方面, 现有 GPU 对线程同步的支持又不完善, 制约了并行算法的优化. 因此, 从优化线程同步的角度出发探索神经网络的优化空间是一个重要的研究方向, 能够有效提高神经网络在 GPU 中的编程效率和性能. 其中, 如何使并行算法和线程同步二者互相促进是未来要解决的关键问题, 包括权衡 GPU 不同层次线程同步能力, 并据此探索神经网络并行算法的优化空间; 根据应用计算需求的本质, 进一步丰富 GPU 的线程同步能力.

6 结束语

GPU 线程同步的研究对于 GPU 应用性能的提升和通用性的继续发展有重要意义, 但 GPU 独特的体系结构和并行模式给 GPU 线程同步正确、高效的表达和执行带来很多问题和挑战. 本文的贡献在于根据不同的线程同步目的和粒度对 GPU 并行编程中的线程同步进行了归类说明; 分析总结了 GPU 并行编程中线程同步的表达及执行面临的线程同步开销巨大、死锁活锁问题频繁、执行效率低的关键问题和挑战; 在综述国内外 GPU 线程同步表达和性能优化方法研究现状的基础上, 指出了 4 个重要的 GPU 线程同步的未来研究方向, 并针对每个研究方向给出可能的研究思路, 以及会遇到的问题及挑战. 能够为相关学者提供一定程度的帮助, 不仅有利于 GPU 体系架构中线程同步的研究, 还对其他单指令多数据 (single instruction multiple data)、多指令多数据 (multiple instructions multiple data) 等体系结构中线程同步的研究具有较好的指导意义. 本文认为, 鉴于当前 GPU 通用计算的快速发展和普及, 线程同步将对 GPU 应用性能和硬件利用率的提升起到越来越关键的作用.

References:

[1] ElTantawy A, Aamodt TM. MIMD synchronization on SIMT architectures. In: Proc. of the 49th Annual IEEE/ACM Int'l Symp. on Microarchitecture. IEEE, 2016. 1–14. [doi: 10.1109/MICRO.2016.7783714]

[2] Jones S. Introduction to dynamic parallelism. In: Proc. of the 2012 GPU Technology Conf. San Jose: NVIDIA, 2012. 338–2012.

[3] Wu HC, Li D, Becchi M. Compiler-assisted workload consolidation for efficient dynamic parallelism on GPU. In: Proc. of the 2016 IEEE Int'l Parallel and Distributed Processing Symp. Chicago: IEEE, 2016. 534–543. [doi: 10.1109/IPDPS.2016.98]

[4] El Hajj I, Gómez-Luna J, Li C, Chang LW, Milojicic D, Hwu WM. KLAP: Kernel launch aggregation and promotion for optimizing dynamic parallelism. In: Proc. of the 49th Annual IEEE/ACM Int'l Symp. on Microarchitecture. IEEE, 2016. 1–12. [doi: 10.1109/MICRO.2016.7783716]

[5] Wang J, Rubin N, Sidelnik A, Yalamanchili S. Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on GPUs. In: Proc. of the 42nd Annual Int'l Symp. on Computer Architecture. Portland: ACM, 2015. 528–540. [doi: 10.1145/2749469.2750393]

[6] Nelson J, Miller D, Palmieri R. Don't forget about synchronization! Guidelines for using locks on graphics processing units. Concurrency and Computation: Practice and Experience, 2022, 34(2): e5757. [doi: 10.1002/cpe.5757]

[7] Wang K, Don F, Calvin L. Fast fine-grained global synchronization on GPUs. In: Proc. of the 24th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Providence: ACM, 2019. 793–806. [doi: 10.1145/3297858.3304055]

[8] Xu YL, Gao L, Wang R, Luan ZZ, Wu WG, Qian DP. Lock-based synchronization for GPU architectures. In: Proc. of the 2016 ACM Int'l Conf. on Computing Frontiers. Como: ACM, 2016. 205–213. [doi: 10.1145/2903150.2903155]

[9] Gao L, Xu YL, Wang R, Luan ZZ, Yu ZB, Qian DP. Thread-level locking for SIMT architectures. IEEE Trans. on Parallel and Distributed Systems, 2020, 31(5): 1121–1136. [doi: 10.1109/TPDS.2019.2955705]

