GPU事务性内存技术研究

林玉哲[1,2]，张为华[1,2]
1. 复旦大学软件学院，上海 201203；
2. 上海市数据科学重点实验室，上海 201203

摘要
GPU是并行计算领域重要的体系结构之一，然而在面对高数据竞争的场景时，程序员往往需要设计复杂的并行方案。为了简化这一过程，GPU事务性内存实现了复杂的数据同步和并行，对外则仅提供简单的API。首先介绍了GPU事务性内存的研究背景。其次，讨论了近年的GPU事务性内存的设计方案与策略，分析了不同设计方案遇到的问题和解决方案，包括硬件和软件上的实现。最后对GPU事务性内存的现状和未来的发展做出了总结和展望。

关键词
GPU；事务性内存；并行计算

中图分类号：TP302 文献标识码：A doi: 10.11959/j.issn.2096-0271.2020029

A research on GPU transactional memory

LIN Yuzhe[1,2], ZHANG Weihua[1,2]
1. Software School, Fudan University, Shanghai 201203, China
2. Shanghai Key Laboratory of Data Science, Shanghai 201203, China

Abstract
GPU is one of the important architectures in parallel computing, however, when dealing with high data racing scenarios, programmers often need to design complex parallel schemes. In order to simplify this process, GPU transactional memory implements complex data synchronization and parallelism, and only provides simple API. The research background of GPU transactional memory was introduced. Then, the designs and strategies of GPU transactional memory in recent years were discussed, and the problems and solutions of different designs were analyzed, including the implementation of hardware and software. Finally, the current situation and future development of GPU transactional memory were summarized and prospected.

Key words
GPU, transactional memory, parallel computing

1 引言

随着对高性能计算的需求越来越大，GPU因其拥有比CPU更丰富的计算资源、线程资源和更高的内存带宽，被广泛应用于大数据处理和图形计算。

在大数据领域，有大量的GPU被服务商组织起来用于数据分析和数据处理。其中，有一类任务往往很少需要线程间的数据竞争，即使需要，一般也是以一种固定的方式对数据进行共享和使用。一般来说，GPU非常适合处理这类任务（如深度学习、图形计算）。对于这类任务来说，程序员可以预先估计访问或修改共享数据的模式，利用GPU提供的原子操作和同步操作进行数据的同步和保护。

然而，大数据分析和处理中的另一类任务需要动态地对共享数据进行并发访问和修改。例如，一个银行系统中可能存在多个线程同时访问或修改某段数据的情况，而这种访问和修改往往是动态的，是由输入的数据指定的。在这种情况下，想要保证程序的准确性，就需要程序员实现更加复杂的并行机制。对于GPU程序来说，由于其线程量巨大以及特殊的单指令多线程（single instruction multiple threads，SIMT）的运行机制，就需要程序员付出更多的努力才能写出正确的程序。

在CPU中也曾存在同样的问题，针对此，人们设计了事务性内存（transactional memory，TM）来简化程序员的工作。事务性内存提供了合适的API，将程序员从复杂的并行程序的设计中解放出来。同理，GPU也可以用同样的方式来解决这个问题，即GPU事务性内存。

本文首先介绍GPU和事务性内存，分析GPU事务性内存的重要性；然后对两类不同的GPU事务性内存——软件事务性内存（software transaction memory，STM）和硬件事务性内存（hardware transaction memory，HTM）的实现方案和重点问题进行分析和探讨；最后，对这些方案进行对比，并对未来的研究方向进行分析和展望。

2 GPU事务性内存介绍

2.1 GPU

GPU是现今非常流行的多核处理器之一，被广泛应用于高性能计算、大数据处理等领域。一种常见的GPU架构如**图1**所示[1]。GPU中有很多流多处理器（streaming multiprocessor，SM），每一个流多处理器中包含多个GPU核心，也称为流处理器（streaming processor，SP）。首先，在同一个SM中的多个SP共同使用寄存器文件以及多种缓存。其中比较重要的是共享内存和L1缓存，它们共同占用一块存储空间，同一个SM的不同SP可以通过共享内存来共享数据，程序员可以人为地划分共享内存的大小，剩下的空间将被用作L1缓存。其次，不同的SM共同拥有L2缓存、专门为常量所使用的常量内存和专门针对纹理信息优化过的纹理内存。最后，在GPU外有属于GPU的全局内存，CPU可以通过数据总线将数据传输到这个GPU的全局内存上。程序在运行时，每一个SP上可以跑一个线程，在GPU中，每32个线程组成一个线程束（warp），这个线程束的线程以SIMT的方式执行指令。具体而言，一个线程束中的32个线程在每一个时刻都是同时执行同一条指令的，每一个线程都拥有独立的地址空间。从线程可以使用的空间和层次考虑，每一个线程可以独

立地使用所属SM中的寄存器文件里的寄存器；通过共享内存与同SM中的其他线程进行数据交互；通过全局内存与其他线程共同使用共有的数据；在全局内存上拥有一段独立的地址空间，其被作为自己的本地内存（local memory）。除此之外，同一个线程束中的线程还可以通过GPU提供的原语进行数据交互。

GPU具有高计算能力、高并发能力、高访存速度的特性，利用这些特性实现的一些常用的数据结构比CPU实现的数据结构拥有更高的性能[2-6]。需要处理的数据被组织成各种数据结构，并被放置在GPU上等待被管理和使用。其中，在只读的情况下，GPU的性能比比CPU高很多。然而在读写混合的情况下，由于存在对数据的保护和同步的需求，GPU想要保持与CPU相同的性能比就会非常困难，并且，程序员编程的难度也大大增加。因此GPU迫切需要一个通用的并行程序设计的解决方案。

2.2 事务性内存

事务性内存是一种常用的并行程序设计的解决方案。针对并行程序，程序员可以通过使用锁结构（如mutex lock）、原子操作（如比较再交换（compare and swap，CAS））和内存屏障（memory barrier）来完成并行程序对共享数据的保护和使用。但是对于这些设计，尤其是复杂的并行程序，往往需要程序员认真考虑程序的正确性和效率，这个过程需要很长时间。为了简化这一过程，事务性内存作为一种并行程序设计的方式被提出。

