GPU 持久内存中持久事务的自适应数据路径选择

1st Xinjian Long，网络与交换技术国家重点实验室，
北京邮电大学中国北京 barbiel origin@bupt.edu.cn

–

arXiv:2301.04392v1 [cs.AR] 11 Jan 2023

摘要—**新型非易失性内存技术依靠数据可恢复性，实现计算机应用中字节可寻址持久性的承诺。持久事务（例如日志记录）是提供可恢复数据结构的主要持久性编程模型之一。为实现对持久性内存进行高性能的故障原子事务更新，近期研究开始探索将持久化数据路径分离的多数据路径架构，并将其用于 CPU。鉴于 GPU 是许多应用领域的关键计算平台，我们研究了在 GPU 中用于持久性内存持久事务的多数据路径架构。我们提出的 AGPM 采用自适应数据路径选择策略来处理持久性内存中的日志更新。与最先进的设计相比，AGPM 可将 GPU 内核的执行时间至少缩短 24.37%（最多缩短 66.44%）。**

索引词—**图形处理器、持久内存、持久事务**

I. 引言

持久性内存（PM）或非易失性内存（NVM）技术受到了学术界 [1, 2, 3, 4, 5, 6, 7]和工业界 [8, 9, 10, 11]的广泛关注。PM 有望提供与磁盘存储相当的高集成密度以及与动态随机存取存储器（DRAM）相当的低延迟 [12]，同时支持字节粒度可寻址性和持久性[13]。近期实现商业化的 PM 产品是英特尔的 Optane NVM [14]。多年来，针对 CPU 的 PM 研究已经积累了大量成果 [6, 15, 16, 17, 18, 19, 20]，，但对 GPU 的相应探索却出人意料地十分有限，更不用说在现代 GPU 上实现板载 PM 了。然而，PM 的特性（更大的内存容量、更高的访问速度、数据可恢复性等）已被证明有利于 GPU 工作负载 [21]。因此，GPU 有必要支持 PM，并且面向 GPU 的 PM 设计具有巨大的潜力，能够实现高性能。

数据可恢复性是 GPU 应用程序受益于持久性内存的关键原因之一。例如，长时间运行的 GPU 应用程序（如深度神经网络训练、区块链应用中的工作量证明算法等）可以依靠驻留在主存中的可恢复数据结构来提升性能，而无需进行代价高昂的系统检查点操作 [22, 23]。实现可恢复数据结构需要持久性模型 [6, 7, 16, 24, 25] 。这是因为，由于存在易失性写回缓存，持久性内存更新的提交顺序可能与其到达持久性内存的顺序不同。此外，持久性模型可以提供类似事务的语义 [6, 8, 26],，使程序中由持久事务划定的部分能够以全有或全无的方式恢复，其中所有数据要么全部持久化，要么全部不持久化。持久事务可以根据持久性和一致性需求，采用不同的数据路径将数据写入持久性内存。一方面，可以使用$store$ ，随后执行 $clwb$ 指令，将特定缓存行从易失性缓存层级持久化到持久性内存（时序路径）。另一方面，应用程序也可以使用非时序指令绕过缓存，直接向持久性内存写入数据（非时序路径）。通过分析不同的 GPU 工作负载，我们发现，数据路径选择不当会进一步加剧性能问题。目前大多数持久性模型都设计为静态采用单数据路径架构，这可能导致使用持久性内存的应用程序遭受不必要的性能损失。

近期，Shahri 等人 [27] 提出扩展 x86 内存持久性模型，以利用沿时序路径和非时序路径发送的请求在速度上的差异，减少对使用 $sfence$ 指令的依赖。Jeong 等人 [28] 提出了一种软硬件协同设计方案，利用相互独立的 FIFO 数据路径来强制执行持久化顺序。这些设计仅将 PM 请求的类型（$load$、 $store$ 等）视为选择数据路径的关键因素。例如，在使用撤销日志记录时，日志更新必须先于相应的数据更新持久化。在这种情况下，直观而言，应通过非时序路径而非时序路径来更新日志 [13, 29]。这是因为非时序路径通常比时序路径更快，因为它跳过了处理器与持久化域之间的多级缓存。此外，日志预计在无故障执行期间不会被读取，而利用时序数据路径的写不分配策略，可以避免污染易失性缓存，同时降低撤销日志记录的开销。与这一直觉相反，我们观察到，由于 GPU 应用程序的内存访问行为各不相同，使用不同数据路径的性能也会有所差异。在某些情况下，我们发现通过较慢的时序路径更新日志，比使用较快的非时序路径更有利。另一方面，对于具有时间和

空间局部性的数据更新，时序路径也并非总是更有利。具体而言，当数据局部性过强，且重用距离过长，以至于缓存数据在被驱逐前无法得到重用时，相较于时序路径，更推荐使用非时序路径。总的来说，我们的观察表明，采用静态路径选择策略来处理 GPU 持久内存请求并不合适；多路径架构在提升 GPU 工作负载持久化顺序方面仍有很大潜力。

基于上述观察，我们提出一种自适应方法（AGPM），为 GPU 中的持久事务选择数据路径。该方法在 GPU 内存层次结构中增加 AGPM 缓冲区，用于记录持久内存日志更新与后续数据更新之间的数据局部性，以及 GPU L1 D 缓存和共享 L2 缓存中日志更新之间的局部性。系统会定期将 AGPM 缓冲区中的统计信息传递给每个流式多处理器中新增加的路径选择器，并将其转换为 8 种原因/模式。转换依据包括持久内存日志数据在 L1D 缓存和 L2 缓存中的时间局部性与空间局部性、GPU 内核对共享内存的使用情况，以及日志更新的数量。这些原因/模式用于推导撤销日志记录的数据路径选择决策。总的来说，本文的贡献如下：