[10] Kaleem R, Venkat A, Pai S, Hall M, Pingali K. Synchronization trade-offs in GPU implementations of graph algorithms. In: Proc. of the 2016 IEEE Int'l Parallel and Distributed Processing Symp. Chicago: IEEE, 2016. 514–523. [doi: 10.1109/IPDPS.2016.106]

[11] Liu LF, Liu ML, Wang CJ, Wang J. Compile-time automatic synchronization insertion and redundant synchronization elimination for GPU kernels. In: Proc. of the 22nd IEEE Int'l Conf. on Parallel and Distributed Systems. Wuhan: IEEE, 2016. 826–834. [doi: 10.1109/ICPADS.2016.0112]

[12] Yilmazer A, Kaeli D. HQL: A scalable synchronization mechanism for GPUs. In: Proc. of the 27th IEEE Int'l Symp. on Parallel and

Distributed Processing. Cambridge: IEEE, 2013. 475–486. [doi: 10.1109/IPDPS.2013.82]
[13] NVIDIA. CUDA programming guide. 2022. https://docs.nvidia.com/cuda/cuda-c-programming-guide/contents.html
[14] Liu JW. Efficient synchronization for GPGPU [Ph.D. Thesis]. Pittsburgh: University of Pittsburgh, 2018.
[15] Durant L, Giroux O, Harris M, Stam N. Inside Volta: The world's most advanced data center GPU. 2022. https://developer.nvidia.com/blog/inside-volta/
[16] Ramamurthy A. Towards scalar synchronization in SIMT architectures [MS. Thesis]. British Columbia: University of British Columbia, 2011.
[17] Fung WWL, Singh I, Brownsword A, Aamodt TM. Hardware transactional memory for GPU architectures. In: Proc. of the 44th Annual IEEE/ACM Int'l Symp. on Microarchitecture. Porto Alegre: ACM, 2011. 296–307. [doi: 10.1145/2155620.2155655]
[18] Gao L, Wang J, Zhang WG. Adaptive contention management for fine-grained synchronization on commodity GPUs. ACM Trans. on Architecture and Code Optimization, 2022, 19(4): 58. [doi: 10.1145/3547301]
[19] ElTantawy A, Aamodt TM. Warp scheduling for fine-grained synchronization. In: Proc. of the 2018 IEEE Int'l Symp. on High Performance Computer Architecture. Vienna: IEEE, 2018. 375–388. [doi: 10.1109/HPCA.2018.00040]
[20] Burtscher M, Pingali K. An efficient CUDA implementation of the tree-based barnes hut *n*-body algorithm. GPU Computing Gems Emerald Edition. Burlington: Morgan Kaufmann, 2011. 75–92. [doi: 10.1016/B978-0-12-384988-5.00006-1]
[21] The H-Store Team. SmallBank Benchmark. 2022. https://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/
[22] Che S, Boyer M, Meng JY, Tarjan D, Sheaffer JW, Lee SH, Skadron K. Rodinia: A benchmark suite for heterogeneous computing. In: Proc. of the 2009 IEEE Int'l Symp. on Workload Characterization. Austin: IEEE, 2009. 44–54. [doi: 10.1109/IISWC.2009.5306797]
[23] Liu YX, Yu ZB, Eeckhout L, Reddi VJ, Luo YW, Wang XL, Wang ZL, Xu CZ. Barrier-aware warp scheduling for throughput processors. In: Proc. of the 2016 Int'l Conf. on Supercomputing. Istanbul: ACM, 2016. 42. [doi: 10.1145/2925426.2926267]
[24] Khronos. OpenCL. 2022. http://www.khronos.org/opencl/
[25] Sanders J, Kandrot E. CUDA by Example: An Introduction to General-purpose GPU Programming. Boston: Addison-Wesley Professional, 2010.
