# 第十章：异构系统的 Consistency 和 Coherence

这是专业化的时代。今天的服务器、移动和桌面处理器不仅包含传统的 CPU，还包含各种类型的加速器。加速器中最突出的是图形处理单元 (GPU)。其他类型的加速器，包括数字信号处理器 (DSP)、AI 加速器（例如 Apple 的神经引擎、Google 的 TPU）、加密加速器和现场可编程门阵列 (FPGA) 也变得越来越普遍。

然而，这种异构架构带来了可编程性挑战。如何在加速器内和加速器之间进行同步和通信？而且，如何有效地做到这一点？一个有希望的趋势是公开跨 CPU 和加速器的全局共享内存接口 (global shared memory interface)。回想一下，共享内存不仅通过为通信提供直观的 load-store 接口来帮助提高可编程性，而且还有助于通过对程序员透明的缓存获得局部性的好处。

CPU、GPU 和其他加速器可以紧密集成并实际上共享相同的物理内存，就像移动片上系统中的情况一样，或者它们可能具有物理上独立的内存，运行时提供共享内存的逻辑抽象。除非另有说明，我们假设前者。如图 10.1 所示，每个 CPU、GPU 和加速器都可能有多个核心，每个核心都有私有 L1 和共享 L2。CPU 和加速器还可以共享内存端的最后一级高速缓存 (last-level cache, LLC)，除非另有说明，否则该高速缓存不包含在内。（回想一下，内存端 LLC 不会造成 coherence 问题）。LLC 还用作片上内存控制器。

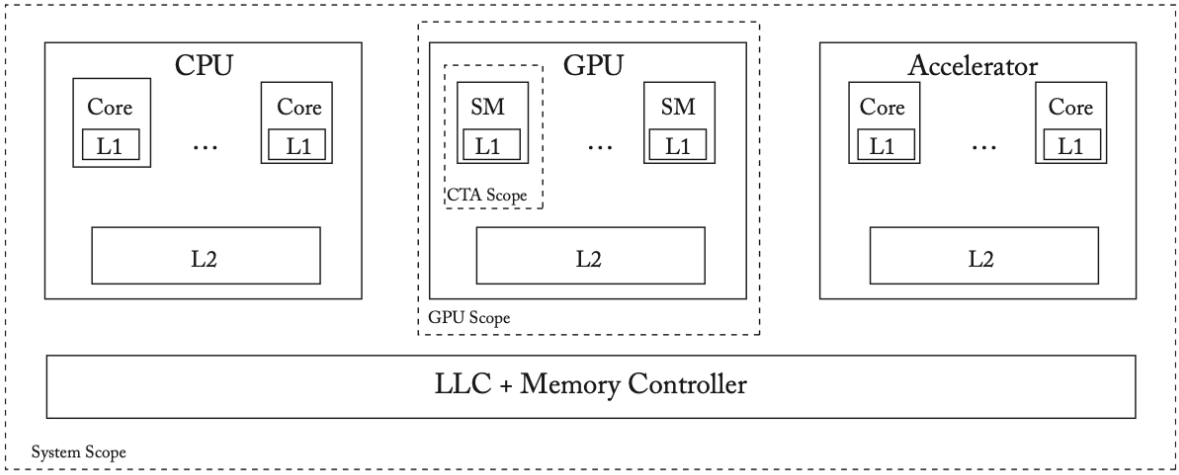


Figure 10.1: System model of a heterogeneous system-on-chip containing a CPU, GPU, and accelerator. SM refers to a streaming multi-processor; CTA refers to cooperative thread array.

共享内存会自动引出问题。Consistency model 是什么？如何（有效地）执行 consistency model？特别是，加速器 (L1) 内以及 CPU 和加速器 (L1 and L2) 之间的缓存如何保持 coherent？

在本章中，我们将讨论在这个快速发展的领域中这些问题的潜在答案。我们首先研究加速器内的 consistency 和 coherence，重点是 GPU。然后，我们考虑加速器和 CPU 之间的 consistency 和 coherence。

## 10.1 GPU Consistency 和 Coherence

我们通过简要总结早期的 GPU 架构及其编程模型来开始本节。这将帮助我们了解此类 GPU 中关于 consistency 和 coherence 的设计选择，这些设计主要针对图形工作负载。然后，我们讨论使用类似 GPU 的架构在所谓的通用图形处理单元 (GPGPU) [5] 中运行通用工作负载的趋势。具体来说，我们讨论了 GPGPU 对 consistency 和 coherence 提出的新要求。然后，我们详细讨论了最近为满足这些要求而提出的一些建议。

## 10.1.1 早期 GPU：架构和编程模型

早期的 GPU 主要是针对令人尴尬的并行图形工作负载量身定制的。粗略地说，工作负载涉及独立计算构成显示器的每个像素。因此，工作负载的特点是数据并行度非常高，数据共享、同步和通信程度低。

GPU 架构
为了充分利用并行性，GPU 通常有数十个核心，称为流式多处理器 (SM)（注1），如图 10.1 所示。每个 SM 都是高度多线程的，能够运行大约一千个线程。映射到 SM 的线程共享 L1 高速缓存和本地暂存器内存（未显示）。所有的 SM 共享一个 L2 缓存。

原书作者注1：NVIDIA 用语中的 SM。在 AMD 用语中也称为计算单元 (CU)。
为了分摊为所有这些线程获取和解码指令的成本，GPU 通常在称为 warp 的组中执行线程。（注2）Warp 中的所有线程共享程序计数器 (PC) 和栈，但仍然可以独立执行使用掩码位的线程特定路径，指定 warp 中哪些线程是活跃的，哪些线程不应该执行。这种并行方式被称为“单指令多线程” (SIMT)。因为 warp 中的所有线程共享 PC，传统上 warp 中的线程被调度在一起。然而，最近，GPU 开始允许 warp 中的线程拥有独立的 PC 和栈，从而允许独立线程调度 (Independent Thread Scheduling)。在接下来的讨论中，我们假设线程可以独立调度。

原书作者注2：NVIDIA 用语中的 Warp。在 AMD 用语中也称为 Wavefront。
由于图形工作负载不会频繁共享数据或同步（同步和通信通常很少以较粗的粒度级别发生），早期的 GPU 选择不为 L1 缓存实现硬件 cache coherence。（他们没有强制 SWMR 不变量。）硬件 cache coherence 的缺失使得同步对于程序员来说是一个更棘手的前景，正如我们接下来看到的那样。

GPU 编程模型
与 CPU 指令集架构 (ISA) 类似，GPU 也具有虚拟 ISA。例如，NVIDIA 的虚拟 ISA 被称为并行线程执行 (PTX)。此外，还有更高级的语言框架。两个这样的框架特别受欢迎：CUDA 和 OpenCL。这些框架中的高级程序被编译成虚拟 ISA，然后在内核 (kernel) 安装时 (installation time) 被翻译成本地二进制文件。内核是指由 CPU 卸载到 GPU 上的工作单元，通常由大量软件线程组成。

GPU 虚拟 ISA 以及语言框架选择通过称为范围 (scope) [15] 的线程层次结构 (thread hierarchy) 向程序员公开 GPU 架构的层次结构。与所有线程“平等”的 CPU 线程相比，来自内核的 GPU 线程被分组到称为协作线程阵列 (CTA) 的集群中。（注3）CTA 范围 (CTA scope) 是指来自同一 CTA 的线程集。这样的线程保证映射到同一个 SM，因此共享同一个 L1。因此，CTA 范围也隐含地引用了所有这些线程共享的内存层次结构级别 (L1)（注4）。GPU 范围 (GPU scope) 是指来自同一 GPU 的线程集。这些线程可能来自于 GPU 中相同或不同的 CTA。所有这些线程共享 L2。因此，GPU 范围隐含地指的是 L2。最后，系统范围 (system scope) 是指整个系统中所有线程的集合。这些线程可以来自构成系统的 CPU、GPU 或其他加速器。所有这些线程都可以共享一个缓存 (LLC)。因此，系统范围隐含地指的是 LLC，如果没有共享 LLC，则指统一共享内存 (unified shared memory)。

原书作者注3：PTX 用语中的 CTA。在 OpenCL 中也称为 Workgroup，在 CUDA 中称为线程块。
原书作者注4：和本地暂存器内存（如果存在的话）
为什么要向软件公开线程和内存层次结构？在没有硬件 cache coherence 的情况下，这允许程序员和硬件协同高效地实现同步。具体来说，如果程序员确保两个同步线程在同一个 CTA 内，则线程可以通过 L1 缓存高效地同步和通信。

属于同一 GPU 内核的不同 CTA 的两个线程如何同步？有点令人惊讶的是，早期的 GPU consistency model 并没有明确允许这样做。但在实践中，程序员可以通过绕过 L1 并在共享 L2 上同步的特制 load 和 store 来实现 GPU 范围的同步。不用说，这在程序员中引起了很大的混乱[8]。在下一节中，我们将通过示例详细探讨 GPU consistency。

GPU Consistency
GPU 支持 relaxed memory consistency model。与 relaxed consistency CPU 一样，GPU 仅强制执行程序员指示的内存顺序，例如，通过 FENCE 指令。