事务（transaction）源自数据库系统，在数据库系统中，事务必须满足ACID的原则，即原子性（atomicity）、一致性（consistency）、隔离性（isolation）和持久性（durability）。

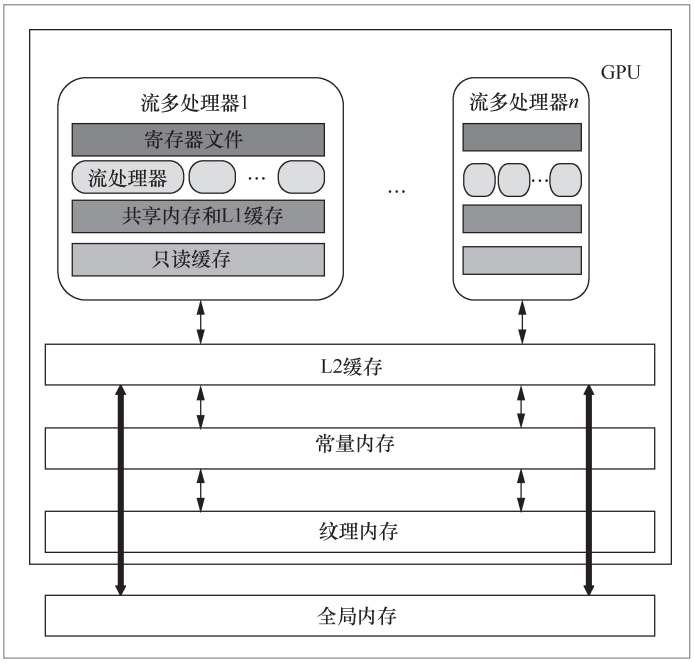


图1 一种常见的GPU架构

事务性内存正是借鉴了数据库中事务的概念实现的。事务性内存往往会提供一整套的API供程序员使用。一般而言，事务性内存提供的API至少包括TX_BEGIN和TX_END。TX_BEGIN代表事务的开始，TX_END代表事务的结束，在这之间的所有操作都被视为在同一个事务之中。图2展示了一个事务性内存的使用案例，从第2行到第6行，TX_BEGIN和TX_END包含的部分即一个事务的范围。在这个范围里分别读取了内存中保存的A的值，修改了内存

```
1.  void fun() {
2.    TX_BEGIN();
3.    newVal1 = read(A);
4.    write(B, newVal1);
5.    write(C, newVal2);
6.    TX_END();
7.    write(A, newVal3);
8.  }
```

图2 事务性内存的使用案例

中B和C的值。这3个操作在一定程度上满足了事务的原则①，即3个操作必然全都成功或是全都失败（原子性）；在事务提交之前，其他事务无法读到新的B和C（隔离性）；根据不同的事务性内存的设计，也可以保证在事务被提交前，A的值不会被其他人修改（一致性）。而对于在事务之外的操作，如第7句对A值的修改，一般会直接导致其他涉及A值的读写的事务被中止（abort）。

事务之中所有对内存的修改在一定程度上满足了ACID的要求（不同的系统和算法提供的事务性内存的标准可能略有区别）。这样，程序员通过使用这种简单方便的API，可以大大提高编程效率，同时也提高了程序的准确性。依据实现方式，事务性内存内部机制的实现被分为软件事务性内存和硬件事务性内存。目前，CPU上关于各种事务性内存的设计方案和策略以及如何高效地利用事务性内存解决实际问题的研究已经十分丰富[7-11]，因此本文将讨论的重点放在GPU上事务性内存的设计上。

2.3 GPU事务性内存

由于GPU使用SIMT的编程模型并且拥有大量的线程，其在复杂的并行程序上面临的问题比CPU更加复杂。因此，虽然GPU提供了原子操作和内存同步机制，但是面对复杂的并行程序，GPU上的事务性内存是十分必要的。

GPU事务性内存也同样根据其实现的方式被分为软件事务性内存和硬件事务性内存。但是相比于CPU事务性内存，其需要考虑的内容要更加贴合GPU本身的诸多性质。后文会详细分析在不同实现方式下，事务性内存实现所需要考虑的问题和解决它们的具体方法。

3 GPU STM

GPU STM是指利用现有的GPU提供的原语用软件方法实现的事务性内存。整体来说，虽然不同GPU STM的具体实现不同，但是其指导思想是一致的。

首先需要确定的是内存保护的粒度，粒度可以以字（4个字节）为单位，也可以以指定的步长为单位。系统会将自己所拥有的所有内存以设定好的粒度记录在一张被称为锁表（lock-table）的表里，该表的每一行对应一个单位的内存，记录了对应内存的版本号以及其是否正在被占用。

其次需要确定的是执行的粒度。具体来讲，需要决定执行一个事务的是一个线程，还是一个线程束。这样的一个粒度被称为执行单元。每一个执行单元在执行一个事务时，会有一组用于追踪所有内存的读和写的读集和写集，分别记录这个事务访问和修改的内存位置及其版本号。

事务在需要访问或修改一段内存时，会先访问锁表，确定该段内存是否被占用，在确定可用时，会将其记录在自己的读集或写集中，并获取相应的锁（根据方案的不同也可以不获取锁，只是确定其是否被占用）。接着，就可以执行想要执行的指令和操作（一般指内存的读和写）。这里的写操作是特殊的，有可能并不是直接写回内存，而是写在一个缓冲区里。不同的算法将数据写回内存的时机并不相同，一般来说，它们会将时机选择在进行写操作或提交的时候。但是不论何时写回，这个写回过程都是受到锁保护的。如果在写入内存之前发现锁表记录的版本号与自己的读/写集记录的版本号不一致或是锁表中显示该段内存已被占用，则该事务会被中止并回退

① 一般不包括持久性，因为事务性内存针对的是对内存的修改，而一般内存中保存的数据并不是持久的（宕机就会失去）

（roll back）。成功完成的事务被称为提交（commit）成功。

在以上的设计中，GPU STM和CPU STM的设计是类似的，但是在具体的策略方面，GPU STM需要一些单独的考虑。

3.1 执行粒度与锁问题

GPU STM的执行粒度一般有两种：以一个线程束为粒度[12]和以一个线程为粒度[13-14]。

在GPU中，一个线程束里的32个线程是以SIMT的方式运行的，也就是说，这32个线程每时每刻都在执行同一条指令。以线程束为粒度的STM可以用于一个线程束处理一个任务的应用。这个线程束通过共享内存来使用同一组读/写集。这种设计将一个线程束看作一个执行单元，回避了很多SIMT独有的问题，但是这种设计同时也限制了GPU STM的使用范围，即要求使用者必须以线程束为粒度来处理问题。