[26] Ren XW, Lis M. High-performance GPU transactional memory via eager conflict detection. In: Proc. of the 2018 IEEE Int'l Symp. on High Performance Computer Architecture. Vienna: IEEE, 2018. 235–246. [doi: 10.1109/HPCA.2018.00029]
[27] Chen S, Peng L, Irving S. Accelerating GPU hardware transactional memory with snapshot isolation. In: Proc. of the 44th Annual Int'l Symp. on Computer Architecture. Toronto: ACM, 2017. 282–294. [doi: 10.1145/3079856.3080204]
[28] Villegas A, Asenjo R, Navarro A, Plata O, Kaeli D. Lightweight hardware transactional memory for GPU scratchpad memory. IEEE Trans. on Computers, 2018, 67(6): 816–829. [doi: 10.1109/tc.2017.2776908]
[29] Chen S, Peng L. Efficient GPU hardware transactional memory through early conflict resolution. In: Proc. of the 2016 IEEE Int'l Symp. on High Performance Computer Architecture. Barcelona: IEEE, 2016. 274–284. [doi: 10.1109/HPCA.2016.7446071]
[30] Fung WWL, Aamodt TM. Energy efficient GPU transactional memory via space-time optimizations. In: Proc. of the 46th Annual IEEE/ACM Int'l Symp. on Microarchitecture. Davis: ACM, 2013. 408–420. [doi: 10.1145/2540708.2540743]
[31] Villegas A, Navarro Á, Asenjo-Plaza R, Plata-González ÓG, Ubal R, Kaeli D. Hardware support for local memory transactions on GPU architectures. In: Proc. of the 2015 SIGPLAN Workshop on Transactional Computing. Portland: ACM, 2015.
[32] Irving S, Peng L, Busch C, Peir JK. BifurKTM: Approximately consistent distributed transactional memory for GPUs. In: Proc. of the 2021 PARMA-DITAM in Conjunction with European Network on High-performance Embedded Architecture and Compilation. 2021. 2.
[33] Holey A, Zhai A. Lightweight software transactions on GPUs. In: Proc. of the 43rd Int'l Conf. on Parallel Processing. Minneapolis: IEEE, 2014. 461–470. [doi: 10.1109/ICPP.2014.55]
[34] Shen Q, Sharp C, Blewitt W, Ushaw G, Morgan G. PR-STM: Priority rule based software transactions for the GPU. In: Proc. of the 21st Int'l Conf. on Parallel and Distributed Computing. Vienna: Springer, 2015. 361–372. [doi: 10.1007/978-3-662-48096-0_28]
[35] Xu YL, Wang R, Goswami N, Li T, Gao L, Qian DP. Software transactional memory for GPU architectures. In: Proc. of the 2014 Annual IEEE/ACM Int'l Symp. on Code Generation and Optimization. Orlando: ACM, 2014. 1–10. [doi: 10.1145/2581122.2544139]
[36] Cederman D, Tsigas P, Chaudhry MT. Towards a software transactional memory for graphics processors. In: Proc. of the 10th Eurographics Conf. on Parallel Graphics and Visualization. Norrköping: ACM, 2010. 121–129.
[37] Shi XH, Luo X, Liang JL, Zhao P, Di S, He BS, Jin H. Frog: Asynchronous graph processing on GPU with hybrid coloring model. IEEE Trans. on Knowledge and Data Engineering, 2018, 30(1): 29–42. [doi: 10.1109/tkde.2017.2745562]
[38] Liu WF, Li A, Hogg J, Duff IS, Vinter B. A synchronization-free algorithm for parallel sparse triangular solves. In: Proc. of the 22nd Int'l Conf. on Parallel and Distributed Computing. Grenoble: Springer, 2016. 617–630. [doi: 10.1007/978-3-319-43659-3_45]
[39] Narse R, Burtscher M, Pingali K. Morph algorithms on GPUs. In: Proc. of the 18th ACM SIGPLAN Symp. on Principles and Practice of