1) 我们分析了 GPU 内核与持久性内存进行事务性交互（撤销日志记录）时的 L1D 和 L2 数据局部性，以及其在不同类型 GPU 工作负载中的性能表现。2) 我们提出一种用于 GPU 中持久内存请求的数据路径选择自适应方法，称为 AGPM；该方法根据内核访问的数据局部性和特征，选择持久内存日志更新的数据路径。3) 实验结果表明，与静态数据路径选择方法相比，AGPM 显著提升了性能。同时，AGPM 的性能优于面向持久性内存的最先进多数据路径架构。与最先进的设计相比，AGPM 至少可将 GPU 内核的执行时间缩短 24.37%（最多缩短 66.44%）。

本文其余部分安排如下：第二节介绍背景。第三节讨论本研究的研究动机。第四节介绍所提出的 AGPM 设计。第五节将所提出的 AGPM 与其他方法的结果进行比较。第六节讨论相关工作，最后在第七节给出总结。

II. 背景

本节回顾 CPU 和 GPU 上现有的持久性模型。我们还将依据 NVIDIA/CUDA 术语介绍 GPU 架构和编程模型。值得注意的是，本节提到的技术以及后续章节所述的设计，都可以应用于 NVIDIA 以外的其他 GPU 架构。

A. 内存持久性模型

字节寻址持久内存为借助可恢复数据结构（RDS）实现高性能内存计算带来了光明前景。然而，由于存在易失性写回缓存，$load$ 和 $store$ 请求到达持久内存的顺序可能不同于其提交顺序。这违反了 RDS 的规则。例如，假设 p 是一个持久化结构。'$p \rightarrow$ 数据'和'$p\rightarrow$ 状态'位于不同的缓存行中，并且程序中对'$p\rightarrow$ 数据'的更新先于对'$p \rightarrow$ 状态'的更新。然而，由于缓存的存在，内存中的'$p \rightarrow$ 状态'可能会先于'$p\rightarrow$ 数据'更新。在这种情况下，如果发生故障（例如电源故障），恢复供电后持久内存状态就会变得不正确。为解决这一问题并支持 RDS 的正确实现，研究者提出了内存持久性模型[6, 7, 16, 24, 25, 30] ，以正式规定写入持久内存的顺序。这些模型大致可分为严格模型和宽松模型，两者在并发程度上有所不同。在严格持久性模型中，易失性内存操作的顺序与持久内存操作的顺序完全一致。该模型易于实现，但会限制程序的并发性，从而导致严重的性能下降。宽松持久性模型打破了易失性内存操作与持久内存操作之间的绑定，因此性能更高。它支持更高程度的持久内存写入并发性，但代价是需要程序注释并增加硬件复杂性。

上述内存持久性模型规定了存储操作的持久化顺序，但仅有顺序约束并不能保证数据可恢复性。例如，假设在严格或宽松持久性模型下，持久性内存（PM）中的 '$p \rightarrow$ data' 和 '$p \rightarrow$ state' 按程序顺序更新。然而，仍有可能在 '$p \rightarrow$data' 更新后、'$p \rightarrow$ state' 更新前发生故障。在这种情况下，持久性内存中的内存状态仍不适合数据恢复。为解决这一问题，需要使用持久事务 [6, 8, 26]（例如撤销日志记录），通过撤销中止的事务所做的更改来提供全有或全无保证。为能够回滚更改，事务中的每次更新执行前都会创建一个撤销日志条目。该撤销日志条目包含即将更新的持久化结构的当前值。只有在日志条目创建并持久化后，才会更新实际值。如果事务成功，则会以原子方式发送提交消息，使该事务所属的日志条目失效。如果事务失败，恢复过程会使用所有已持久化的日志条目，回滚该事务造成的部分更改。

目前，内存持久性模型以及持久事务都是为 CPU 设计的。由于工作负载和处理器架构都存在差异，这些模型需要针对 GPU 重新设计。Lin 等人 [31]探索了在 GPU 中实现不同级别（内核级、CTA 级和循环级）内存持久性模型的方法。本研究采用了他们的实现，尤其是针对 GPU 的 CTA 级撤销日志记录。图 1 展示了

一个 CTA 级撤销日志记录的示例。首先，为 CTA 将要更新的输出元素（**Loc1** 和 **Loc2**）创建撤销日志。在确保所有线程使用 $sfence$，随后调用 CUDA $syncthreads$ 函数将其日志持久化之后，将日志标志设为 **inTx** 并使其持久化。CTA 迭代更新输出元素的值（**p→数据** 和 **p→state**）。在 CTA 结束时，使用另一个 $syncthread$ 函数确保所有输出均已持久化，并将标志设为 **complete**。请注意，本研究以下各节中所述的所有撤销日志记录均指 CTA 级撤销日志记录。

```
__device__ void CTA_log_example(…)
{
        Loc1[tid] = p->data[tid];
        Loc2[tid] = p->state[tid];              //p is initialized

        log(Loc1[tid]);clwb;sfence;
        log(Loc2[tid]);clwb;sfence;             //logs for p

        __synchthreads();                       //logs are persisted

        if(tid==0){
                flag=inTx;
                log(flag);clwb;sfence;          //flag is persisted
        }

        for(iterations){
                …                               //p is updated
                Loc1[tid] = p->data[tid];
                clwb;sfence;
                Loc2[tid] = p->state[tid];
                clwb;sfence;
        }

        __synchthreads();                       //CTA completes

        if(tid==0){
                flag=complete;
                log(flag);clwb;sfence;          //flag is persisted
        }
}
```

图 1. GPU 内核的 CTA 级撤销日志记录示例。

B. 配备持久性内存的 *GPU*

目前，GPU 上尚无搭载持久性内存（PM）的硬件。如果 GPU 应用希望利用 PM 的持久化特性，通常会在 GPU 上执行计算，并确保结果在 CPU 上持久化。计算结果需要从 GPU 内存传输到 CPU 内存，并依靠 CPU 保证数据可恢复性。或者，也可以利用文件系统，因为在非易失性内存技术出现之前，只有构建在块存储设备之上的文件系统才能保证持久化。GPU 的计算结果传输到 CPU 内存后，CPU 会将结果写入一个驻留于持久性内存中的文件，然后使用 $fsync$ 函数确保持久化。此外，驻留于持久性内存中的文件可以映射到 CPU 地址空间，而 GPU 的结果可以使用 $cudaMemcpy$ 传输到驻留于 CPU 内存中的内存映射文件。