相比于以线程束为粒度，以一个线程为粒度的方案更加灵活。以一个线程为粒度意味着每一个线程会维护一组读集和写集，并自己负责这个线程的事务的处理和提交。这样的设计适用于更多的应用场景，但是以线程为粒度的STM需要解决可能由SIMT导致的死锁和活锁问题[13-14]。

3.1.1 死锁问题

首先，考虑一种常见的实现方式：自旋锁（spinlock）。如**图3**所示，每一个线程在访问一段内存前都会申请对应的锁（一般用CAS指令将lock-table的对应位置置为真），如果没有获得这个锁，则会不停地重复申请，直到获得这个锁才会继续前进，在使用完这段内存或是提交时再释放这个锁。这样的方案在CPU的设计中可能是可行的，但是在GPU中，设想这样一种情况：位于同一个线程束的线程1和线程2需要同时修改同一段内存，它们会同时运行这段代码，请求那段内存的锁。那么两个线程必然会有一个成功、一个失败。假设线程1成功得到了这个锁，那么线程2会由于获取锁失败而一直重复运行第一行。由于GPU是以SIMT的方式运行的，同一个线程束中的线程总是运行同一条指令，那么线程1会被迫随着线程2在第一句处空转，无法进入critical section进行针对这段内存的操作，自然也无法释放一直被线程2请求的那个锁，于是这里产生了死锁。

解决自旋锁的死锁问题很简单[13]，因为自旋锁产生的原因是本可以正常前进的线程被迫跟着不能正常前进的线程空转，所以只需要改变设计方案，使不能正常前进的线程跟着正常前进的线程空转即可。如**图4**所示，线程2会在第3行失败，但是由于这里只是一个if的判断语句，线程2会跟着线程1继续前进，线程1会进入第4行和第5行，而线程2会跟着线程1一起空

```
1.  while ((locked = CAS(&lock, 0, 1)) != 0);
2.  ...//critical section
3.  locked = 0;
```

图 3 通过自旋锁实现并发控制

```
1.  done = false;
2.  while (done == false) {
3.      if (CAS(&lock, 0, 1) == 0) {
4.          ...//critical section
5.          lock = 0;
6.          done = true;
7.      }
8.  }
```

图 4 解决了死锁的自旋锁实现方案

转。线程1在第5行释放了锁，在运行完第6行之后，结束任务，于是线程2可以回到第2行（线程1此时空转），然后在第3行重新申请已经被线程1释放的锁。

3.1.2 活锁问题

虽然**图4**的方案解决了死锁的问题，但是其只考虑了一个需求一个锁的情况，在实际使用时，STM往往会申请多个内存单元的锁，这时**图4**的算法又会带来活锁的问题。

通常来说，在一个线程需要多个锁时，如果它不能获得需要的全部的锁，那么它必须在发现不能获取全部的锁时，释放自己已经获得的锁，这是为了避免自己一直持有的锁和其他线程产生死锁。**图5**展示了需要同时申请两个锁的情况下的一种代码实现。假设一个事务拥有一个需要申请的锁的表单（数组locks，在这个例子中其长度为2），在第3行和第4行分别申请两个锁。一旦第二个锁获取失败，其就会释放第一个锁（第11行）。在CPU的设计中，这么做是有一定概率产生活锁的，因为可能存在两个线程，线程1的数组locks的内容为锁1和锁2，而线程2的数组locks的内容为锁2和锁1。这两个线程首先都执行了第3行，分别获得了锁1和锁2，而后又恰好同时执行了第4行，双方都发现自己无法继续获得锁，继而又同时执行了第11行，各自释放了自己获得的锁，进入重试（retry），然后在重试时又一次经历了这种获得锁和释放锁的过程。

```
1.  done = false;
2.  while (done == false) {
3.      if (CAS(&locks[0], 0, 1) == 0) {
4.          if (CAS(&locks[1], 0, 1) == 0) {
5.              ...//critical section
6.              locks[1] = 0;
7.              locks[0] = 0;
8.              done = true;
9.          }
10.         else {
11.             locks[0] = 0;
12.         }
13.     }
14. }
```

图 5 申请两个锁的情况下的实现方案

时间上的巧合使得线程1和线程2不断地重复获得锁和释放锁的过程。这种活锁的情况对于CPU来说是可能发生的，而在GPU上由于SIMT的特性（同一个线程束内不同的线程同时执行同一个指令），导致这种同时性成为必然，即在GPU编程中，这种设计必然会导致活锁问题。

为了解决活锁问题，不同的研究者给出了不同的解决方案。GPU-STM[13]采用了一种复杂的锁排序的机制来避免活锁的产生。简单来说，产生活锁的前提是两个线程同时获得了对方需要的锁，然后发现无法进一步获取被对方拿在手里的其他锁，因此又同时释放了自己已经拿到的锁，进入这样一个循环状态。而如果线程获取锁的顺序按照一种固定的逻辑，就不会存在这样的情况了。如**图5**所示的例子，如果两个线程需要锁的顺序都是锁1、锁2，就不会出现活锁了。

尽管锁排序解决了活锁的问题，但是排序的代价是很大的，因此Shen Q等人[14]提出了另一种解决方案，为锁设计了优先级。在他们的设计中，拥有较小线程号的事务拥有更高的优先级，它们能够从拥有较低优先级的事务那里将锁“抢”过来。于是，在**图5**的情境中，线程1和线程2首先分别获得了锁1和锁2，在接下来的一步中，线程1的优先级比线程2高，因此线程1可以将锁2从线程2那里“抢”过来，现在线程1拥有了全部的锁，线程2没有获得锁，线程2会陪着线程1空转，待线程1完成自己的事务释放了两个锁之后，线程2才会重新开始自己申请锁的流程。

3.2 执行策略

与CPU TM相同，GPU STM也需要考虑执行策略，其内容主要包括版本管理（version management）、冲突检测（conflict detection）、重试和回退，大部分与GPU TM相关的文章对这些策略进行了一定的分析和讨论[12-15]。