Parallel Programming. Shenzhen: ACM, 2013. 147–156. [doi: 10.1145/2442516.2442531]

[40] He BS, Yu JX. High-throughput transaction executions on graphics processors. Proc. of the VLDB Endowment, 2011, 4(5): 314–325. [doi: 10.14778/1952376.1952381]

[41] Ren XW, Lis M. Efficient sequential consistency in GPUs via relativistic cache coherence. In: Proc. of the 2017 IEEE Int'l Symp. on High Performance Computer Architecture. Austin: IEEE, 2017. 625–636. [doi: 10.1109/HPCA.2017.40]

[42] Alsop J, Orr MS, Beckmann BM, Wood DA. Lazy release consistency for GPUs. In: Proc. of the 49th Annual IEEE/ACM Int'l Symp. on Microarchitecture. IEEE, 2016. 1–14. [doi: 10.1109/MICRO.2016.7783729]

[43] Sinclair MD, Alsop J, Adve SV. Efficient GPU synchronization without scopes: Saying no to complex consistency models. In: Proc. of the 48th Int'l Symp. on Microarchitecture. Waikiki: ACM, 2015. 647–659. [doi: 10.1145/2830772.2830821]

[44] Singh A, Aga S, Narayanasamy S. Efficiently enforcing strong memory ordering in GPUs. In: Proc. of the 48th Int'l Symp. on Microarchitecture. Waikiki: ACM, 2015. 699–712. [doi: 10.1145/2830772.2830778]

[45] Singh I, Shriraman A, Fung WWL, O'Connor M, Aamodt TM. Cache coherence for GPU architectures. In: Proc. of the 19th IEEE Int'l Symp. on High Performance Computer Architecture. Shenzhen: IEEE, 2013. 578–590. [doi: 10.1109/HPCA.2013.6522351]

[46] Andersch M, Palmer G, Krashinsky R, Stam N, Mehta V, Brito G, Ramaswamy S. NVIDIA Hopper architecture in-depth. 2022. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

[47] Lin YZ, Zhang WH. A research on GPU transactional memory. Big Data Research, 2020, 6(4): 3–17 (in Chinese with English abstract). [doi: 10.11959/j.issn.2096-0271.2020029]

[48] Xiao SC, Feng WC. Inter-block GPU communication via fast barrier synchronization. In: Proc. of the 2010 IEEE Int'l Symp. on Parallel and Distributed Processing. Atlanta: IEEE, 2010. 1–12. [doi: 10.1109/IPDPS.2010.5470477]

[49] Stuart JA, Owens JD. Efficient synchronization primitives for GPUs. arXiv:1110.4623, 2011.

[50] Li A, van den Braak GJ, Corporaal H, Kumar A. Fine-grained synchronizations and dataflow programming on GPUs. In: Proc. of the 29th ACM on Int'l Conf. on Supercomputing. Newport Beach: ACM, 2015. 109–118. [doi: 10.1145/2751205.2751232]

[51] Kim JY, Batten C. Accelerating irregular algorithms on GPGPUs using fine-grain hardware worklists. In: Proc. of the 47th Annual IEEE/ACM Int'l Symp. on Microarchitecture. Cambridge: IEEE, 2014. 75–87. [doi: 10.1109/MICRO.2014.24]

[52] Harris M, Perelygin K. Cooperative Groups: Flexible CUDA thread programming. 2017. https://developer.nvidia.com/blog/cooperative-groups/

[53] Chen GY, Shen XP. Free launch: Optimizing GPU dynamic kernel launches through thread reuse. In: Proc. of the 48th Int'l Symp. on Microarchitecture. Waikiki: ACM, 2015. 407–419. [doi: 10.1145/2830772.2830818]

[54] Belviranli ME, Deng P, Bhuyan LN, Gupta R, Zhu Q. PeerWave: Exploiting wavefront parallelism on GPUs with peer-SM synchronization. In: Proc. of the 29th ACM on Int'l Conf. on Supercomputing. Newport Beach: ACM, 2015. 25–35. [doi: 10.1145/2751205.2751243]

[55] Gupta K, Stuart JA, Owens JD. A study of persistent threads style GPU programming for GPGPU Workloads. In: Proc. of the 2012 Innovative Parallel Computing. San Jose: IEEE, 2012. 1–14. [doi: 10.1109/InPar.2012.6339596]

[56] Khorasani F, Esfeden HA, Abu-Ghazaleh N, Sarkar V. In-register parameter caching for dynamic neural nets with virtual persistent processor specialization. In: Proc. of the 51st Annual IEEE/ACM Int'l Symp. on Microarchitecture. Fukuoka: IEEE, 2018. 377–389. [doi: 10.1109/MICRO.2018.00038]

[57] Diamos G, Sengupta S, Catanzaro B, Chrzanowski M, Coates A, Elsen E, Engel J, Hannun A, Satheesh S. Persistent RNNs: Stashing recurrent weights on-chip. In: Proc. of the 33rd Int'l Conf. on Machine Learning. New York: JMLR.org, 2016. 2024–2033.