Pandey 等人提出了 GPM [21]，利用英伟达的统一虚拟寻址（UVA）技术 [32] 将持久性内存映射到 GPU 的地址空间，并通过系统作用域栅栏以及有选择地禁用数据直接 I/O（DDIO）功能，使 GPU 无需 CPU 参与即可访问持久性内存并将数据持久化。GPM 利用了这样一个事实：现代持久性内存（英特尔傲腾 [14]）与动态随机存取存储器（DRAM）并列放置（如典型的英特尔至强服务器中），并且 GPU 可通过 PCIe 互连访问它。在这种情况下，可使用 UVA 将所需的非易失性存储器区域映射到 GPU 内核的虚拟地址空间。随后，GPU 内核无需 CPU 协助，即可使用 $load$和 $store$以字节粒度直接访问和操作驻留在持久性内存中的数据结构。

III. 研究动机

本节讨论本研究的动因。为简单起见，我们假设采用配备持久性内存的独立 GPU 系统（图 2）。这一假设使我们能够专注于 GPU 内存层次结构，而不考虑主机端存储系统可能产生的影响，以及系统与 GPU 设备内存之间代价高昂的数据传输。我们进一步假设持久内存控制器支持异步 DRAM 刷新（ADR） [33] 功能，且处于持久化域中。所有写入持久性内存的更新一旦到达持久内存控制器，即可持久化。英特尔近期宣布推出增强型 ADR（eADR） [34] 功能。发生电源故障时，eADR 会将 CPU 缓存的全部内容刷新到持久性内存中，因此在未来的处理器中，无需刷新 CPU 缓存中的缓存块即可保证持久化。此外，仍需使用栅栏来维持写入持久性内存的顺序 [35]。由于持久化数据的数据路径选择发生在刷新缓存行之前，因此本研究的结论可推广到配备 eADR 的未来系统。本研究聚焦于 GPU 的持久事务（撤销日志记录）。请注意，GPU 中还存在其他用于控制持久化顺序的持久性内存模型，但这些模型不在本研究的讨论范围内。

表 I 按使用时序路径和非时序路径进行撤销日志记录时的性能差异，对不同类型的基准测试进行分类。

Type	指标	基准测试
I	$perf^{T} > perf^{NT}$, $diff > 5\%$	SAD1、GRID1、GRID2, 2DCONV、Backprop1, Pathfinder，2MM1, 3MM1, GEMM
II	$perf^{T} < perf^{NT}$, $diff > 5\%$	SGEMM, Backprop2, ATAX1，ATAX2，NW, SSSP2，MVT1, GESUMMV, RA
III	$diff \leq 5\%$	Stencil1, SSSP1, StreamTriad，BFS

图 3 展示了采用静态数据路径选择进行撤销日志记录的 22 个 GPU 基准测试内核的实验结果。图 3 揭示了几个重要发现。**首先，**

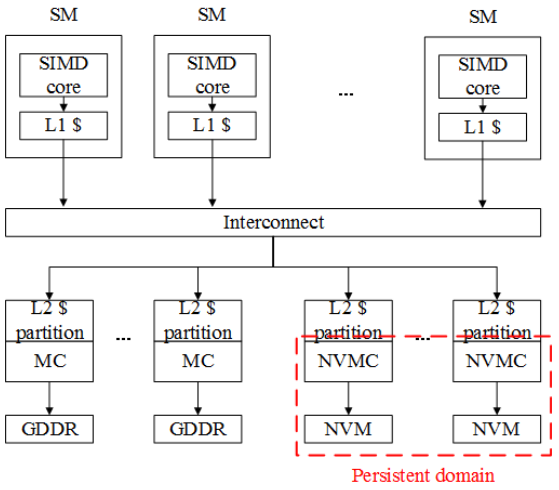

图 2. 配备 NVM/PM 的假设 GPU 系统概览。$ 表示缓存。MC 表示内存控制器。NVMC 表示非易失性内存控制器。

撤销日志记录带来的数据可恢复性，会给不同的 GPU 基准测试内核引入不同程度的性能开销。例如，调用 $nt\text{-}store$ 指令的持久内存操作会使 SAD1 内核的执行时间增至未进行撤销日志记录时的 325.5%，而 $nt\text{-}store$ 会使 SGEMM 的执行时间增至 102.7%。此外，我们可以看到，在若干基准测试内核（SAD1、GRID1、GRID2、2DCONV、Pathfinder、2MM1、3MM1、GEMM）中，$store{+}clwb$ 的执行时间短于 $nt\text{-}store$。这些结果引出了我们的第二个发现：**认为 $nt\text{-}store$ 更有利于撤销日志记录的假设并不适用于所有 GPU 应用程序**。根据图 3，我们依据两个指标（$perf$ 和 $diff$），将 22 个基准测试内核大致分为三类，如表 I 所示。$perf^{T}$ 表示使用时序路径（$store{+}clwb$）进行撤销日志记录时的基准测试性能（执行时间等），而 $perf^{NT}$ 表示使用非时序路径（$nt\text{-}store$）时的性能。$diff$ 表示 $perf^{T}$ 与 $perf^{NT}$ 之间的绝对差值。

由于本研究与 GPU 中的缓存绕过问题 [36, 37] 相似，我们尝试依据缓存绕过逻辑来解释基准测试内核的性能差异。该领域最重要的理念之一是根据 GPU 内核的内存合并行为 [36] 执行缓存绕过。这些研究得出结论，未合并的 $load$s/$store$s 应绕过易失性缓存，因为与其他内存指令相比，这类请求会产生大量内存访问。此外，这些请求获取的数据可能局部性较差，不值得消耗 GPU 架构中稀缺的缓存资源。图4展示了具有不同内存访问次数的 $load$s/$store$s（包括撤销日志和其他内存更新）的分布。我们可以看到，性能差异与图3所示的基准测试内核的合并行为并不一致。例如，StreamTriad、ATAX2 和 3MM1 的内存合并度都很高，其内存未合并度均在 $[1, 2]$ 以内。然而，如表 I 所述，这些内核的类型从 I 型到 III 型不等。因此，我们可以看出，本研究与 GPU 中的缓存绕过问题并不完全相同，需要采用新的设计。