但是，由于 GPU 中缺乏硬件 cache coherence，FENCE 指令的语义是不同的。回想一下，多核 CPU 中的 FENCE 指令确保在执行 FENCE 之前的 load 和 store，相对于所有线程，在 FENCE 之后的 load 和 store 之前。GPU FENCE 提供类似的语义，但仅针对属于同一 CTA 的其他线程。一个推论是 GPU store 缺乏原子性。回想一下，store 原子性（第 5.5.2 节）要求所有其他线程在逻辑上立刻看到一个线程的 store。然而，在 GPU 中，一次 store 可能在其他线程可见之前、对属于同一 CTA 的某个线程可见。

考虑表 10.1 中显示的消息传递示例，其中程序员打算让 load Ld2 读取 NEW 而不是旧值 0。如何确保这一点？

Table 10.1: Message passing example. T1 and T2 belong to the same CTA.

| Thread T1 | Thread T2 | Comments |
|---|---|---|
| St1: St data = NEW;<br>FENCE;<br>St2: St flag = SET; | Ld1: Ld r1 = flag;<br>B1: if (r1 ≠ SET) goto Ld1;<br>FENCE;<br>Ld2: Ld r2 = data; | // Initially data and flag are 0.<br><br><br>Can r2==0? |

首先，请注意，如果没有两条 FENCE 指令，Ld2 可以读取旧值 0——宽松的 GPU 可能会乱序执行 load 和 store。

现在，让我们假设两个线程 T1 和 T2 属于同一个 CTA，因此映射到同一个 SM。在微架构级别，GPU FENCE 的工作方式与 XC 中的 FENCE 类似，正如我们在第 5 章中讨论的那样。重新排序单元 (reorder unit) 确保 Load/Store -> FENCE 和 FENCE -> Load/Store 的顺序被强制执行。因为 T1 和 T2 共享一个 L1，所以遵守上述规则足以确保 T1 中的两个 store 按程序顺序对 T2 可见，从而确保 load L2 读取 NEW。

另一方面，如果两个线程 T1 和 T2 属于不同的 CTA，因此映射到不同的 SM（SM1 和 SM2），则 load Ld2 可能会读取 0。要了解如何，请考虑以下事件序列。

初始化时，数据和标志都缓存在两个 L1 中。
St1 和 St2 按程序顺序执行，分别在 SM1 的 L1 缓存中写入 NEW 和 SET。
持有标志的缓存行从 SM1 的 L1 中被驱逐，并且 flag=SET 被写入 L2。
SM2 的 L1 中的持有标志的缓存行被驱逐。
Load Ld1 在 SM2 的 L1 中执行并 miss，因此从 L2 中取出行并读取 SET。
Load Ld2 执行，hit SM2 的 L1，并读取 0。
因此，尽管 FENCE 确保两个 store 以正确的顺序写入 SM1 的 L1，但在没有硬件 cache coherence 的情况下，它们可能以不同的顺序对 T2 可见。这就是为什么早期的 GPU 编程手册明确禁止在同一内核的线程之间进行这种类型的 CTA 间同步。

那么 CTA 之间的同步是不可能实现的吗？在实践中，一种解决方法是可能的，利用直接针对内存层次结构的特定级别的特殊 load 和 store 指令。正如我们在表 10.2 中看到的，T1 中的两个 store 绕过 L1 显式写入 GPU 范围（即 L2）。同样，来自 T2 的两个 load 绕过 L1 并直接从 L2 读取。因此，GPU 范围的两个线程 T1 和 T2 明确地通过使用绕过 L1 的 load 和 store 在 L2 同步。

Table 10.2: Message passing example. T1 and T2 belong to different CTAs.

| Thread T1 | Thread T2 | Comments |
|---|---|---|
| St1: St.GPU data = NEW;<br>FENCE;<br>St2: St.GPU flag = SET; | Ld1: Ld.GPU r1 = flag;<br>B1: if (r1 ≠ SET) goto Ld1;<br>FENCE;<br>Ld2: Ld.GPU r2 = data; | // Initially data and flag are 0.<br><br><br>Can r2==0? |

但是，上述解决方法存在问题——主要问题是由于绕过 L1 的 load 和 store 导致的性能低效。在这个简单的示例中，所讨论的两个变量必须在 SM 之间进行通信，因此绕过是必要的邪恶做法。但是，如果有更多变量，绕过可能会有问题，其中只有一些变量通过 SM 进行通信。在这种情况下，程序员面临着繁重的任务，即仔细地将 load/store 引导到适当的内存层次结构级别，以便有效地利用 L1。

**Flashback to Quiz Question 8:** GPUs do not support hardware cache coherence. Therefore, they are unable to enforce a memory consistency model. *True or false?*
**Answer:** *False!* Early GPUs did not support cache coherence in hardware, yet supported scoped relaxed consistency models.

总结：限制和要求
早期的 GPU 主要针对既不经常同步也不频繁共享数据的令人尴尬的并行工作负载。因此，此类 GPU 选择不支持用于保持 local cache coherent 的硬件 cache coherence，代价是只允许 CTA 内同步的 scoped memory consistency 模型。更灵活的 CTA 间同步要么效率太低，要么给程序员带来了巨大的负担。

程序员开始将 GPU 用于通用工作负载。此类工作负载往往涉及相对频繁的细粒度同步和更通用的共享模式。因此，GPGPU 最好具有：

允许跨所有线程同步的严格和直观的 memory consistency model；和
执行 consistency model 的 coherence 协议，同时允许有效的数据共享和同步，同时保持传统 GPU 架构的简单性，因为 GPU 仍将主要满足图形工作负载。

## 10.1.2 重点：GPGPU Consistency 和 Coherence

我们已经概述了 GPGPU consistency 和 coherence 所需的属性。满足这些需求的一种直接方法是使用类似多核 CPU 的方法来实现 consistency 和 coherence，即使用一种 consistency-agnostic 的 coherence 协议（我们在第 6-9 章中详细介绍过）来理想地实现强 consistency model，例如 sequential consistency (SC)。尽管勾选了几乎所有的方框——consistency model 肯定是直观的（没有范围的概念），并且 coherence 协议允许有效地共享数据——但这种方法并不适合 GPU。这有两个主要原因 [30]。

首先，在写入时使共享者无效的类似 CPU 的 coherence 协议会在 GPU 上下文中产生高流量开销。这是因为 GPU 中本地缓存 (L1) 的总容量通常与 L2 的大小相当，甚至更大。例如，NVIDIA Volta GPU 的总容量约为 10 MB 的 L1 缓存，而只有 6 MB 的 L2 缓存。一个独立的包含目录 (inclusive directory) 不仅会因为重复的标签而产生很大的面积开销，而且还会因为目录需要高度关联而导致显著的复杂性。另一方面，考虑到聚合 L1 的大小，嵌入式包容性目录会在 L2 驱逐时导致大量的召回流量。

其次，由于 GPU 维护着数千个活跃的硬件线程，因此需要跟踪相应数量的 coherence transaction，这将花费大量的硬件开销。

如果没有写入发起的失效，如何将 store 传播到其他非本地 L1 缓存？（即，如何使存储对属于其他 CTA 的线程可见？）如果没有写入发起的失效，如何强制执行 consistency model——更不用说强 consistency model 了？

在第 10.1.3 节和第 10.1.4 节中，我们讨论了两个采用自失效 (self-invalidation) [18] 的提议，即处理器使本地缓存中的行失效，以确保来自其他线程的 store 变得可见。

自失效协议可以有效地执行什么样的 consistency model？我们认为此类协议可以直接有效地执行 relaxed consistency model，而不是执行与 consistency-agnostic 的不变量，例如 SWMR 不变量。

Consistency model 是否应该包括范围 (scope) 正在争论中 [15, 16, 23]。尽管范围增加了程序员的复杂性，但它们可以说简化了 consistency 实现。因为不能排除范围，我们在 10.1.4 节中概述了一个 scoped consistency model，它不会将同步限制在范围子集内。值得注意的是，我们引入的 consistency model 在精神上与当今工业界 GPU 中使用的模型相似（它们都使用不将同步限制在范围子集内的范围模型）。

## 10.1.3 Temporal Coherence

在本节中，我们将讨论一种基于自失效 (self-invalidation) 的方法来执行称为时间一致性 (temporal coherence) 的一致性 [30]。关键思想是每个 reader 在称为租约 (lease) 的有限时间段内引入一个缓存块，在这个时间段结束时，该块将自失效。我们讨论了时间一致性的两种变体：(1) 一种强制 SWMR 的 consistency-agnostic 的变体，在这种变体中，writer stall 直到块的所有租约到期；(2) 一种更有效的 consistency-directed 变体，它直接强制执行 relaxed consistency model，其中 FENCE stall，而不是 writer stall。对于以下讨论，我们假设共享缓存（L2）是 inclusive 的：共享 L2 中不存在的块意味着它不存在于任何本地 L1 中。我们还假设 L1 使用 write-through/no write-allocate 策略：写入直接写入 L2（直写），写入 L1 中不存在的块不会在 L1 中分配块L1（非写分配）。

Consistency-agnostic Temporal Coherence
与其让非本地缓存中的所有共享者失效（这在 CPU 一致性中很常见），不如考虑一种协议，在该协议中，让写入者等待所有共享者都驱逐了该块。通过让写入等待直到没有共享者，该协议确保在写入成功的那一刻没有并发读取者 - 从而强制执行 SWMR。

写入者怎么知道要等多久？也就是说，写入者如何确定该块不再有共享者？时间一致性通过利用全局时间概念 (a global notion of time) 来实现这一点。具体来说，它要求 L1 和 L2 中的每一个都可以访问一个跟踪全局时间的寄存器。