3.2.1 版本管理

版本管理一般包括积极的版本管理（eager version management）和消极的版本管理（lazy version management），其主要决定了事务在何时将修改的内容真正地写回内存。

积极的版本管理指的是事务在写回数据的策略上是积极的，具体来说，在积极的版本管理中，事务会立即将自己修改的内容写回内存。在积极的版本管理的流程中，事务首先申请获得要访问的内存的锁以及版本号（根据一致性保护程度的不同，可以在需要用到时申请锁，也可以在事务的开头一次性全部申请），然后进行自己的操作，读和写都在已经得到保护的内存上进行，其中写操作还需要将旧值记录在一个被称为undo-log的缓存中。在提交时，事务将释放所有的锁，事务正常结束，到此即可视为提交成功。当出现冲突（如获取锁失败）或宕机时，根据undo-log的内容进行回退。

消极的版本管理指的是事务执行写操作时，并不会立刻将其写回内存，而是先写在一处缓存中，在提交时一次性将所有的内存改变写回内存。也就是说，首先，事务检查要访问的内存是否被占用。然后，对于读操作，在读取数据的同时，读取其版本号并记录在读集中；对于写操作，将要写的数据写入一段被称为redo-log的缓存中，并记录其版本号。在所有的操作结束后，进入提交阶段，提交阶段首先会根据写集和读集获取相关内存的版本号，并申请写集中需要写回的内存的锁，比较读/写集记录的版本号与新获得的版本是否匹配，在匹配成功且获得了需要的锁的情况下，事务会将缓存中的内容写回内存，然后将锁释放。

这两种版本管理的设计在GPU上都是可行的。总体来说，积极的版本管理在面对冲突较低的情形时拥有更高的效率，因为积极的版本管理回退的代价要高于正确提交的代价，而消极的版本管理则相反，在高竞争的场景中效率更高。

在以上的设计中，为了能够充分利用GPU的存储特性，如果容量足够，redo-log或undo-log会尽量放置在共享内存或L1缓存中，而lock-table等全局共用的数据则被安置在全局内存中。

3.2.2 冲突检测

冲突检测包括积极的冲突检测和消极的冲突检测。积极的冲突检测会选择在尽可能早的时间点进行冲突检测（或者说版本号的检测），而消极的冲突检测在提交时才进行冲突检测。

从理论上说，积极的冲突检测可以避免线程做很多无用功，并尽早地进行中止和回退，但是这会带来一个问题，即GPU是以SIMT的方式运行的，一个线程的中止和回退并不代表其真的可以重新开始这个事务，它必须要空转，直到同一个线程束里的其他线程完成自己的事务或也进入回退。因此这种设计往往更适合以线程束为粒度的TM设计，否则就需要让同一个线程束里的所有线程处理的事务尽可能地有相似的行为。

对于消极的冲突检测来说，当其检查到冲突时，事务已经处于提交状态，此时写集的内容大多已经被写过一次，这意味

着这些操作都会变成无效的内容，会造成资源的浪费。但是由于检测的次数少，在消极的冲突检测中，出现冲突的次数会减少，回退的次数也会减少。

版本管理的方案也会影响冲突检测的策略。积极的版本管理在写集上的冲突检测必然也是积极的，因为其为写集上的内容申请锁的条件之一就是要确认版本号。但是积极的版本管理中在读集上的冲突检测既可能是积极的，也可能是消极的，因为读集在某些情况下是可以不申请锁的。因此，读集里的内容可以在需要的时候立刻检查版本（积极的冲突管理），也可以在最后提交时再检查版本（消极的冲突管理）。

3.2.3 重试和回退

对于锁的访问，笔者使用了诸多的方法来保证不会出现死锁或是活锁问题。接下来需要讨论的是，在真正遇到无法获得锁的情况下，尤其在积极的版本管理的情况下，是选择重复尝试获取锁，还是选择中止并回退。一般而言，回退会直接放弃线程之前已经做完的内容，而重试意味着还有机会将当前的事务继续完成。但是也有相关研究显示[16]，在GPU上，回退比重试有更高的性能。原因是获取锁的行为本质上是针对GPU全局内存中lock-table里表示目标内存的条目的，通过CAS操作将其表示是否被占用的标志位赋值为真，而这种原子操作会由GPU特定的硬件单元来完成，这意味着重复进行这种CAS操作会频繁地占用GPU资源，导致其他线程的CAS操作无法尽快完成。因此，理论上调用原子操作的次数越少越好。

4 GPU HTM

GPU HTM是指从硬件的角度对GPU的体系结构进行改进，从而实现的事务性内存。GPU HTM涉及的策略问题与GPU STM涉及的策略问题是类似的，区别在于GPU HTM旨在使用硬件的方法解决这一问题。GPU HTM一般需要对GPU的体系结构做一定程度的更改[17-19]。

一般来说，为了实现GPU HTM，对硬件的设计应当着重于解决这几个问题：如何在SIMT的硬件模型下解决事务控制流的回退问题，如何设计读集和写集，如何完成事务的提交。

4.1 SIMT硬件模型下事务的回退

GPU要支持事务性内存，有一个问题是绕不开的，即如何让GPU支持事务的回退。这与CPU事务的回退有所不同，在以线程为粒度的事务性内存中，同一个线程束中并行的复数个事务中，可能只有一部分线程的事务失败需要回退，而GPU是以SIMT的方式来执行指令的，这里就会产生控制流的分歧（divergence）。值得参考的是，GPU在处理判断等语句时也会产生类似的分歧，因此GPU HTM一般会采用与之类似的方法来使GPU的硬件支持事务的回退。

4.1.1 分歧和回退处理的硬件基础

在GPU的程序中，经常会出现由判断或循环导致的分歧，同一个线程束中的不同线程需要运行不同分支上的指令，由于GPU的SIMT的执行方式，这些线程在不同分支上的指令不得不被串行执行，即在一部分线程运行其中一个分歧上的指令时，其他线程也必须跟着这些线程空转。为了减少这种串行带来的影响，GPU使用SIMT指令栈[20]来安排和调度指令，并负责控制每一个线程在分歧结束时的跳转位置。