为了说明使用不同数据路径时的性能差异，我们定义了 10 个变量，用于表征 GPU 内核的时间和空间局部性，如表 II 所述。例如，$l1d\ t\ all$ 表示 L1D 缓存中具有时间局部性的缓存命中数，这些访问由日志更新和其他数据更新共同产生。其他 7 个变量（$l1d$（$l2$）t（s）all（log））也遵循相同的定义。由于我们关注的是 GPU 持久内存中的持久事务，因此我们认为，日志更新（log）和非日志更新（all）的统计信息都是必需的。我们将 L1D 缓存中非日志更新的时间局部性定义为：日志更新之后进行非日志数据更新时，对特定字节的再次命中；将 L1D 缓存中日志更新的时间局部性定义为：一次日志更新之后再次进行日志更新时，对特定字节的再次命中。同样，我们将 L1D 缓存中非日志更新的空间局部性定义为：日志更新之后进行非日志数据更新时，对特定对齐的128字节段（缓存块）的再次命中；将 L1D 缓存中日志更新的空间局部性定义为：一次日志更新之后再次进行日志更新时，对特定字节的再次命中。L2 缓存的统计信息也采用相同的定义。请注意，Maxwell、Pascal 和 Volta GPU 架构使用按需获取缓存，仅获取所请求的数据块，而非获取完整缓存行，以解决数据过度获取问题 [38]。因此，GPU 缓存中具有时间局部性的访问次数不一定与具有空间局部性的访问次数相同。根据表 II，我们总结出 8 个原因来解释图 3 所示的性能差异，详见表III。

直观而言，$nt\text{-}store$ 指令被认为有利于日志更新 [13, 29]。这是因为日志仅被认为是实现崩溃一致性所必需的，在不会发生故障的执行过程中从不读取。在这种情况下，$nt\text{-}store$的写不分配策略会将数据块直接写入内存，而不将其加入缓存，因此更有利于日志记录，并能减少缓存污染。另一方面，$store{+}clwb$的写分配策略将频繁重复使用的数据块引入易失性缓存，有助于节省大量昂贵的内存事务。因此，撤销日志记录的数据路径选择有一种简单思路：依据日志更新与后续数据更新之间的局部性。更具体地说，局部性越强，就越可能采用时序路径（$store{+}clwb$）；否则，应选择非时序路径（$nt\text{-}store$）。然而，表 II 展示的模式与这一直觉并不相同。这引出了我们的第三个观察结果：$store{+}clwb$ **对于 GPU 缓存中局部性良好的数据更新而言，并不总是比 $nt\text{-}store$ 能带来更多好处**。由于 a、b和 d，主要差异取决于 $l1d\ t\ log$的值是否

– –

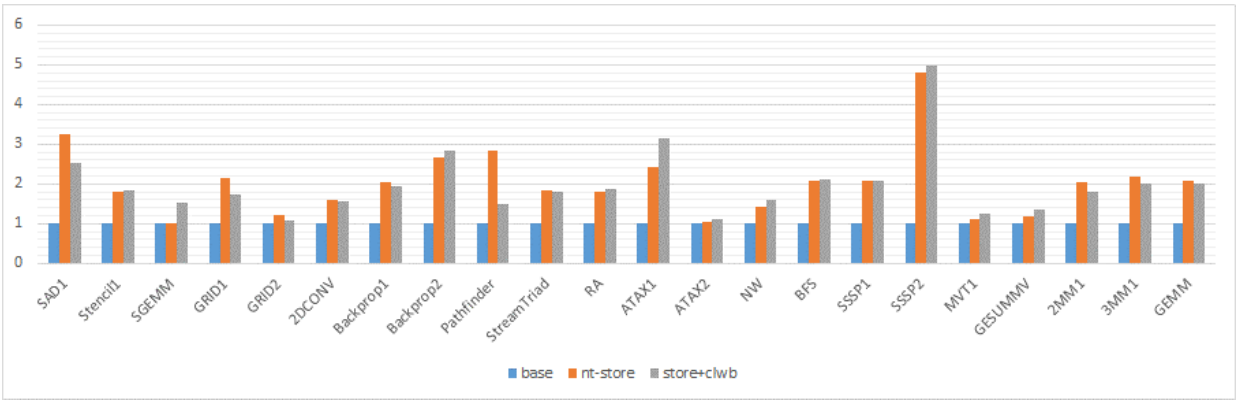

图 3. 使用独立数据路径进行撤销日志记录的 22 个 GPU 基准测试内核的归一化执行时间。ATAX1 和 ATAX2 分别表示 ATAX 基准测试的第一个和第二个内核。其他基准测试也同理。 *base* 表示这些基准测试在不支持持久性的 NVM 上执行。 *nt-store* 表示该实现遵循 [27, 28], 中的描述，使用非时序路径将日志更新持久化到 PM，并使用时序路径进行后续数据更新。 *store+clwb* 表示日志及其对应的数据更新均使用时序路径。所有结果均以使用 *base*运行时的执行时间进行归一化。

表 II

22 个 GPU 基准测试内核在 GPU 的 L1D 和 L2 缓存中的内存访问的时序局部性和空间局部性。 $l1d$ 表示 L1D 缓存。 $l2$ 表示 L2 缓存。 t表示时序局部性。 s 表示空间局部性。 all 表示由日志更新和其他数据更新共同产生的内存访问。log 表示由日志更新产生的内存访问。$type$表示表 I 中所述的内核类型。 $reason$ 表示表 III 中所述的某个内核属于某种类型的原因。