在 L1 未命中时，读取器预测它希望将块在 L1 中保留多长时间，并通知 L2 这个持续时间称为租约 (lease)。每个 L1 缓存块都带有一个时间戳 (timestamp, TS) 标记，该时间戳 (TS) 包含该块的租约。当前时间大于其租用时间的 L1 块的读取被视为未命中。

此外，L2 中的每个块都带有时间戳。当读取器在 L1 缓存未命中时咨询 L2 时，它会通知 L2 其所需的租约；L2 根据"时间戳在所有 L1 中保持该块的最新租约"这一不变量来更新块的时间戳。

每次写入（即使该块存在于 L1 缓存中）都会直接写入 L2；写入请求访问 L2 中保存的块的时间戳，如果时间戳对应于未来的某个时间，则写入停止直到该时间。这种停顿确保在 L2 中执行写入时没有块的共享者，从而确保 SWMR。

例子。为了理解时间一致性是如何工作的，让我们考虑表 10.3 中的消息传递示例，暂时忽略 FENCE 指令。 让我们假设线程 T1 和 T2 来自两个不同的 CTA，映射到两个不同的 SM（SM1 和 SM2），具有单独的本地 L1。

Table 10.3: Message passing example. T1 and T2 belong to different CTAs.

| Thread T1 | Thread T2 | Comments |
|---|---|---|
| St1: St data1 = NEW; | Ld1: Ld r1 = flag; | // Initially all variables are 0. |
| St2: St data2 = NEW; | B1: if (r1 ≠ SET) goto Ld1; | |
| FENCE; | FENCE; | |
| St3: St flag = SET; | Ld2: Ld r2 = data2; | Can r2==0? |

我们在表 10.4 中说明了 SM1、SM2 和共享 L2 的事件时间线。最初，我们假设 flag、data1 和 data2 缓存在 SM2 的本地 L1 中，租约值分别为 35、30 和 20。在 time=1 时，执行 St1。由于 L1 使用 write-through/no write-allocate 策略，因此向 L2 发出写入请求。由于 data1 在 SM2 的 L1 缓存中有效、直到 time=30 才无效，所以写入会 stall、直到这个时间才接触。在 time=31 时，写入在 L2 执行。 St2 然后在时间=37 向 L2 发出写入请求，该请求在时间=42 到达 L2。此时 data2 (20) 的租约已经过期，因此写入在 L2 执行，没有任何停顿。类似地，St3 在 time=48 发出一个写请求，并且 flag 在 time=53 被写入 L2 而没有任何停顿，因为此时 flag 的租约已经到期。同时，在time=50时，Ld1检查自己的L1，发现flag的租约已经过期，于是向L2发出读请求，在time=60时完成。同样，Ld2也在time=61向L2发出读请求，从L2读取NEW的期望值，在time=71完成。因为这种时间一致性变体强制执行 SWMR，只要 GPU 线程按程序顺序发出内存操作，就会强制执行 SC（注5）。

Table 10.4: Temporal coherence: Timeline of events for example in Table 10.3

| /* T1 and T2 (belonging to different CTAs) are mapped to SM1 and SM2, respectively. flag = 0 (lease = 35), data1 = 0 (lease = 30) and data2 = 0 (lease = 20) are cached in L1 of SM2 */ | | | |
|---|---|---|---|
| **Time** | **SM1** | **SM2** | **L2** |
| 1 | St1 issues write request | | |
| 6 | | | write for data1 stalled until lease for data1 (30) |
| 31 | | | data1 = NEW written; Ack sent to SM1 |
| 36 | St1 completes | | |
| 37 | St2 issues write request | | |
| 42 | | | data2 = NEW written without stalling since current time>lease for data2 (20); Ack sent to SM1 |
| 47 | St2 completes | | |
| 48 | St3 issues write request | | |
| 50 | | L1 lease for flag (35) has expired, so Ld1 issues read request | |
| 53 | | | flag = SET written without stalling since current time>lease for flag (35); Ack sent to SM1 |
| 55 | | | flag = SET read; get new lease; sent to L1 of SM2 |
| 58 | St3 completes | | |
| 60 | | Ld1 completes | |
| 61 | | L1 lease for data2 (20) has expired, Ld2 issues read request | |
| 66 | | | data2 = NEW read; gets new lease; sent to L1 of SM2 |
| 71 | | Ld2 completes | |

原书作者注5：原始论文 [30] 认为 warp 等同于线程，而我们考虑更现代的 GPU 设置，其中线程可以单独调度。 因此，我们设置的 consistency 与他们的设置略有不同。

协议规范。我们分别在表 10.5 和表 10.6 中介绍了 L1 和 L2 控制器的详细协议规范。 L1 有两个稳定状态：(I) Invalid 和 (V) Valid。 L1 使用 GetV(t)、Write 和 WriteV(t) 请求与 L2 通信。 GetV(t) 携带时间戳作为参数，请求将指定块带入 L1 在所请求的租用期内处于有效状态，在该时间结束时，该块自失效。 写入请求要求将指定的值写入 L2 而不将块带入 L1。 WriteV(t) 请求用于写入在 L1 中已经有效的块，并携带保存其当前租约的时间戳作为参数。 出于演示的原因，我们仅提供协议的高级规范（以及本章中的所有其他协议）。 具体来说，我们只显示稳定状态和稳定状态之间的转换。

Table 10.5: Enforcing SWMR via Temporal Coherence: L1 Controller

| | Load | Store | Eviction /Expiry |
|---|---|---|---|
| I | send GetV(t) to L2<br>rec. Data from L2<br>/V | send Write to L2<br>rec. Write-Ack from L2 | |
| V | read hit | send WriteV(t) to L2<br>rec. Write-Ack from L2 | -/I |

Table 10.6: Enforcing SWMR via Temporal Coherence: L2 Controller

| | GetV(t) | Write | WriteV(t) | L2: Eviction | L2: Expiry |
|---|---|---|---|---|---|
| I | send Fetch to Mem<br>rec. Data from Mem<br>send Data to L1<br>TS←$t$<br>/P | send Fetch to Mem<br>rec. Data from Mem<br>write<br>send Write-Ack to L1<br>/Exp | | | |
| P | send Data to L1<br>TS←max(TS,$t$)<br>/S | stall (until expiry) | if($t$ = TS)<br>write<br>send Write-Ack to L1<br>else<br>stall (until expiry) | stall<br>(until expiry) | /Exp |
| S | send Data to L1<br>TS←max(TS,$t$) | stall (until expiry) | stall<br>(until expiry) | stall<br>(until expiry) | /Exp |
| Exp | send Data to L1<br>TS←$t$<br>/P | write<br>send Write-Ack to L1 | write<br>send Write-Ack to L1 | if(dirty)<br>send Write-back to Mem<br>rec. Ack from Mem<br>/I | |

状态 I 的 load 导致向 L2 发送 GetV(t) 请求；在从 L2 接收到数据后，状态转换为 V。状态 I 中的 store 导致向 L2 发送写入请求。 在收到来自 L2 的 ack（表明 L2 已写入数据）后，store 完成。 由于 L1 使用 no-write-allocate 策略，数据不会被带到 L1 并且块的状态保持在状态 I。

回想一下，如果全局时间超过该块的租约，则该块将变为无效——这在逻辑上由租约到期 （注6） 时的 V 到 I 转换表示。 在存储到状态 V 的块时，将 WriteV(t) 请求发送到 L2。 WriteV(t) 的目的是利用这样一个事实，即如果块被私有地保存在写入器的 L1 中，则写入不需要在 L2 处停止。 相反，写入可以简单地更新 L2 以及 L1，并继续在 L1 中缓存块，直到其租约用完。 WriteV(t) 请求携带时间戳的原因是为了确定该块是否私有地保存在 writer 的 L1 中，我们将在下面看到。

原书作者注6：值得注意的是，在实际实现中，硬件没有必要主动检测块已过期并将其转换为 I 状态 — 块已过期的事实可能会在该块的后续事务中延迟发现。
我们现在描述 L2 控制器。 L2 有四种稳定状态： (I) Invalid，表示该块既不存在于 L2 中，也不存在于任何 L1 中； (P) Private，表示该块恰好存在于其中一个 L1 中； (S) Shared，表示该块可能存在于一个或多个L1中； 和 (Exp) Expired，表示该块存在于 L2 中，但在任何 L1 中均无效。 在状态 I 接收到 GetV(t) 请求后，L2 从内存中获取块，根据请求的租约更新块的时间戳，然后转换到 P。在状态 I 接收到写入时，它从内存中获取块 ，更新 L2 中的值，发回一个 ack，然后转换到 Exp，因为它在任何 L1 中都无效。

在状态 P 中接收到 GetV(t) 请求后，L2 以数据响应，如果请求的租约大于当前时间戳，则延长时间戳，并转换到 S。对于在状态 P 中接收到的 WriteV(t) 请求，有两种情况。在简单的情况下，唯一持有该块的 SM 私下向它写入；在这种情况下，L2 可以简单地更新块而无需任何停顿并回复 ack。但是有一个棘手的极端情况，其中来自私有块的 WriteV(t) 消息被延迟，因此该块现在被私有地持有，但在不同的 SM 中！为了消除这两种情况的歧义，每个 WriteV(t) 消息都带有一个带有该块租约的时间戳，并且与 L2 中