[58] Appleyard J, Kocisky T, Blunsom P. Optimizing performance of recurrent neural networks on GPUs. arXiv:1604.01946, 2016.

[59] Yan SG, Long GP, Zhang YQ. StreamScan: Fast scan algorithms for GPUs without global barrier synchronization. In: Proc. of the 18th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. Shenzhen: ACM, 2013. 229–238. [doi: 10.1145/2442516.2442539]

[60] NVIDIA PTX ISA. Parallel Thread Execution ISA Version 8.2. 2022. https://docs.nvidia.com/cuda/parallel-thread-execution

[61] Bauer M, Treichler S, Aiken A. Singe: Leveraging warp specialization for high performance on GPUs. In: Proc. of the 19th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. Orlando: ACM, 2014. 119–130. [doi: 10.1145/2555243.2555258]

[62] Bauer M, Cook H, Khailany B. CudaDMA: Optimizing GPU memory bandwidth via warp specialization. In: Proc. of the 2011 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis. Seattle: ACM, 2011. 12. [doi: 10.1145/2063384.2063400]

[63] Zhang LQ, Wahib M, Zhang HY, Matsuoka S. A study of single and multi-device synchronization methods in NVIDIA GPUs. In: Proc. of the 2020 IEEE Int'l Parallel and Distributed Processing Symp. New Orleans: IEEE, 2020. 483–493. [doi: 10.1109/IPDPS47924.2020.00057]

[64] Sorensen T, Donaldson AF, Batty M, Gopalakrishnan G, Rakamarić Z. Portable inter-workgroup barrier synchronisation for GPUs. In: Proc. of the 2016 ACM SIGPLAN Int'l Conf. on Object-oriented Programming, Systems, Languages, and Applications. Amsterdam: ACM, 2016. 39–58. [doi: 10.1145/2983990.2984032]

[65] Park JJK, Park Y, Mahlke S. Chimera: Collaborative preemption for multitasking on a shared GPU. ACM SIGPLAN Notices, 2015, 50(4): 593–606. [doi: 10.1145/2775054.2694346]

[66] Wang ZN, Yang J, Melhem R, Childers B, Zhang YT, Guo MY. Simultaneous multikernel: Fine-grained sharing of GPUs. IEEE Computer Architecture Letters, 2016, 15(2): 113–116. [doi: 10.1109/LCA.2015.2477405]

[67] Tanasic I, Gelado I, Cabezas J, Ramirez A, Navarro N, Valero M. Enabling preemptive multiprogramming on GPUs. ACM SIGARCH Computer Architecture News, 2014, 42(3): 193–204. [doi: 10.1145/2678373.2665702]

[68] Agarwal A, Cherian M. Adaptive backoff synchronization techniques. ACM SIGARCH Computer Architecture News, 1989, 17(3): 396–406. [doi: 10.1145/74926.74970]

[69] Liu JW, Yang J, Melhem R. SAWS: Synchronization aware GPGPU warp scheduling for multiple independent warp schedulers. In: Proc. of the 48th Int'l Symp. on Microarchitecture. Waikiki: ACM, 2015. 383–394. [doi: 10.1145/2830772.2830822]

[70] Li A, Liu WF, Wang LN, Barker K, Song SL. Warp-consolidation: A novel execution model for GPUs. In: Proc. of the 2018 Int'l Conf. on Supercomputing. Beijing: ACM, 2018. 53–64. [doi: 10.1145/3205289.3205294]

附中文参考文献:

[47] 林玉哲, 张为华. GPU事务性内存技术研究. 大数据, 2020, 6(4): 3–17. [doi: 10.11959/j.issn.2096-0271.2020029]

高岚(1987－), 女, 博士, 讲师, CCF 专业会员, 主要研究领域为并行编程优化, 并行编程模型, GPU 体系结构.

王晶(1982－), 女, 博士, 副教授, CCF 专业会员, 主要研究领域为计算机系统结构, 智能芯片设计, 高能效计算, 容错计算.

赵雨晨(1999－), 女, 硕士生, CCF 学生会员, 主要研究领域为并行编程优化, 计算机体系结构.

钱德沛(1952－), 男, 教授, 博士生导师, CCF 会士, 主要研究领域为高性能计算机体系结构, 分布式系统, 众核处理器并行编程.

张伟功(1967－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为高可靠嵌入式计算机体系结构与应用技术.