基准测试	*l1d_t_all*	*l1d_t_log*	*l1d_s_all*	*l1d_s* 日志	*l2_t* 全部	*l2_t* 日志	*l2_s* 全部	*l2_s* 日志	*all*	*log*	*type*	原因
SAD1	0	0	0	0	289160	157652	300509	167001	2288857	324803	I	c
Stencil1	0	0	0	0	0	0	0	0	131871	33849	III	f
SGEMM	0	0	0	0	0	0	0	0	3191435	133274	II	h
GRID1	2398	0	3126	0	7495	3483	9588	3930	1047435	662069	I	a
GRID2	47898	0	47898	0	140745	64156	140745	64156	591011	200163	I	a
2DCONV	0	0	0	0	0	0	0	0	79334	17060	III	f
Backprop1	0	0	7	6	0	0	153	147	88412	65076	I	g
Backprop2	8764	2671	9476	2987	8958	6330	9707	6920	280763	118159	II	b
Pathfinder	1221	97	1225	101	35000	26646	35843	27229	806640	382582	I	d
StreamTriad	0	0	0	0	0	0	0	0	36000	17476	III	f
RA	1	1	2	2	2	1	4	2	6644	2676	II	b
ATAX1	45460	45460	45460	45460	135526	89964	135526	89964	1926411	180170	II	b
ATAX2	89032	45562	134032	45562	135770	90000	135770	90000	545022	180170	II	b
NW	82	40	154	84	204	88	426	156	3195	1392	II	h
BFS	5	0	5	0	2	1	2	1	9528	79	III	e
SSSP1	0	0	0	0	1	0	1	0	1007	17	III	e
SSSP2	1028	771	1028	771	2570	1285	2570	1285	9509	6168	II	b
MVT	63720	63720	63720	63720	196227	130846	196227	130846	9444863	341037	II	b
GESUMMV	60929	60929	60929	60929	193639	129104	193639	129104	15752038	281725	II	b
2MM1	1639464	0	2809772	0	5608094	4905025	6369854	5416431	95599514	51796409	I	a
3MM1	28360	0	28608	0	441916	411072	449205	418361	7281497	4458165	I	a
GEMM	33545	2031	34045	2031	1533475	1501161	1559512	1527197	7332760	4503201	I	d

以及 $l1d\_s\_log$ 是否等于 0。 $l1d\_t\_log$ 和 $l1d_s_log$揭示了日志更新是否反复引用相同的缓存块或相同的缓存行。当这两个变量等于 0（原因 a），或仅占具有局部性的所有内存访问中的很小比例（≤ 0.25）（原因 d）时，实验结果符合上述直觉：使用 *store+clwb* 指令比使用 *nt-store*（I型）性能更好。然而，当日志更新导致的 L1D 缓存访问中出现过强的局部性时（原因 b），我们可以看到，在这种情况下， *nt-store*相比 *store+clwb*能带来更多收益。这是因为 L1D 缓存之间的缓存颠簸

而通过速度较慢的互连网络访问 L2 缓存，会阻碍利用缓存处理局部性良好数据所带来的性能优势。图 5 显示了使用时间性（*store+clwb*）和非时间性（*nt-store*）数据路径时，属于某一特定原因的内核的归一化平均传输字节数。随着易失性缓存中的数据局部性增强，由于 warp 停顿减少且昂贵的内存事务减少，内存流量和流水线利用率都会提高。然而，当局部性变得过强，且重用距离过长，以致缓存数据在被驱逐之前无法得到重用时，GPU 易失性缓存的失效频率将急剧上升

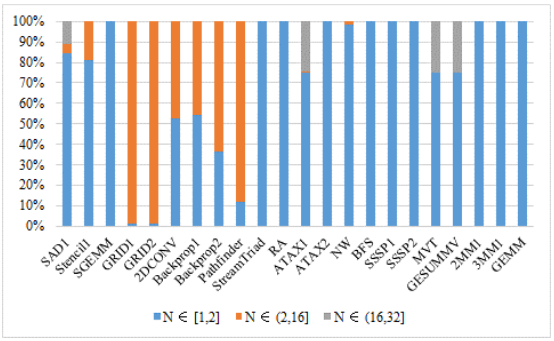

图 4. 不同内存非合并程度下 *loads*/*stores* 的分布。 N 表示内存合并后，同一线程束中属于 *load*/*store* 的内存访问次数。

表III 基于统计的推理。

原因	指标	*Type*
a	$l1d\_t\_all \neq 0$ *and* $l1d\_s\_all \neq 0$ *and* $l1d\_t\_log = 0$ *and* $l1d\_s\_log = 0$	I
b	$l1d\_t\_all \neq 0$ *and* $l1d\_s\_all \neq 0$ *and* $l1d\_t\_log \neq 0$ *and* $l1d\_s\_log \neq 0$ *and* $l1d\_t(s)\_log \,/\, l1d\_t(s)\_all > 0.25$	II
c	$l1d\_t\_all = 0$ *and* $l1d\_s\_all = 0$ *and* $l1d\_t\_log = 0$ *and* $l1d\_s\_log = 0$ *and* $l2\_t\_all \neq 0$ *and* $l2\_s\_all \neq 0$ *and* $l2\_t\_log \neq 0$ *and* $l2\_s\_log \neq 0$	I
d	$l1d\_t\_all \neq 0$ *and* $l1d\_s\_all \neq 0$ *and* $l1d\_t\_log \neq 0$ *and* $l1d\_s\_log \neq 0$ *and* $l1d\_t(s)\_log \,/\, l1d\_t(s)\_all \leq 0.25$	I
e	$log \leq 100$	III
f	$l1d\_t\_all = 0$ *and* $l1d\_s\_all = 0$ *and* $l1d\_t\_log = 0$ *and* $l1d\_s\_log = 0$ *and* $l2\_t\_all = 0$ *and* $l2\_s\_all = 0$ *and* $l2\_t\_log = 0$ *and* $l2\_s\_log = 0$	II、III
g	$l1d\_t\_all = 0$ *and* $l1d\_s\_all \neq 0$ *and* $l1d\_t\_log = 0$ *and* $l1d\_s\_log \neq 0$	I
h	共享内存的使用	II