保存的时间戳匹配表示前一种简单的情况。不匹配表示后者，在这种情况下，WriteV(t) 请求停止，直到块到期，此时 L2 被更新并且 ack 被发送回 L1。另一方面，在状态 S 中收到的 WriteV(t) 或写入请求将必须一直等到 L2 的块租约到期。最后，只允许在 Exp 状态下驱逐 L2 块，因为写入必须知道要停止多长时间才能强制执行 SWMR。

总之，我们看到了时间一致性如何使用租用读取和停止写入来强制执行 SWMR。 结合按程序顺序呈现内存操作的处理器，时间一致性可用于强制执行顺序连贯性 (SC)。

Consistency-directed Temporal Coherence
如前所述，时间一致性强制执行 SWMR，但成本很高，每次写入未过期的共享块都需要在 L2 控制器处停止。 由于 L2 由所有线程共享，因此在 L2 处停止会间接影响所有线程，从而降低整体 GPU 吞吐量。

回想一下，在第 2.3 节中，我们讨论了两类一致性接口：consistency-agnostic 和 consistency-directed。Consistency-agnostic coherence 接口通过在写入返回之前将写入同步传播到其他线程来强制执行 SWMR。 考虑到在 GPU 设置中执行 SWMR 的成本，我们可以探索连贯性导向的一致性吗？ 也就是说，我们可以在不违反一致性的情况下使它们异步可见，而不是使写入对其他线程同步可见吗？ 具体来说，诸如第 5 章的 XC（注7） 之类的宽松一致性模型仅要求程序员通过 FENCE 指令指示的内存顺序。 这样的模型允许写入异步传播到其他线程。

原书作者注：现在，让我们假设内存模型中没有作用域 (scope)。
考虑表 10.3 中显示的消息传递示例，XC 仅要求 St1 和 St2 在 St3 变得可见之前对 T2 可见。 它不要求在 St1 执行时它必须传播到 T2。 换句话说，XC 不需要 SWMR。 因此，XC 允许时间一致性的变体，其中只有 FENCEs 而不是写入需要停顿。 当写入请求到达 L2 并且块被共享时，L2 只需使用与块关联的时间戳回复启动写入的线程。 我们将此时间称为全局写入完成时间 (GWCT)，因为这表示线程在遇到 FENCE 时必须停止的时间，以确保写入对所有线程全局可见。

对于映射到 SM 的每个线程，SM 会跟踪为每个线程停止时间寄存器中的写入返回的最大 GWCT。 在遇到 FENCE 时，将线程暂停到此时可确保 FENCE 之前的所有写入都变得全局可见。

例子。对于同一个消息传递示例（表 10.3），我们在表 10.7 中说明了事件的时间线，但现在考虑到 FENCE 指令的影响，因为我们正在寻求实现一个宽松的 XC 类模型。

Table 10.7: Consistency-directed Temporal coherence: Timeline for Table 10.3

| /* T1 and T2 (belonging to different CTAs) are mapped to SM1 and SM2, respectively. flag = 0 (lease = 35), data1 = 0 (lease = 30), and data2 = 0 (lease = 20) are cached in L1 of SM2 */ | | | |
|---|---|---|---|
| **Time** | **SM1** | **SM2** | **L2** |
| 1 | St1 issues write request | | |
| 6 | | | data1=NEW written although current time < lease for data1 (30); Ack sent to SM1 with GWCT=30 |
| 11 | St1 completes, stall-time ←30 | | |
| 12 | St2 issues write request | | |
| 17 | | | data2=NEW written although current time < lease for data2 (20); Ack sent to SM1 with GWCT=20 |
| 22 | St2 completes; GWCT (20)< stall-time(30), so stall-time unchanged | | |
| 23 | FENCE stalls thread until stall-time (30) | | |
| 31 | St3 issues write request | | |
| 36 | | | flag=SET written since current time > lease for flag (35); Ack sent to SM1 |
| 40 | | L1 lease for flag (35) has expired, so Ld1 issues read request | |
| 41 | St3 completes | | |
| 45 | | | flag=SET read; gets new lease; sent to L1 of SM2 |
| 50 | | Ld1 completes | |
| 51 | | L1 lease for data2 (20) has expired so Ld1 issues read request | |
| 56 | | | data2=NEW read; gets new lease; sent to L1 of SM2 |
| 57 | | Ld2 completes | |

在 time=1 时，由于 St1 的写请求被发布到 L2。在 time=6 时，写入在 L2 执行，没有任何停顿，尽管其租约 (30) 那时还没有到期；L2 以 30 的 GWCT 进行回复，该 GWCT 在 SM1 的每线程停顿时间寄存器中被记住。类似地，由于 St2 的写请求在时间=12 时发出。在 time=17 时，在 L2 执行写入，L2 回复 GWCT 为 20。收到 GWCT 后，SM1 不会更新其停顿时间，因为当前值 (30) 更高。 FENCE 指令在时间=23 执行并阻塞线程 T1，直到其停止时间为 30。在时间=31 时，由于 St3 向 L2 发出写入请求，并在时间=36 时在 L2 执行写入。因为标志 (35) 的租约已经到期，所以 L2 不会以 GWCT 响应，并且 St3 在 time=41 完成。同时，在time=40 时，Ld1 尝试从 L1 读取标志，但它的租约将在time=35 时到期。因此，向L2发出对标志的读取请求，在time=45从L2读取SET，并在 time=50 完成。类似地，由于 Ld2 的读取请求在 time=51 发出，在 time=56 从 L2 读取并在 time=57 完成，返回 NEW 的预期值。

协议规范。Consistency-directed 的时间一致性协议规范与 consistency-agnostic 变体大多相似——我们在表 10.8（L1 控制器）和表 10.9（L2 控制器）中用粗体突出显示了差异。

Table 10.8: Consistency-directed Temporal Coherence: L1 Controller (diffs in bold)

| | Load | Store | Eviction/ Expiry |
|---|---|---|---|
| I | send GetV(t) to L2<br>rec. Data from L2<br>/V | send Write to L2<br>rec. Write-Ack+**GWCT** from L2<br>**stall-time←max(stall-time,GWCT)** | |
| V | read hit | send WriteV(t) to L2<br>rec. Write-Ack+**GWCT** from L2<br>**stall-time←max(stall-time,GWCT)** | -/I |

Table 10.9: Consistency-directed Temporal Coherence: L2 Controller (diffs in bold)

| | GetV(t) | Write | WriteV(t) | L2: Eviction | L2: Expiry |
|---|---|---|---|---|---|
| I | send Fetch to Mem<br>rec. Data from Mem<br>send Data to L1<br>TS←*t*<br>/P | send Fetch to Mem<br>rec. Data from Mem Write<br>send Write-Ack to L1<br>/Exp | | | |
| P | send Data to L1<br>TS←max(TS,*t*)<br>/S | ~~stall~~<br>**write**<br>**send Write-Ack+GWCT to L1** | if(*t* = TS)<br>write<br>send Write-Ack to L1<br>else<br>stall (until expiry) | stall<br>(until expiry) | /Exp |
| S | send Data to L1<br>TS←max(TS,*t*) | ~~stall~~<br>**write**<br>**send Write-Ack+GWCT to L1** | ~~stall~~<br>**write**<br>**send Write-Ack+GWCT to L1** | stall<br>(until expiry) | /Exp |
| Exp | send Data to L1<br>TS←*t*<br>/P | write<br>send Write-Ack to L1 | write<br>send Write-Ack to L1 | if(dirty)<br>send Write-back to Mem<br>rec. Ack from Mem<br>/I | |

与 L1 控制器（表 10.8）的主要区别在于来自 L2 的 Write-Acks 现在携带 GWCT。 因此，在接收到 Write-Ack 时，如果传入的 GWCT 大于该线程当前保持的停顿时间，则 L1 控制器会延长停顿时间。（回想一下，在遇到 FENCE 时，线程会停止，直到停止时间寄存器中保存的时间。）

与 L2 控制器（表 10.9）的主要区别在于写入请求不会导致停顿；相反，执行写操作并返回 GWCT 和 Write-Ack。 同样，状态 S 中的 WriteV(t) 请求也不会停止。

Temporal Coherence：总结和限制

我们看到了如何使用时间一致性来强制执行 SWMR 或直接强制执行宽松的一致性模型，例如 XC（注 8）。 后一种方法的主要好处是它消除了在 L2 的昂贵停顿；而是在碰到 FENCE 时在 SM 处写入停顿。更多优化是可能的，以进一步减少停滞[30]。 然而，时间一致性存在一些严重的限制。

支持 non-inclusive L2 缓存很麻烦。 这是因为时间一致性要求在一个或多个 L1 中有效的块必须在 L2 上具有可用的租用时间。 一个复杂的解决方法是可能的，其中一个未过期的块可能会从 L2 中被驱逐，前提是被驱逐的块的租约被保存在某个地方，例如与 L2 未命中状态保持寄存器 (MSHR) [30] 一起。

它需要全局时间戳。 由于现代 GPU 的面积相对较大，因此保持全局同步的时间戳可能很困难。 然而，最近的一项提议显示了如何在不使用全局时间戳的情况下实现时间一致性的变体 [25]。

性能可能对租约的选择很敏感。 租约太短会增加 L1 未命中率；过长的租约会导致写入（或 FENCE）停止更多。

时间一致性不能直接利用范围同步。 例如，CTA 范围同步中涉及的存储（直观地）不需要写入 L2，但在

时间一致性中，每个存储都写入 L2，因为它是在直写/无写入分配 L1 的假设下设计的 缓存。
时间一致性涉及时间戳。 时间戳在设计和验证过程中引入了复杂性（例如，时间戳翻转）。
原书作者注8：时间一致性 [30] 强制执行 XC 的变体，其中写入不是原子的。
总而言之，尽管上述大多数限制都可以采用变通方法，但它们确实会增加已经非常规的基于时间戳的一致性协议的复杂性。