SIMT指令栈的实现如**图6**所示。SIMT栈中保存了汇合地址、下一条指令地址和在该地址活跃的线程（活跃线程对应的比特置为1）。GPU每从栈中出栈一个条目，就会根据该条目令相应的活跃线程执行相应的指令，令非活跃线程空转，并根据情况将下一条指令的条目入栈。**图6**左侧所示为将栈首标记的条目出栈，右侧所示为执行的情况[21]。

图6展示了一个只有4个线程的线程束，其在A处进行了一个判断，然后第1线程和第4线程进入了B，第2线程和第3线程进入了C，在执行完B或C后，4个线程于D处汇合。首先如**图6**（a）所示，4个线程执行A，栈中的第一个条目包括了这条指令的地址以及相应的活跃线程。然后该条目出栈，令4个线程执行A。由于该语句是一条判断，会再入栈3个条目，结果如**图6**（b）所示，按入栈先后分别表示：在分支结束后最终4个线程会共同执行D；有两个线程进入了C分支执行C处指令，它们将会在D处与其他线程汇合；有两个线程进入了B分支执行B处指令，它们也会在D处与其他线程汇合。接下来依次将3个条目出栈，并分别执行语句B、C、D。以**图6**（b）将要出栈的表示B的条目为例，将该条目出栈，执行B，理论上应该再入栈一条表示紧接着B之后那条指令的条目，但是由程序流程图可知，那条指令即D，为汇合处的指令，因此不需要再将该条目入栈了。**图6**（c）和**图6**（d）同理，在将**图6**（d）中的条目出栈后，流程结束。

4.1.2 利用SIMT栈实现事务回退

在GPU HTM的设计中，往往需要对SIMT指令栈的细节进行更改，使其能够在事务回退时正确地安排需要回退的线程回退到指定的指令位置，并使不需要回退的线程保持空转[17-18]。

利用SIMT指令栈实现事务的回退如**图7**所示，该指令栈是由Fung W W L等人[17]设计的能够支持回退的SIMT指令栈。在这个例子中，该段程序使用了一个事务，该事务除了开始与提交外，只包括了一条语句A，在事务结束之后，又执行了一条语句B。其中，事务在提交时有一定的可能性会失败，提交失败的线程会回退到事务的开始处，并重新执行。在改进后的SIMT指令栈中设置了几种状态，其中N（normal）代

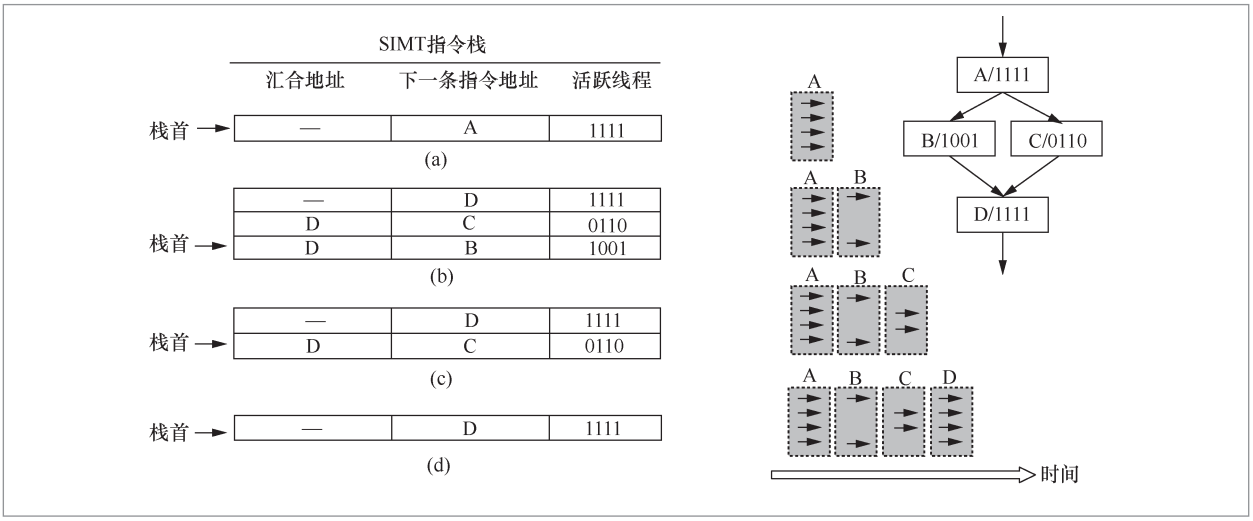


图 6 SIMT 指令栈的实现

表一般状态，R（transaction retry）代表事务重试，T（transaction top）代表事务执行到的最前面的位置。

当执行TX_BEGIN时（如**图7**（a）所示），SIMT指令栈会保留TX_BEGIN的条目，并入栈两个特殊的条目（如**图7**（b）所示），按时间顺序分别代表回退线程的起始地址和事务真正要执行的地址，因此状态分别为R和T，其中前者的活跃线程为空（因为目前没有线程要回退），后者的活跃线程应与TX_BEGIN的活跃线程相同，这里为4个线程。当执行到TX_COMMIT时，第2线程、第3线程提交失败（如**图7**（c）所示），那么在R条目中这两个线程对应的比特就会被置为1（如**图7**（d）所示）。接着为了执行R条目，SIMT栈会复制一份除状态为T外，其他与R条目完全相同的条目入栈，并将R条目的活跃线程清空。这样，新的T条目就会被作为回退线程的起始（如**图7**（e）所示）。在回退线程也成功提交后（如**图7**（d）所示），剩下的R条目回退线程为空，这就代表没有线程需要回退，该R条目也将被出栈（如**图7**（f）所示）。此时，可以根据之前保留的TX_BEGIN条目确定该事务的活跃线程，并入栈相应的TX_COMMIT之后的指令地址（如**图7**（g）所示）。