最终导致缓存争用和颠簸。对于属于原因 c 的内核，由于在 L1D 缓存中未观察到局部性，因此在享受 *store+clwb* 带来的优势时，不存在发生严重颠簸的风险；这些内核被归类为 I型。对于原因 f，内核在 L1D 缓存和 L2 缓存中均不具有数据局部性。由于采用写不分配策略， *nt-store* 直观上被认为适合这种模式。然而，这些 GPU 内核表现出一种有趣的行为：使用 *store+clwb* 和 *nt-store* 的性能相近（III型）。我们认为，这是因为 GPU 的大规模多线程能够有效隐藏缓存分配开销，因此无论使用时间性路径还是非时间性路径来实现持久性，平均性能都相当。对于原因 g，在 L1D 缓存和 L2 缓存中都观察到中等程度的局部性，因此属于这一类别的内核为 I型。对于原因 e 和原因 h，这是两个特殊情况，无法通过日志更新与非日志更新的局部性，以及时间性或非时间性数据路径之间的适应性来解释。对于原因 e，日志更新的数量过于

数量较少（<100），因此使用 *store+clwb* 与 *nt-store* 指令（III 型）之间的差异可以忽略不计。对于原因 h，这些内核使用共享内存进行通信，并同步同一 CTA（TB）内的线程。在这种情况下，从易失性缓存中提取的大多数数据不会被重复使用，因此使用 *nt-store* 指令比使用 *store+clwb* 指令（II 型）更有利。

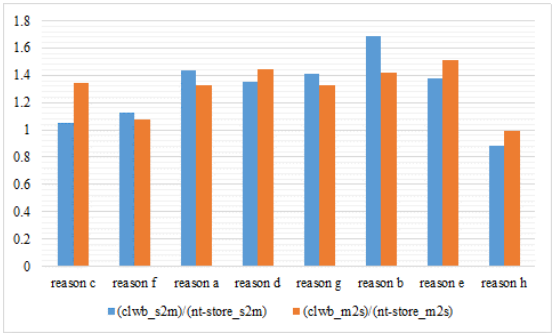

图 5. 使用 *store+clwb* 和 *nt-store* 指令时，GPU L1D 缓存与 L2 缓存之间传输字节数的归一化均值对比。$s2m$ 表示从 L1D 缓存到 L2 缓存的传输， $m2s$ 表示从 L2 缓存到 L1D 缓存的传输。

IV. 我们的解决方案

如第三节所述，我们利用 10 个变量和 8 个原因来解释 GPU 撤销日志记录在使用 *store+clwb* 和 *nt-store* 指令时的性能差异。基于这些观察，我们提出了 GPU 持久内存中持久事务的自适应数据路径选择方案（AGPM）。其核心思想是根据统计信息，自适应地改变日志更新的数据路径；这些统计信息用于跟踪 GPU L1D 缓存和 L2 缓存中日志更新与非日志更新之间的局部性。当 L1D 缓存与 L2 缓存之间存在中等局部性，或不存在通过 GPU 互连网络在不同级别的易失性缓存之间发生争用的风险时，应为日志更新选择时间性数据路径（*store+clwb*）。当多个 GPU 缓存中同时存在过强的局部性时，应为日志更新选择非时序路径。对于 GPU 内存层次结构中不存在局部性的情况，路径选择较为灵活，建议使用非时序路径，以便将稀缺的缓存资源留给局部性较高的其他数据。对于使用共享内存或日志更新数量过少的特殊情况，建议采用非时序路径，以避免浪费易失性缓存。

A. 基于周期的策略

尽管我们希望利用第三节所述的观察结果来实现自适应数据路径选择，但表 II 中的结果并不适合直接用于运行时执行，因为这些结果是在内核执行完成后记录的。为了通过捕捉表 II 所揭示的模式，使撤销日志记录能够自适应地选择路径，我们在内核执行期间分阶段收集统计信息，并使用这些统计信息

为图6所示的路径选择器提供输入。第四节对此作了进一步讨论。

为所有类型的 GPU 内核确定一个基于周期数的周期颇具挑战。这是因为不同 GPU 内核的执行时间可能相差很大，同一内核中不同协作线程阵列（CTA）的执行时间也可能大相径庭，从而导致 GPU [31] 中出现不同层级（内核级、CTA 级、迭代级等）的内存持久性模型。本研究将一个周期定义为向持久性内存（PM）写入日志更新次数的阈值。当日志数量超过阈值时，所有 AGPM 缓冲区都会被刷新，并在新周期中重新捕获数据局部性。根据表 II，我们将该阈值初始化为 10000。我们定义了一个指标——每个 PM 请求等待的周期数（CWPPR），用于更新阈值。CWPPR 表示每个 PM 请求完成服务所需的平均周期数。我们利用这一指标并采用一种采样方法，在每个周期完成时更新阈值。我们将当前周期（P_1）定义为采样周期。如果采样周期（P_1）的 CWPPR 大于前一个周期（P_0），则认为阈值长度不足以捕获正确的模式。在这种情况下，阈值增加前一阈值的 10%。例如，如果前一阈值为 10000，则更新后的阈值为 11000。否则，阈值减少前一阈值的 10%。最后，更新后的阈值会被存储起来，并用于下一个采样周期。在某些情况下，如果整个 GPU 内核向 PM 写入的日志更新数量少于 10000，则会将阈值初始化为一个较小的值。

*B. AGPM*架构

图6展示了 AGPM 的整体架构。GPU 流式多处理器内部的 SIMD 核心与 L1缓存之间的灰色菱形表示路径选择器，它决定来自持久内存日志更新的内存请求采用哪条特定数据路径到达持久化域（本研究中为非易失性内存控制器）。连接到 L1缓存和 L2 缓存分区的灰色方块表示 AGPM 缓冲区，用于捕获日志更新与后续数据更新之间的局部性。连接到非易失性内存的灰色方块表示预留缓冲区，用于保存从 AGPM 缓冲区刷出的信息。蓝色和红色实线分别表示时间性数据路径和非时间数据路径。绿色虚线表示控制信号，用于指示撤销日志记录应选择哪条路径。请注意，图6展示了一个流式多处理器与一个内存分区之间的 AGPM 相关内存事务；在 GPU 中，不同流式多处理器与不同非易失性内存分区之间也可以执行这些事务。