10.1.4 释放连贯性导向的一致性
基于强大的租约 (lease) 理念的时间一致性 (temporal coherence) 具有足够的通用性，可以强制执行与连贯性无关 (consistency-agnostic) 和连贯性导向 (consistency-directed) 的一致性变体。另一方面，涉及租约和时间戳 (timestamp) 的协议可以说是繁琐的。在本节中，我们将讨论 GPGPU 一致性的另一种方法，称为释放连贯性导向的一致性 (release consistency-directed coherence, RCC)，它直接强制执行释放连贯性 (release consistency, RC)，它与 XC 的不同之处在于区分获取 (acquire) 和释放 (release)，而 XC 将所有同步视为相同的。

RCC 在灵活性上妥协，因为它只能强制执行 RC 的变体。但作为这种灵活性降低的回报，RCC 可以说更简单，可以自然地利用范围 (scope) 信息，并且可以使用 non-inclusive L2 cache。在下文中，我们首先简要回顾一下 RC 内存模型，然后用 scope 对其进行扩展。然后，我们描述了一个简单的 RCC 协议，然后进行了两个优化。每个协议都可以强制执行 RC 的 scoped 和 non-scoped 变体。

Release Consistency: Non-scoped and scoped variants
在本节中，我们从 non-scoped 变体开始讨论 RC 内存模型，然后用 scope 对其进行扩展。

回想一下，RC（在第 5.5.1 节中介绍）具有特殊的原子操作，可以在一个方向上对内存访问进行排序，而不是由 FENCE 强制执行的双向排序。具体来说，RC 有一个 release (Rel) store 和一个 acquire (Acq) load，它们强制执行以下顺序。

Acq Load -> Load/Store
Load/Store -> Rel Store
Rel Store/Acq Load -> Rel Store/Acq Load
考虑表 10.10 中显示的消息传递示例。将 St2 标记为 release 可确保 St1 -> St2 的排序；将 Ld1 标记为 acquire 可确保 Ld1 -> Ld2 的排序。Acquire（注9）(Ld1) 读取 release (St2) 写入的值。这样做时，release 与 acquire 同步，确保 St2 -> Ld1 在全局内存顺序中。结合所有上述排序意味着 St1 -> Ld2，从而确保 Ld2 看到新值而不是 0。

原书作者注9：acquire 是指 load 被标记为 acquire，release 是指 store 被标记为 release。
Table 10.10: Message passing example. Non-scoped RC.

| Thread T1 | Thread T2 | Comments |
|---|---|---|
| St1: St data1 = NEW;<br>St2: **Rel** St flag = SET; | Ld1: **Acq** Ld r1 = flag;<br>B1: if (r1 ≠ SET) goto Ld1;<br>Ld2: Ld r2 = data1 | // Initially all variables are 0.<br><br>Can r2==0? |

在没有 scope 的内存模型的变体中，只要 acquire 返回 release 写入的值，release 就会与 acquire 同步，无论它们所属的线程来自相同 scope 还是不同 scope。因此，在上面的示例中，无论 T1 和 T2 属于同一个 CTA 还是不同的 CTA，Ld2 都会看到新值。

A Scoped RC Model.

在 scoped RC 模型中，每个原子操作都与一个 scope 相关联。如果满足以下条件，则称 release 与 acquire 同步：（1）acquire load 返回 release store 写入的值；（2）每个原子操作的 scope 包括了执行其他操作的线程。

例如，仅当执行 acquire 和 release 的两个线程属于同一个 CTA 时，才说 CTA scope 的 release 与 CTA scope 的 acquire 同步。另一方面，GPU scope 的 release 被称为与 GPU scope 的 acquire 同步，无论这两个线程是来自同一个 CTA 还是不同的 CTA（只要它们是在同一个 GPU 上发射的）。

为了更直观，请考虑表 10.11 中显示的消息传递示例的 scoped 变体。正如我们所见，release St2 带有一个 GPU scope，而 acquire Ld1 只带有一个 CTA scope。如果 T1 和 T2 来自不同的 CTA，则 acquire (Ld1) 的 scope 不包括 T1。因此，在这种情况下，release 与 acquire 不同步，这意味着 r2 实际上可以读取旧值 0。另一方面，如果 T1 和 T2 来自同一个 CTA，则 r2 无法读取 0。

Table 10.11: Message passing example. Scoped RC.

| Thread T1 | Thread T2 | Comments |
|---|---|---|
| St1: St data1 = NEW;<br>St2: **GPU Rel** St flag = SET; | Ld1: **CTA Acq** Ld r1 = flag;<br>B1: if (r1 ≠ SET) goto Ld1;<br>Ld2: Ld r2 = data1 | // Initially all variables are 0.<br><br>Can r2==0? (could be 0, if T1 and T2 are from different CTAs) |

形式化 (formalizing) scoped RC 的一种方法是使用 Shasha 和 Snir 的形式化方法的一种变体 [28]，并使用偏序 (partial order) 而不是全局内存顺序 (global memory order) 来对冲突 (conflicting) 操作进行排序。更具体地说，并非所有冲突的操作都是有序的——只有相互同步的 release 和 acquire 是有序的。值得注意的是，NVIDIA 采用这种方法来形式化其 PTX 内存连贯性模型 [20]。

Release Consistency-directed Coherence (RCC)
在本节中，我们将介绍一种直接执行 RC 而不是执行 SWMR (single-writer-multiple-readers) 的协议。具体来说，该协议不会急切地将写入传播到其他线程——即，它不强制执行 SWMR。相反，一个线程的写操作在 release 时写入 L2，另一个线程的读操作在 acquire 时使自己的 L1 自失效 (self-invalidate) 并从 L2 中 pull 新值。这种交互方式使得一个线程的写操作对另一个线程的读操作可见 (visible)。

译者注：原文为 "Instead, writes are written to the L2 upon a release and become visible to another thread when that thread self-invalidates the L1 on an acquire and pulls in new values from the L2."
对于下面的讨论，让我们假设 write-back/write-allocate L1 cache。另外，现在让我们忽略 scope 并假设同步是在来自两个不同 CTA 的线程之间 (inter-CTA) 进行的。稍后我们将描述 RCC 如何有效地处理 CTA 内 (intra-CTA) 同步。RCC 涉及的主要步骤如下。

未标记为 acquire 或 release 的 load 和 store 的行为类似于在 write-back/write-allocate L1 cache 中的正常 load 和 store。
在 store 标记为 release 时，L1 中的所有脏块（由 release 写入的块除外）都写入 L2。然后，release 写入的block 写入 L2，保证 Load/Store -> Rel Store。
标记为 acquire 的 load 从 L2 读取块的新副本。然后，L1 中的所有有效块，除了由 acquire 读取的块之外，都将自失效，从而确保 Acq Load -> Load/Store。
译者注：一个疑问是，Rel Store 和 Acq Load 可以直接访问 L2 吗（不从 L1 中转）？PTX 里没有查到 Rel Store、Acq Load 直接访问 L2 的指令（不可以指定 cache operator）。原因是什么呢？
例子。考虑表 10.10 中的消息传递示例，假设 T1 和 T2 来自不同的 CTA，映射到两个不同的 SM（SM1 和 SM2）。最初让我们假设包含 data1 和 flag 的 cache block 同时缓存在 SM1 和 SM2 的 L1 中。我们在表 10.14 中说明了事件的时间线。在 time=t1 时，St1 被执行，导致 NEW 被写入缓存在 SM1 的 L1 中的 data1。在 time=t2 时，St2（标记为 release）导致 SM1 的 L1 中的所有脏块都写入 L2。因此，data1 在 time=t3 时被写入 L2。然后，在 time=t4 时，执行 release 写入，导致 SET 被写入缓存在 SM1 的 L1 中的 flag，随后在 time=t5 时写回 L2。在 time=t7 时，Ld1（标记为 acquire）发出读取 flag，导致在 time=t8 时从 L2 读取 flag=SET。在 time=t9 时，接收到值时，SM2 的 L1 中的所有有效 block，除了 acquire 读取的 block，都自失效。因此，data1 是自失效的。在 time=t10 时，Ld2 导致向 L2 发送

对 data1 的读取请求，并读取 NEW 的最新值。

Table 10.14: RCC: Timeline of events (for Table 10.10)

| /* T1 and T2 (belonging to different CTAs) are mapped to SM1 and SM2, respectively. flag = 0 and data1 = 0 are cached in L1s of both SM1 and SM2 */ | | | |
|---|---|---|---|
| **Time** | **SM1/L1** | **SM2/L1** | **L2** |
| $t_1$ | St1 writes data1=NEW and completes | | |
| $t_2$ | **Rel** St2 issues Write-back of data1 | | |
| $t_3$ | | | data1=NEW written and Ack sent back to L1 of SM1 |
| $t_4$ | Ack received; flag=SET written and its Write-back initiated | | |
| $t_5$ | | | flag=SET written and Ack sent back to L1 of SM1 |
| $t_6$ | **Rel** St2 completes | | |
| $t_7$ | | **Acq** Ld1 issues read for flag | |
| $t_8$ | | | flag=SET read and sent to L1 of SM2 |
| $t_9$ | | value received; data1 self-invali-dated; **Acq** Ld1 completes | |
| $t_{10}$ | | Ld2 issues read for data1 | |
| $t_{11}$ | | | data1=NE W read and sent to L1 of SM2 |
| $t_{12}$ | | Ld2 completes | |