4.2 读集和写集的硬件设计

GPU HTM和GPU STM一样，需要使用读集和写集来记录自己访问或修改的内存条目，同时，读集和写集也可以作为

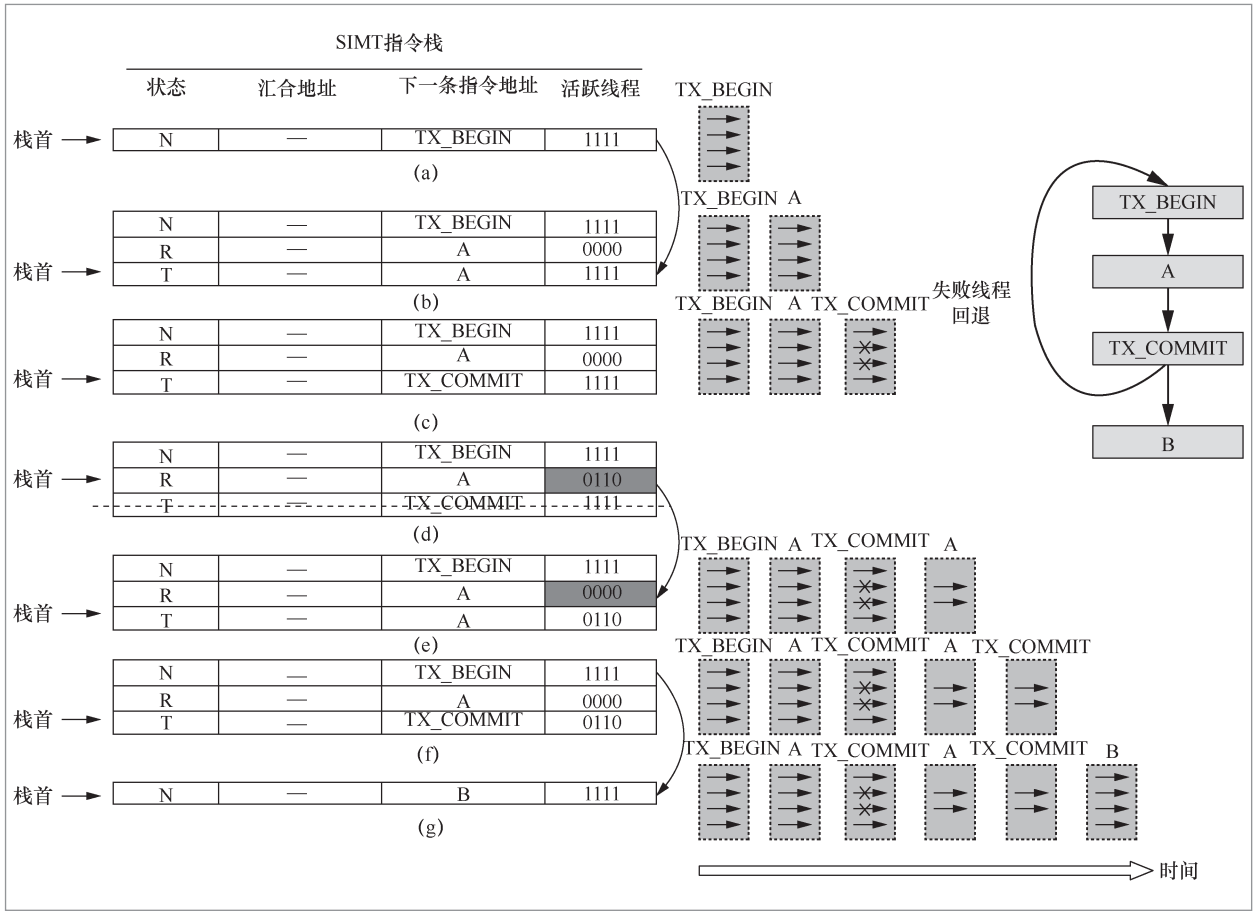


图 7 利用 SIMT 指令栈实现事务的回退

redo-log或undo-log来实现不同的版本控制。因此需要在GPU中开辟出一个空间作为读集和写集。

考虑到事务性内存的粒度和第2.1节介绍的GPU体系结构中各层级的存储空间，GPU HTM往往以L1缓存或共享内存为首选的读/写集的位置，然后在提交时，将读/写集中的内容按照规则提交到L2缓存中，进而提交到内存里。这样的设计一般符合redo-log的模式，即将写集视为redo-log。但是在实际情况中，上层的缓存一般是有限的，因此往往只是作为对读/写集的缓存，真正的存储位置为全局内存中线程所拥有的本地内存（local memory）。

4.3 GPU HTM的冲突检测和提交

GPU HTM的冲突检测和提交是通过在GPU中加入新的硬件实现的。如**图8**所示，这是Fung W W L等人[17-18]提出的方案，他们通过在GPU中加入日志单元（log unit）和提交单元（commit unit）来实现HTM的事务的冲突检测和提交。他们的方案采用了基于值的冲突检测[17]，即冲突检测中不使用版本号，而使用数据的值。GPU HTM中的读集和写集分别记录了一个事务中读取的值和更改的值。在提交时，日志单元负责收集每一个线程的读集和写集中的记录，并传送给提交单元，提交单元会将收到的写集里的值写回内存，对比读集里的值与对应的全局内存，若相同，则通过了冲突检测。

提交单元用于提交事务并将修改的值写回内存的单元，为了能够并行提交尽量多的数据，一般将内存分成几段，每一段对应一个提交单元。每一个提交单元维护一个队列，按顺序以一种流水线的方式分阶段同时处理传递过来的多个事务的记录，为了提高效率，提交单元一般会在提交