在加载/存储单元中合并后，内存指令发出的持久内存请求可能需要访问 L1D 缓存。在 AGPM 中，这些请求会先发送至路径选择器。根据请求地址以及记录在 L1D 和 L2 缓存中扩展的 AGPM 缓冲区内的相应统计信息，如果观察到与表 III 中所述原因 a、 c、 d和 g 相同的模式，则执行持久内存指令的同一线程束中的请求会访问 GPU 的易失性缓存。由于假定非易失性内存控制器（NVMC）支持 ADR，因此，当脏缓存行从 L2 缓存分区发送至 NVMC 时，数据便会持久化。否则，如果观察到与原因 b、 e、 f、 g和 h 相同的模式，请求将绕过缓存，并直接插入 NVMC 中的写入待处理队列（WPQ）。在这种情况下，数据一旦到达 NVMC 即会持久化，无需采取额外操作。

AGPM 缓冲区的结构类似于一个 2 路组相联缓存（如图6所示）。一路用于记录专属持久内存日志更新的统计信息（log 路），另一路用于记录各类内存请求的统计信息（all 路）。在每一路中，组相联缓存的每个条目都有两个字段：标签和数据。标签是按 128 字节对齐的缓存块地址，数据是一个计数器向量，用于记录对缓存块中每个字节的引用次数。除了字节计数器外，数据字段中还使用了两个额外的计数器。一个额外计数器用于记录对特定缓存块的引用次数。另一个额外计数器用于标记数据路径的变化。如果某次日志更新的数据路径从时态路径变为非时态路径，则应丢弃原有的 $clwb$ ，因为数据会直接发送到持久内存控制器。反之，如果数据路径从非时态路径变为时态路径，则需要在时态 $store$之后添加一个类似 $clwb$ 的操作，以确保数据持久化。发出新的持久内存请求时，AGPM 缓冲区的两路中都会添加一个关联条目，并将其初始化为 0。请求涉及的字节计数器会加 1。之后，对于后续的内存请求（包括持久内存请求和其他请求），页表遍历器（PTW）会开始遍历。一旦遍历器确认请求命中，就会将字节地址通知 AGPM 缓冲区。随后计算缓存块地址和偏移量。AGPM 缓冲区会搜索缓存块地址对应的条目。如果 AGPM 缓冲区未命中，且请求并非来自持久内存指令，则不执行任何操作。如果命中，则根据偏移量进行处理：若对应的字节计数器为 0，则只增加缓存块计数器；若字节计数器大于 0，则同时增加字节计数器和缓存块计数器。如果后续请求属于持久内存指令，则更新 AGPM 缓冲区两路中的计数器；否则，仅更新 all 路中的计数器。当查询 AGPM 缓冲区以选择路径时，会根据记录的计数器计算表III中所述的变量。例如， $l1d\ t\ all$ 是 L1 缓存级别的 AGPM 缓冲区 all 路中所有值大于 1 的字节计数器之和；$l1d\ s\ all$ 是 L1 缓存级别的 AGPM 缓冲区 all 路中所有值大于 1 的字节计数器与缓存块计数器之和。表中描述的其他变量也以此类推。

– –

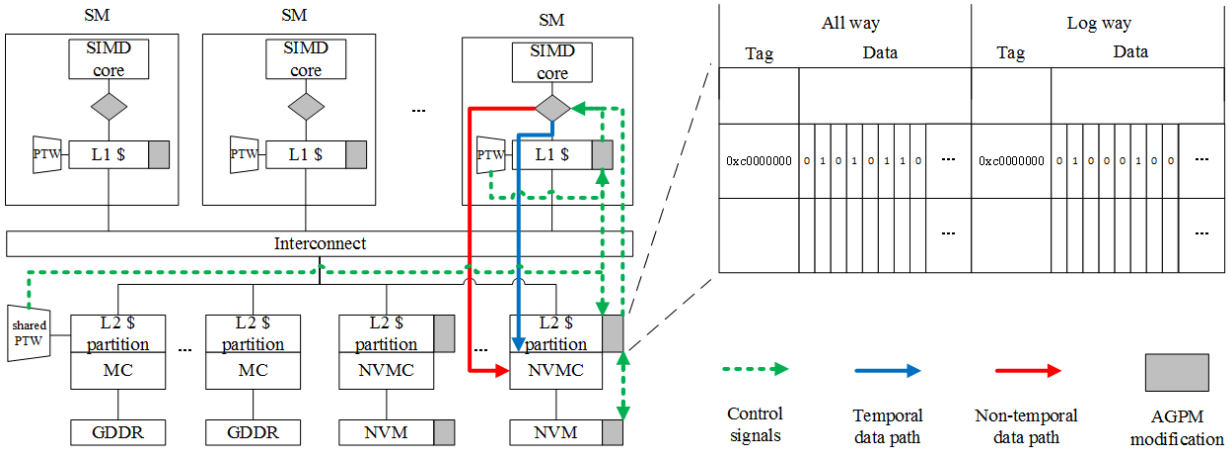

图 6. AGPM 架构概览。

III.