利用范围 (scope)。如果所有原子操作都属于 GPU scope，则上述协议可按原样适用。这是因为协议在一次 release 操作时已经将所有脏数据 push 到 L2（与 GPU scope 对应的缓存级别），并在一次 acquire 操作时从 L2 pull 数据。值得注意的是，RCC 协议可以利用 CTA scope：CTA-scoped 的 release 不需要将脏块写回 L2；CTA-scoped 的 acquire 不需要使 L1 中的任何有效块自失效。

译者注：下表为译者添加。Scoped RC 的一个例子。

dThreadT1 ThreadT2　Comments

St1: St data1 = NEW;　Ld1: GPU Acq Ld r1 = flag;

St2: GPU Rel St flag = SET;　B1: if (r1 != SET) goto Ld1;

Ld2: Ld r2 = data1;

协议规范。我们在表 10.12 和 10.13 中展示了 L1 和 L2 controller 的详细规范。总共有两种稳定 (stable) 状态：(I)nvalid 和 (V)alid。我们假设 block 中的每个字节都带有一个脏标记位；如果任何字节被标记为脏，则整个 block 也被标记为脏。L1 使用 GetV 和 Write-back 请求与 L2 通信。GetV 要求将指定的 block 带入 L1，并处于 Valid 状态。与 temporal coherence 中使用的 GetV(t) 不同，GetV 不携带时间戳作为参数；通过 GetV 带入 L1 的 block 在 acquire 时会自失效。Write-back 请求将指定 block 内的脏字节从 L1 拷贝到 L2，且不会将 block 从 L1 中逐出。

Table 10.12: RCC: L1 controller

| | Load/<br>Acq.scope = CTA | Store/<br>Rel.scope = CTA | Acq.scope = GPU/<br>Acq (no scope) | Rel.scope = GPU/<br>Rel (no scope) | L1:<br>Eviction |
|---|---|---|---|---|---|
| I | send GetV to L2<br>rec. Data from L2<br>read/V | send GetV to L2<br>rec. Data from L2<br>write/V | send GetV to L2<br>rec. Data from L2<br>read/V<br>forall dirty blocks<br>send Write-back to L2<br>rec. Ack from L2<br>forall other valid blocks<br>Invalidate | forall dirty blocks<br>send Write-back to L2<br>rec. Ack<br>send GetV to L2<br>rec. Data from L2<br>write/V<br>send Write-back to L2<br>rec. Ack | |
| V | read hit | write hit | send GetV to L2<br>rec. Data from L2<br>read<br>forall dirty blocks<br>send Write-back to L2<br>rec. Ack from L2<br>forall other valid blocks<br>Invalidate | forall dirty blocks<br>send Write-back to L2<br>rec. Ack<br>write hit<br>send Write-back to L2<br>rec. Ack | if(dirty)<br>send Write-back to L2<br>rec. Ack from L2<br>/I |

Table 10.13: RCC: L2 controller

| | GetV | Write-back | L2:<br>Eviction |
|---|---|---|---|
| I | send Fetch to Mem<br>rec. Data from Mem<br>send Data to L1<br>/V | allocate block<br>write<br>send Ack to L1<br>/V | |
| V | send Data to L1 | write<br>send Ack to L1 | if(dirty)<br>send Write-back to Mem<br>rec. Ack from Mem<br>/I |

表 10.12 显示了 L1 controller。CTA scope 的 release 和 acquire 的行为类似于 write-back/write-allocate 中的普通的 store 和 load。如果 release 和 acquire 属于 GPU scope（或者如果 release 和 acquire 不携带 scope 信息），则协议必须在 release 时 write-back 所有脏块，并在 acquire 时使所有有效块自失效。在 acquire 时 block 自失效之前，脏块（如果有）需要被写入 L2 以确保它们的值不会丢失。（也可以延迟写回脏块，只要在下一个 release 之前写回即可。）

最后，值得注意的是该协议不依赖 cache inclusion。实际上，如表 10.13 所示，一个有效的 L2 block 可以在不通知 L1 的情况下被静默驱逐 (silently evivted)。直观地说，这是因为 L2 不包含任何关键的元数据 (metadata)，例如共享者 (sharer) 或所有权 (ownership) 信息。

总结。 综上所述，RCC 是一个直接执行 RC 的简单协议。因为它在 L2 上不保存任何协议元数据，所以它不需要 L2 是 inclusive 的，并允许 L2 block 被静默地逐出。该协议可以利用 scope 信息——特别是，如果 release 和 acquire 属于 CTA scope，则不需要昂贵的 write-back 或自失效。另一方面，在不了解 scope 的情况下，intra-CTA 同步效率低下：RCC 必须保守地假设 release 和 acquire 属于 GPU scope，因此即使同步发生在一个 CTA 内，也会从 L1 write-back/self-invalidate 数据 。

Exploiting Ownership: RCC-O

在上述的简单协议中，release（导致所有 dirty line 被写回 L2）和 acquire（导致所有 valid line 自失效）是昂贵的操作。降低 release 和 acquire 成本的一种方法是跟踪 (track) 所有权 (ownership) [13, 16]，因此 owned block 无需在 acquire 时自失效，也无需在 release 时写回。

关键思想是添加一个 O(wned) 状态——每个 store 都必须在后续 release 之前获得 L1 cache line 的 ownership。对于每个 block，L2 都维护着其所有者 (owner)，即拥有该 block 的 L1 的身份 (identity)。在请求 ownership 时，L2 首先降级 (downgrade) 之前的 owner（如果有的话），导致之前的 owner 将脏数据写回 L2。将当前数据发送给新 owner 后，L2 更改 ownership。因为处于状态 O 的块意味着没有 remote writer，所以不需要在 acquire 时使 Owned block 自失效。由于 L2 跟踪 owner，因此无需在 release 时将 Owned block 写回。

跟踪 ownership 还有另一个好处：即使在没有 scope 信息的情况下，跟踪 ownership 也有助于降低 intra-CTA 的 acquire 的成本。考虑表 10.10 中所示的消息传递 (message passing) 示例，假设现在线程 T1 和 T2 属于同一个 CTA。在没有 scope 信息的情况下，回想一下，RCC 将不得不保守地对待 release 和 acquire，在 release 时写回所有脏数据，并在 acquire 时使所有 valid block 自失效。使用 RCC-O，仍然必须保守地对待 release；release 前的所有 store 仍然必须获得 ownership。如果 acquire 的 block 处于 Owned 状态，则意味着相应的 release 必须来自同一 CTA 内的线程。因此，可以将 acquire 视为 CTA-scoped 的 acquire，并且可以避免 valid block 的自失效。

协议规范。我们在表 10.15 中提供了 L1 controller 的详细规范，在表 10.16 中提供了 L2 controller 的详细规范。主要变化是增加了 O 状态。每个 V（或 I）状态的 store 都必须联系 L2 controller 并请求 ownership。获得 ownership 后，该 line 的任何后续 store 都可以简单地更新 L1 中的 line，而无需联系 L2。因此，处于 O 状态的 block 在 L2 上可能是陈旧的。为此，当 L2 收到对已经处于 O 状态的 line 的 ownership 的请求时，L2 首先请求前一个 owner 写回该 block，将最新数据发送给当前请求者，然后更改 ownership。

Table 10.15: RCC-O: L1 controller. In non-scoped version, scope=GPU.

| | Load/<br>Acq.scope = CTA | Store/<br>Rel.scope = CTA | Acq.scope = GPU/<br>Acq (no scope) | Rel.scope = GPU/<br>Rel (no scope) | L1:<br>Eviction | From L2:<br>Req-Write-back |
|---|---|---|---|---|---|---|
| I | send GetV to L2<br>rec. Data from L2<br>read/V | send GetO to L2<br>rec. Data from L2<br>write/O | send GetV to L2<br>rec. Data from L2<br>read/V<br>forall other valid non-O blocks<br>Invalidate | send GetO to L2<br>rec. Data from L2<br>write/O | | |
| V | read hit | send GetO to L2<br>rec. Data from L2<br>write/O | send GetV to L2<br>rec. Data from L2<br>read<br>forall other valid non-O blocks<br>Invalidate | send GetO to L2<br>rec. Data from L2<br>write/O | /I | |
| O | read hit | write hit | read hit | write hit | send Write-back to L2<br>rec. Ack from L2/I | send Data to L2/V |

Table 10.16: RCC-O: L2 controller

| | GetV | GetO | Write-back | L2: Eviction |
|---|---|---|---|---|
| I | send Fetch to Mem<br>rec. Data from Mem<br>send Data to L1<br>/V | send Fetch to Mem<br>rec. Data from Mem<br>send Data to L1<br>set owner/O | allocate block<br>write<br>send Ack to L1/V | |
| V | send Data to L1 | send Data to L1<br>set owner/O | write<br>send Ack to L1 | /I |
| O | send Req-Write-back to L1 (owner)<br>rec. Data from L1 (owner)<br>write/V<br>send Data to L1 (requestor) | send Req-Write-back to L1 (owner)<br>rec. Data from L1 (owner)<br>send Data to L1 (requestor)<br>update owner/O | write<br>send Ack to L1/V | send Req-Write-back to L1 (owner)<br>rec. Data from L1 (owner)<br>send Write-back to Mem<br>rec. Ack from Mem<br>/I |