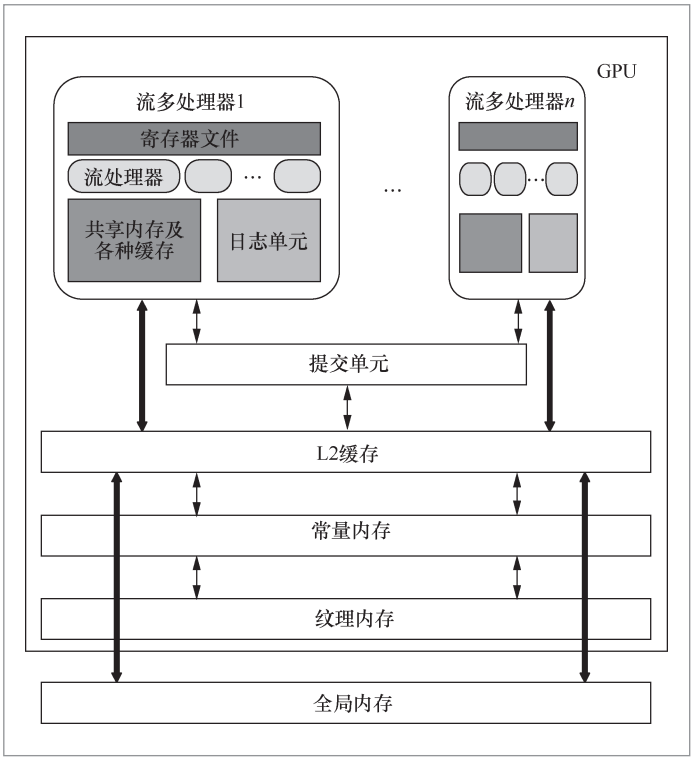


图 8 添加了新硬件的 GPU HTM 的架构

的最后同时写回多个事务的记录。处理流程大致分为5个步骤：第一步，提交单元获得日志单元传输过来的记录，检查其中读集的内容和内存中的内容是否一致，由于这里的检查一般需要一些时间，因此提交单元仅仅发起检查，而不用等待检查结束就开始下一步。第二步，检查同时提交的事务中是否存在冒险，即两个同时提交的事务是否存在读写的冲突。第三步，等待第一步的检查结束，对于产生冒险的事务，会待其他事务提交之后再根据更新后的内存重新进行检查。第四步，由于每一个提交单元负责一部分内存，因此提交单元要将自己对某事务的预计提交结果广播出去，并回收其他提交单元的预计提交结果，如果每个提交单元都确定可以提交，则通过这一步。第五步，将可以提交的事务提交，释放相关内存，并通知相关的

线程提交是否成功。

5 GPU STM和GPU HTM的性能分析与比较

5.1 性能分析

在GPU的性能测试上，事务性内存常见的测试集包括Bank、哈希表等[17]，由于不同的工作负载（如读写比例、冲突比例、线程数目、内存大小）等原因，其性能表现有很大的差异。

在一般的GPU STM研究中，研究者多会将GPU STM与CPU TM进行性能比较，以试图体现他们的设计比一般的CPU设计有更高的性能和更好的应用前景。参考文献[14]比较了PR-STM（GPU STM）和TinySTM[20]（CPU STM）的性能差异，在前者使用512个线程，后者使用8个线程，测试集为哈希表的情况下，相比于TinySTM，PR-STM性能更高，是TinySTM性能的1~5倍，并且随着事务大小（transaction size）的增加，PR-STM的性能优势逐渐增大。其中，PR-STM视情况大约能达到8×10^6 TX/s的吞吐率，相对地，另一个GPU STM的设计GPU-STM[13]能达到6×10^6 TX/s，而基于CPU STM设计的TinySTM能达到2×10^6 TX/s。考虑到GPU拥有更多的线程、更高的带宽，这种性能差异并不令人满意。在lightweight STM[15]（GPU STM）与CPU TM的对比中，lightweight STM性能也只能达到CPU性能的5~7倍，并且实验中CPU为8核，GPU使用的是能达到最佳性能的配置。考虑到一般情况下CPU HTM比CPU STM拥有更高的性能，这个性能差异会更小。

由于目前的GPU HTM的设计[17-19]大多是在模拟器（GPGPU-Sim[22]）中实现的，因此对GPU HTM的性能分析都停留在GPU HTM不同方案在不同情况下的速度差异，无法与GPU STM或CPU TM（包括STM和HTM）进行比较，故在本文中缺少这个方面的数据。

5.2 GPU STM和GPU HTM的对比

GPU STM和GPU HTM各有优劣，其主要区别是实现方法不同，但是它们想要实现的目标、实现目标所需要的策略是一致的。不同的实现方法决定了它们的性能和实现难易程度的不同。**表1**总结了它们的异同。

从目标上来说，GPU STM和GPU HTM想要达到的目标是一致的，即提供一组满足程序员需求的API，使他们能够避免烦琐的并发设计、复杂的锁的实现，以及使用原子操作和同步操作。

从需要考虑和关注的问题上来说，GPU STM和GPU HTM是类似的，都需要解决SIMT带来的执行的问题，包括活锁、死锁和回退，读集和写集的实现，版本管理，冲突检测等策略选择。

从性能上来说，GPU HTM比GPU STM拥有更大的潜力。因为理论上硬件的设计更容易达到更好的效果。但是目前的情况是GPU HTM大多是在模拟器上实现的。

从实现难易程度上来说，GPU HTM比GPU STM复杂得多，因为GPU的硬件更新换代很快，GPU HTM也需要跟着硬件设计的改变而改变，除此之外还可能要有相适应的编译工具。而GPU STM理论上不会遇到这些问题，程序员甚至可以在没有现成的GPU STM的情况下，自己写一个简易的版本在程序中使用。

表1 GPU STM 和 GPU HTM 的异同

对比项	GPU STM	GPU HTM
实现方法	软件	硬件
目标	提供减少并行编程难度的API	提供减少并行编程难度的API
需要考虑和关注的问题	解决SIMT导致的死锁、活锁问题； 版本管理、冲突检测	使SIMT栈支持回退； 版本管理、冲突检测
性能	受制于软件设计	潜力更大，但现在多为模拟器实现
实现难易程度	相对容易	需要对硬件进行设计修改

6 GPU事务性内存的总结和展望

就目前的发展情况来看，GPU事务性内存还不够成熟。由于GPU本身的特性，GPU事务性内存的设计和使用受到诸多因素的制约。

首先，GPU的特性会导致事务性内存的设计面临诸多挑战。由于GPU的SIMT的执行方式，事务回退时会产生高额的代价；由于GPU具有大量的线程，在面对相同的处理数据时，GPU能够并发地处理更多的数据，但也会遇到更多的竞争，产生回退的可能性更大；由于GPU的访存模式，GPU在处理规则的数据时拥有更高的性能，但一般需要使用事务性内存的应用的数据大部分是动态的，这并不十分适合使用GPU进行处理。

其次，从实现方案和性能的角度来说，GPU事务性内存不同的实现方法会造成不同程度的额外开销。具体来说，尽管GPU STM可以给使用者带来便利，但是其理论上的性能并不会高于使用者自己手动设计的并发策略。并且，不同的事务性内存的策略往往适用于不同的应用，因此，使用者依然有必要仔细考虑自己的应用更适合使用哪种方案的事务性内存，甚至有必要根据自己的应用对GPU STM进行专门的优化。而GPU HTM因为是硬件的设计，理论上可以达到比STM或使用者自己手动实现的方案更好的性能，但是GPU HTM提供给使用者的选项也会相应地减少，使用者依然要考虑已有的GPU HTM方案是否适合自己的应用场景。

尽管GPU事务性内存在诸多方面有所限制，但是其依然具有十分重要的潜在价值。GPU事务性内存往往应用于并发数据结构的设计，因此它的一个潜在的应用场景是GPU数据库系统。越来越多的研究试图利用GPU来加速数据库运算以及利用GPU实现高效的底层数据结构[2-5]，而这也催生了对事务性内存等通用的并发编程解决方案的需求。

GPU事务性内存的发展还有很长的路要走，现在虽然已有针对GPU HTM的研究，但是门槛较高，研究者需要足够精通GPU的硬件以及相关的模拟器。相比而言，针对GPU STM的研究相对容易，但是暂时缺乏非常高效和通用的版本。

总而言之，GPU事务性内存仍然有很大的发展空间。

参考文献:

[1] NVIDIA. CUDA C++ programming guide[Z]. 2020.

[2] YAN Z F, LIN Y Z, PENG L, et al.

Harmonia: a high throughput B+tree for GPUS[C]//The 24th Symposium on Principles and Practice of Parallel Programming. New York: ACM Press, 2019: 133–144.

[3] SHAHVARANI A, JACOBSEN H A. A hybrid B+tree as solution for in–memory indexing on CPU–GPU heterogeneous computing platforms[C]//The 2016 International Conference on Management of Data. [S.l.:s.n.], 2016: 1523–1538

[4] KRZYSZTOF K. B+–tree optimized for GPGPU[C]//OTM Confederated International Conferences. [S.l.]: Springer, 2012: 843–854.

[5] JORDAN F, ANDREW W, KEVIN S. Accelerating braided B+tree searches on a GPU with CUDA[C]//The 2nd Workshop on Applications for Multi and Many Core Processors: Analysis, Implementation, and Performance. [S.l.:s.n.], 2011: 1–11.

[6] ZHANG W H, YAN Z F, LIN Y Z, et al. A high throughput B+tree for SIMD architectures[J]. IEEE Transaction on Parallel and Distributed Systems, 2020, 31(3): 707–720.

[7] HERLIHY M, ELIOT J, MOSS B. Transactional memory: architectural support for lock–free data structures[C]//The 20th Annual International Symposium on Computer Architecture. [S.l.:s.n.], 1993: 289–300.

[8] SHAVIT N, TOUITOU D. Software transactional memory[C]//The 14th ACM Symposium on Principles of Distributed Computing. New York: ACM Press, 1995: 204–213.

[9] LOMET D B. Process structuring, synchronization, and recovery using atomic actions[C]//The ACM Conference on Language Design for Reliable Software. New York: ACM Press, 1977: 128–137.

[10] HARRIS T, ADRIÁN C, UNSAL O S, et al. Transactional memory: an overview[J]. IEEE Micro, 2007, 27(3): 8–29.

[11] WANG X, ZHANG W H, WANG Z G, et al. Eunomia: scaling concurrent search trees under contention using HTM[C]//The 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York: ACM Press, 2017: 385–399.

[12] CEDERMAN D, TSIGAS P, CHAUDHRY M T. Towards a software transactional memory for graphics processors[C]//Eurographics Conference on Parallel Graphics & Visualization. [S.l.:s.n.], 2010: 121–129.

[13] XU Y L, WANG R, GOSWAMI N, et al. Software transactional memory for GPU architectures[J]. Computer Architecture Letters, 2014, 13(1): 49–52.

[14] SHEN Q, SHARP C, BLEWITT W, et al. Priority rule based software transactions for the GPU[C]//The European Conference on Parallel Processing. [S.l.:s.n.], 2015: 361–372.

[15] HOLEY A, ZHAI A. Lightweight software transactions on GPUs[C]//The 43rd International Conference on Parallel Processing. [S.l.:s.n.], 2014: 461–470.

[16] AWAD M A, ASHKIANI S, JOHNSON R, et al. Engineering a high–performance GPU B–tree[C]//The 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel. [S.l.:s.n.], 2019: 145–157.

[17] FUNG W W L, SINGH I, BROWNSWORD A, et al. KILO TM: hardware transactional memory for GPU architectures[J]. IEEE Micro, 2012, 32(3): 7–16.

[18] FUNG W W L, AAMODTT M. Energy efficient GPU transactional memory via space–time optimizations[C]//The 46th Annual International Symposium on Microarchitecture. New York: ACM Press, 2013: 408–420.

[19] SUI C, LU P, SAMUEL I. Accelerating GPU hardware transactional memory with snapshot isolation[C]//The 44th Annual International Symposium on Computer Architecture. New York: ACM Press, 2017: 282–294.

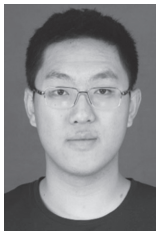
[20] FELBER P, FETZER C, RIEGEL T. Dynamic performance tuning of word–

based software transactional memory[C]// The 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York: ACM Press, 2008: 237–246.

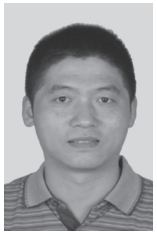
[21] FUNG W W L, SHAM I, YUAN G L, et al. Dynamic warp formation: efficient MIMD control flow on SIMD graphics hardware[J]. ACM Transactions on Architecture and Code Optimization, 2009, 6(2).

[22] BAKHODA A, YUAN G L, FUNG W W L, et al. Analyzing CUDA workloads using a detailed GPU simulator[C]//2009 IEEE International Symposium on Performance Analysis of Systems and Software. Piscataway: IEEE Press, 2009: 163–174.

作者简介



林玉哲（1996– ），男，复旦大学软件学院硕士生，主要研究方向为GPU、并行计算、事务性内存等。



张为华（1974– ），男，复旦大学软件学院教授，主要研究方向为编译、体系结构、并行计算、系统软件等。

收稿日期： 2020–05–09