在碰到 release 时（CTA-scoped、GPU-scoped 或 non-scoped），无需将脏数据从 L1 写回 L2。相反，该协议仅要求所有以前的 store 都必须获得 ownership。在 acquire 时（GPU-scoped 或 non-scoped），只有 non-owned 的 valid block 需要自失效。如果 acquire 是 CTA-scoped 的，或者如果 acquire 是针对已经 owned 的 block，则意味着同步是 intra-CTA 的，因此不需要自失效。

最后，由于该协议依赖于维护 ownership 信息的 L2，因此不能静默驱逐 (silently evicted) 处于 O 状态的 L2 block；相反，它必须首先降级当前 owner，请求它写回所有有问题的 block。

总结。通过跟踪 ownership，与 RCC 相比，RCC-O 允许减少自失效的数量。在没有 scope 的情况下，跟踪 ownership 还可以检测 intra-CTA 的 acquire，并避免在这种情况下进行自失效。但是，intra-CTA 的 release 被保守地对待——每个到之前 unowned block 的 store 仍然需要在后续 release 之前获得 ownership。最后，RCC-O 不能允许处于状态 O 的 block 被静默地从 L2 中逐出；它必须首先降级当前 owner。

Lazy Release Consistency-directed Coherence: LRCC
在没有 scope 信息的情况下，RCC 会保守地对待 release 和 acquire。在 RCC-O 中，可以检测并有效处理 intra-CTA 的 acquire，但对 release 还是需要进行保守处理。

有没有办法有效地处理 intra-CTA 的 release 呢？延迟释放连贯性导向的一致性 (Lazy Release Consistency-directed Coherence, LRCC) 实现了这一点。在 LRCC 中，普通 store 不获得 ownership。只有 release store 可以获得 ownership，代表它程序顺序之前的 store。当一个 release 的 block 在稍后被 acquire 时，acuqire 时 block 的状态决定了一致性动作 (coherence action)。

如果 block 为 acquire 线程的 L1 所拥有 (owned)，则意味着同步为 intra-CTA 的；因此，不需要自失效。如果该 block 为 remote L1 所拥有，则意味着同步是 inter-CTA 的。在这种情况下，该 remote L1 中的脏 cache block 首先被写回，并且 acquire 线程的 L1 中的 valid block 自失效。因此，通过把一致性动作延迟到 acquire 时发生，LRCC 能够非常有效地处理 intra-CTA 同步。

示例：Inter-CTA 同步。考虑表 10.10 所示的消息传递示例，假设线程 T1 和 T2 属于不同的 CTA。最初，让我们假设包含 data1 和 flag 的 cache block 缓存在 SM1 和 SM2 的 L1 中。我们在表 10.17 中说明了事件的时间线。在 time=t1 时，执行 St1 导致 NEW 被写入缓存在 SM1 的 L1 中的 data1，且没有获得 ownership。在 time=t2 时，执行 St2（标记为 release），导致向 L2 发送对 flag 的 ownership 的请求。由于 flag 之前不为任何 SM 所拥有，因此 L2 通过简单地将 owner 设置为 SM1 并响应 SM1 来授予 (grant) ownership。收到响应后，SET 被写入 SM1 的 L1 的 flag，St2 在 time=t4 时完成。在 time=t5 时，执行 Ld1（标记为 acquire），导致向 L2 发送对 flag 的读取请求。在 L2，发现包含 flag 的 block 归 SM1 的 L1 所拥有；因此，L2 请求 SM1 的 L1 将 flag 写回。在 time=t7 时收到这个请求后，SM1 的 L1 首先写回所有脏的 non-owned block（包括 data1=NEW ）；然后，在 time=t9 时，它写回 flag=SET。在 time=t10 时，L2 接收到 flag，将其写入本地并将值转发 (forward) 给 SM2。在 time=t11 时，SM2 收到 flag=SET，Ld1 完成。最后，在 time=t12，Ld2 导致向 L2 发送对 data1 的读

取请求，并读取最新值 NEW。

Table 10.17: LRCC: Timeline of events (for Table 10.10), inter-CTA synchronization

| /* T1 and T2 (belonging to different CTAs) are mapped to SM1 and SM2, respectively. flag = 0 and data1 = 0 are cached in L1s of both SM1 and SM2 */ | | | |
|---|---|---|---|
| **Time** | **SM1/L1** | **SM2/L1** | **L2** |
| $t_1$ | St1 writes data1=NEW and completes | | |
| $t_2$ | **Rel** St2 issues ownership request for flag | | |
| $t_3$ | | | ownership for flag obtained; and sent to SM1 |
| $t_4$ | ownership received; flag=SET written and St2 completes | | |
| $t_5$ | | **Acq** Ld1 issues read for flag | |
| $t_6$ | | | write-back request for flag issued to L1 of SM1 (since it owns flag) |
| $t_7$ | (before responding with flag) write-back of data1 initiated | | |
| $t_8$ | | | data1=NEW written and Ack sent to L1 of SM1 |
| $t_9$ | Ack received; flag=SET sent to L2 | | |
| $t_{10}$ | | | flag value written and sent to L1 of SM2 |
| $t_{11}$ | | **Acq** Ld1 completes | |
| $t_{12}$ | | Ld2 issues read for data1 | |
| $t_{13}$ | | | data1=NEW read; sent to L1 of SM2 |
| $t_{14}$ | | Ld2 completes | |

示例：Intra-CTA 同步。让我们考虑表 10.10 所示的消息传递示例，现在假设线程 T1 和 T2 属于同一个 CTA，因此映射到同一个 SM，即 SM1。我们在表 10.18 中说明了事件的时间线。在 time=t1 时，执行 St1，导致 NEW 被写入缓存在 L1 中的 data1。在 time=t2 时，执行 St2（标记为 release），导致向 L2 发送对 flag 的 ownership 的请求。获得 ownership 后，将 SET 写入缓存在 L1 中的 flag。在 time=t5，执行Ld1（标记为 acquire）。由于 flag 归 SM1 所有，因此没有自失效，并且 flag=SET 只是从 L1 中读

取。在 time=t6 时，执行 Ld2，从 L1 读取 data1=NEW。

Table 10.18: LRCC: Timeline of events (for Table 10.10), intra-CTA synchronization

| /* Both T1 and T2, belonging to the same CTAs, are mapped to SM1. flag = 0 and data1 = 0 are cached in SM1's L1. */ | | | |
|---|---|---|---|
| **Time** | **T1 (SM1/L1)** | **T2 (SM1/L1)** | **L2** |
| $t_1$ | St1 writes data1=NEW and completes | | |
| $t_2$ | **Rel** St2 issues ownership request | | |
| $t_3$ | | | ownership for flag obtained and sent to SM1 |
| $t_4$ | ownership received; flag=SET written and St2 completes | | |
| $t_5$ | | **Acq** Ld1 reads flag=SET from L1 and completes | |
| $t_6$ | | Ld2 reads data1=NEW from L1 and completes | |

协议规范。我们在表 10.19 中给出了 L1 controller 的详细规范。L2 controller 与 RCC-O 的 controller 相同（表 10.16）。与 RCC-O 相比，并非所有 store 都获得 ownership；只有 non-scoped/GPU-scoped 的 release 才能获得 ownership。与 RCC-O 一样，CTA-scoped 的 acquire、或者对由 L1 所拥有的 block 的 acquire、意味着 intra-CTA 同步；因此，没有自失效。相反，对于不属于 L1 的 block 的 GPU-scoped/non-scoped 的 acquire 意味着 inter-CTA 同步并涉及自失效。具体地，向 L2 发送 GetV 请求，以获取 acquire 的 block 的最新值。（如果在 L2，请求的 block 由另一个 L1 所拥有，L2 会导致 L1 在写回请求的 block 之前、写回其所有脏的、非拥有的 block。）然后所有 valid non-owned L1 block，除了刚刚被 acquire 读取的 block 之外，都是自失效的。在块自失效之前，non-owned dirty block（如果有）被写回 L2 以确保它们的值不会丢失。（或者，可以延迟写回脏块、直到下一个 release。）最后，就像在 RCC-O 中一样，LRCC 取决于 L2 维护 ownership 信息。因此，状态 O 的 L2 块不能被静默驱逐；相反，它必须降级当前 owner，要求它不仅写回所有有问题的 block，而且还写回该 L1 中的任何其他非拥有的脏块。

Table 10.19: LRCC: L1 controller. In non-scoped version scope is set to GPU.

| | **Load/ Acq.scope = CTA** | **Store/ Rel.scope = CTA** | **Acq.scope = GPU/ Acq (no scope)** | **Rel.scope = GPU/ Rel (no scope)** | **L1: Eviction** | **From L2: Req-Write-back** |
|---|---|---|---|---|---|---|
| I | send GetV to L2<br>rec. Data from L2<br>read/V | send GetV to L2<br>rec. Data from L2<br>write/V | send GetV to L2<br>rec. Data from L2<br>read/V<br>forall dirty non-O blocks<br>send Write-back to L2<br>rec. Ack from L2<br>forall other valid non-O blocks<br>Invalidate | send GetO to L2<br>rec. Data from L2<br>write/O | | |
| V | read hit | write hit | send GetV to L2<br>rec. Data from L2<br>read<br>forall dirty non-O blocks<br>send Write-back to L2<br>rec. Ack from L2<br>forall other valid non-O blocks<br>Invalidate | send GetO to L2<br>rec. Data from L2<br>write/O | if(dirty)<br>send Write-back to L2<br>rec. Ack from L2<br>/I | |
| O | read hit | write hit | read hit | write hit | for all other dirty non-O blocks<br>send Write-back to L2<br>rec. Ack from L2/V<br>send Write-back to L2<br>rec. Ack from L2/I | for all other dirty non-O blocks<br>send Write-back to L2<br>rec. Ack from L2/V<br>send Data to L2/V |

总结。 通过延迟一致性动作直到获取，LRCC 允许在没有范围信息的情况下有效地检测和处理 CTA 内同步。 LRCC 不能允许状态 O 中的块被悄悄地从 L2 中逐出，但这样做的影响仅限于同步对象，因为只有释放存储才能获得所有权。

Release Consistency-directed Coherence: Summary
在本节中，我们讨论了释放连贯性导向的一致性的三种变体，它们可用于强制执行范围和非范围的释放连贯性。 虽然所有变体都可以利用范围知识，但 LRCC 可以有效地处理 CTA 内同步，即使在没有范围的情况下也是如此。

鉴于 LRCC 即使在没有作用域的情况下也可以有效地处理同步，我们能否摆脱作用域内存连贯性模型？ 在我们看来，不完全是。 虽然惰性有助于避免在 CTA 内同步的情况下进行浪费的回写，但它会使 CTA 间的获取速度变慢。 这是因为惰性迫使来自释放线程的 L1 的所有回写都在获取的关键路径上执行。 在两个设备之间（例如 CPU 和 GPU 之间）同步的情况下，这种效果变得更加明显。

因此，GPU 和异构内存模型是否必须涉及范围是一个复杂的可编程性与性能权衡。

# 10.2 比 GPU 更多的异构性

TODO

# 10.3 进一步阅读

TODO

# 10.4 参考文献

[1] The AMBA CHI Specification. AMBA 246

[2] The CCIX Consortium. CCIX 246

[3] The GenZ Consortium. Gen-Z Consortium: Computer Industry Alliance Revolutionizing Data Access 246

[4] The OpenCAPI Consortium. OpenCAPI Consortium: Official Site 246

[5] T. M. Aamodt, W. W. L. Fung, and T. G. Rogers. General-Purpose Graphics Processor Architectures. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2018. DOI: 10.2200/s00848ed1v01y201804cac044 212

[6] Y. Afek, G. M. Brown, and M. Merritt. Lazy caching. ACM Transactions on Programming Languages and Systems, 15(1):182–205, 1993. DOI: 10.1145/151646.151651 247

[7] N. Agarwal, D. W. Nellans, E. Ebrahimi, T. F. Wenisch, J. Danskin, and S. W. Keckler. Selective GPU caches to eliminate CPU-GPU HW cache coherence. In IEEE International Symposium on High Performance Computer Architecture (HPCA), Barcelona, Spain, March 12–16, 2016. DOI: 10.1109/hpca.2016.7446089 246, 247

[8] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson. GPU concurrency: Weak behaviours and programming assumptions. In Proc. of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 577–591, Istanbul, Turkey, March 14–18, 2015. DOI: 10.1145/2775054.2694391 214

[9] J. Alsop, M. S. Orr, B. M. Beckmann, and D. A. Wood. Lazy release consistency for GPUs. In 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 26:1–26:13, Taipei, Taiwan, October 15–19, 2016. DOI: 10.1109/micro.2016.7783729 247

[10] J. Alsop, M. D. Sinclair, and S. V. Adve. Spandex: A flexible interface for efficient heterogeneous coherence. In ISCA, 2018. DOI: 10.1109/isca.2018.00031 245, 246

[11] L. Alvarez, L. Vilanova, M. Moretó, M. Casas, M. González, X. Martorell, N. Navarro, E. Ayguadé, and M. Valero. Coherence protocol for transparent management of scratchpad memories in shared memory manycore architectures. In Proc. of the 42nd Annual International Symposium on Computer Architecture, pages 720–732, Portland, OR, June 13–17, 2015. DOI: 10.1145/2749469.2750411 247

[12] J. F. Cantin, J. E. Smith, M. H. Lipasti, A. Moshovos, and B. Falsafi. Coarse-grain coherence tracking: RegionScout and region coherence arrays. IEEE Micro, 26(1):70–79, 2006. DOI: 10.1109/mm.2006.8 246

[13] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C. Chou. DeNovo: Rethinking the memory hierarchy for disciplined parallelism. In International Conference on Parallel Architectures and Compilation Techniques (PACT), pages 155–166, Galveston, TX, October 10–14, 2011. DOI: 10.1109/pact.2011.21 231, 247

[14] M. Elver and V. Nagarajan. TSO-CC: Consistency directed cache coherence for TSO. In 20th IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 165–176, Orlando, FL, February 15–19, 2014. DOI: 10.1109/hpca.2014.6835927 247

[15] D. R. Hower, B. A. Hechtman, B. M. Beckmann, B. R. Gaster, M. D. Hill, S. K. Reinhardt, and D. A. Wood. Heterogeneous-race-free memory models. In Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 427–440, Salt Lake City, UT, March 1–5, 2014. DOI: 10.1145/2541940.2541981 213, 217, 239, 247

[16] R. Komuravelli, M. D. Sinclair, J. Alsop, M. Huzaifa, M. Kotsifakou, P. Srivastava, S. V. Adve, and V. S. Adve. Stash: Have your scratchpad and cache it too. In Proc. of the 42nd Annual International Symposium on Computer Architecture, pages 707–719, Portland, OR, June 13–17, 2015. DOI: 10.1145/2872887.2750374 217, 231, 247

[17] L. I. Kontothanassis, M. L. Scott, and R. Bianchini. Lazy release consistency for hardware-coherent multiprocessors. In Proc. Supercomputing, page 61, San Diego, CA, December 4–8, 1995. DOI: 10.21236/ada290062 247

[18] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In Proc. of the 22nd Annual International Symposium on Computer Architecture (ISCA), pages 48–59, Santa Margherita Ligure, Italy, June 22–24, 1995. DOI: 10.1109/isca.1995.524548 217, 247

[19] M. Lis, K. S. Shim, M. H. Cho, and S. Devadas. Memory coherence in the age of multicores. In IEEE 29th International Conference on Computer Design (ICCD), Amherst, MA, October 9–12, 2011. DOI: 10.1109/iccd.2011.6081367 247

[20] D. Lustig, S. Sahasrabuddhe, and O. Giroux. A formal analysis of the NVIDIA PTX memory consistency model. In Proc. of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2019. DOI: 10.1145/3297858.3304043 228

[21] D.Lustig, C.Trippel, M.Pellauer, and M.Martonosi. ArMOR: Defending against memory consistency model mismatches in heterogeneous architectures. In Proc. of the 42nd Annual International Symposium on Computer Architecture, pages 388–400, Portland, OR, June 13–17, 2015. 240

DOI: 10.1145/2749469.2750378

[22] L. E. Olson, M. D. Hill, and D. A. Wood. Crossing guard: Mediating host-accelerator coherence interactions. In Proc. of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 163–176, Xi'an, China, April 8–12, 2017. DOI: 10.1145/3093336.3037715 246

[23] M. S. Orr, S. Che, A. Yilmazer, B. M. Beckmann, M. D. Hill, and D. A. Wood. Syn- chronization Using Remote-Scope Promotion. In Proc. of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 73–86, Istanbul, Turkey, March 14–18, 2015. DOI: 10.1145/2694344.2694350 217

[24] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood. Heterogeneous system coherence for integrated CPU-GPU systems. In The 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46), pages 457–467, Davis, CA, December 7–11, 2013. DOI: 10.1145/2540708.2540747 246

[25] X. Ren and M. Lis. Efficient sequential consistency in GPUs via relativistic cache coherence. In IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 625–636, Austin, TX, February 4–8, 2017. DOI: 10.1109/hpca.2017.40 226, 247

[26] A. Ros and S. Kaxiras. Complexity-effective multicore coherence. In International Conference on Parallel Architectures and Compilation Techniques (PACT), pages 241–252, Min- neapolis, MN, September 19–23, 2012. DOI: 10.1145/2370816.2370853 247

[27] Y. S. Shao, S. L. Xi, V. Srinivasan, G. Wei, and D. M. Brooks. Co-designing accelerators and SOC interfaces using gem5-aladdin. In 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 48:1–48:12, Taipei, Taiwan, October 15–19, 2016. DOI: 10.1109/micro.2016.7783751 247

[28] D. E. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. ACM Transactions on Programming Languages and Systems, 10(2):282–312, 1988. DOI: 10.1145/42190.42277 228

[29] M. D. Sinclair, J. Alsop, and S. V. Adve. Chasing away RAts: Semantics and evaluation for relaxed atomics on heterogeneous systems. In Proc. of the 44th Annual International Symposium on Computer Architecture (ISCA), pages 161–174, New York, NY, ACM, 2017. DOI: 10.1145/3079856.3080206 247

[30] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt. Cache coherence for GPU architectures. In 19th IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 578–590, Shenzhen, China, February 23–27, 2013. DOI: 10.1109/mm.2014.4 216, 217, 220, 223, 226, 247

[31] X. Yu and S. Devadas. Tardis: Time traveling coherence algorithm for distributed shared memory. In International Conference on Parallel Architecture and Compilation (PACT), pages 227–240, San Francisco, CA, October 18–21, 2015. DOI: 10.1109/pact.2015.12 247