当一个缓存块被刷新到较低层内存层次结构时，其关联的 AGPM 条目也会被刷新到较低层的 AGPM 缓冲区。当 AGPM 缓冲区中记录的缓存块最终因常规替换或类似 $clwb$ 的指令而被写回时，AGPM 缓冲区中的关联内容会先存入位于 GPU内存中的保留缓冲区，然后再从 AGPM 缓冲区中刷新出去。在当前周期或内核执行完成之前，如果该缓存块再次被取回易失性缓存，则会将 GPU内存中保留的信息重新填入 AGPM 缓冲区。当一个周期结束或内核执行完成时，AGPM 缓冲区和保留缓冲区中的所有内容都会被刷新。

C. 开销分析

我们使用一个包含 1024 个条目的 2 路 AGPM 缓冲区，来记录日志更新与后续数据更新之间的局部性。每个条目都有一个标签和一个数据字段。假设系统为 64 位宽，则标签为 57 位。数据包含 128+2=130 个计数器，在大多数情况下，5 位计数器即可满足需求。因此，数据字段总共需要 650 位。每个条目需要 707 位（约 89 B），1024 个条目对应 89 KB 的存储开销。由于不同流式多处理器之间的内存访问模式几乎相同，乐观估计只需实现 1 个 AGPM 缓冲区，即可跟踪 L1 级数据局部性。还需要另一个 AGPM 缓冲区，供所有 L2分区共享。总开销为 89*2=178 KB。

V. 评估

我们通过将 AGPM 与采用静态路径策略进行撤销日志记录的传统内存持久性模型进行比较，来评估 AGPM。此外，我们还比较基准测试在使用 AGPM 与使用其他最先进的持久内存多数据路径架构及缓存绕过方法时的归一化执行时间，以验证所提出方案在 GPU 持久内存应用中的可行性。

A. 评估方法

我们使用由 [31]实现的 GPGPU-Sim 扩展。该扩展为针对 GPU 重新设计的内存持久性模型提供功能和时序模拟支持。这些模型根据 GPU 工作负载的特征、GPU 对带宽敏感的特性以及 GPU 的内存层次结构进行了调整和优化。为了利用该扩展，编译器使用内联汇编插入持久内存指令，例如 $clwb$ 和 $sfence$，并对模拟器进行修改以支持这些指令的语义。在我们的实验中，持久事务由基于软件的撤销日志记录提供支持。具体而言，在事务开始之前，通过复制待更新的数据创建撤销日志，并将该日志持久化。随后设置并持久化一个标志，以表明事务正在进行。在事务执行期间，数据会被更新并持久化。事务完成后，标志会更新，撤销日志也会被释放。采用撤销日志记录时，恢复代码会检查该标志以确定事务状态。如果事务在完成之前中断，则会应用撤销日志来恢复数据。

我们的实验采用了来自 Rodinia [39]，Lonestar [40]，Polybench [41],和 Parboil [42] GPU 基准测试套件的一组常规和非常规应用。这些基准测试经过修改，使用持久内存指令来构建持久事务，其实现方式与 [31] 类似。GPGPU-Sim 的模拟配置如表 IV 所示。

B. 性能比较

图 7 展示了 22 个 GPU基准测试内核在撤销日志记录中采用不同数据路径选择策略时的性能比较。 $nt\text{-}store$ 和 $store{+}clwb$表示对所有日志更新采用静态策略，即使用非时间数据路径或时间性数据路径。Themis [27]为持久性内存提出了一种多数据路径架构，可区分时序存储和非临时存储。Themis 使用非时间数据路径作为通往持久性内存的快速存储路径，而时序存储

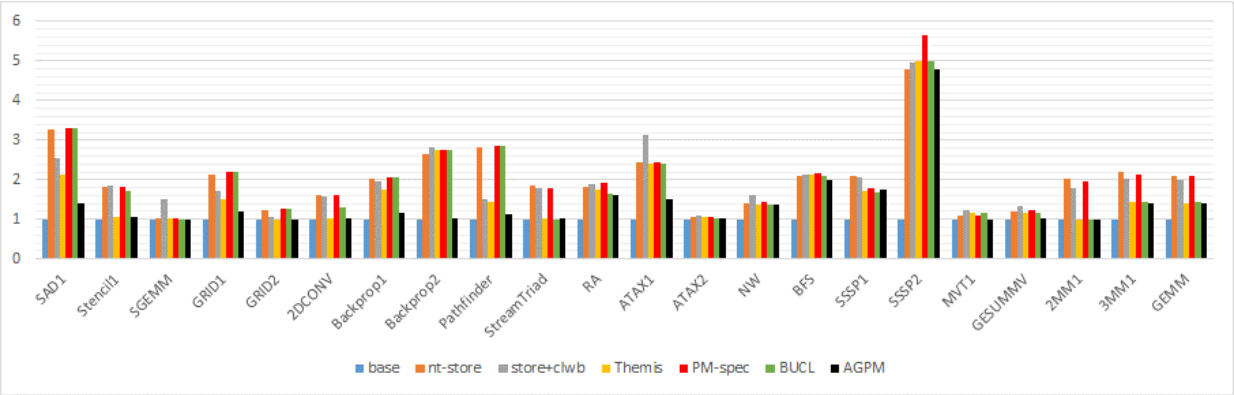

图7. 采用不同撤销日志记录数据路径选择策略的22个GPU基准测试内核的归一化执行时间。所有结果均以使用 $base$运行时的执行时间进行归一化。

表 IV GPGPU-Sim 的配置参数。

GPU核心	20个流式多处理器，SIMD宽度=32，1.8GHz
着色器核心配置	最多 2048 个线程、64 个线程束和 32 个协作线程阵列 每个流式多处理器配备 32 个线程/线程束、4 个 GTO 调度器
每个流式多处理器的L1D缓存	24KB、128B 缓存行、6 路组相联， 256 个未完成暂存请求
每个流式多处理器的共享内存	96KB，32 个存储体
共享 L2 缓存	2048KB，128KB/分区，128B 缓存行， 16 路组相联，256 个未完成暂存请求
一级数据缓存／二级缓存策略	异或索引、未命中时分配、最近最少使用， 一级数据缓存：写到逐、写不分配，二级缓存：写回、写分配
互连	16*16 交叉开关，32B 流控单元大小，1.4GHz
内存控制器	8 个通道，每个通道 2 个 L2 缓存组， FR-FCFS 调度器，1.2GHz，带宽：307GB/s
非易失性存储器延迟	读取：160ns，写入：480ns
DRAM 延迟	读取：160ns，写入：160ns

则使用较慢的数据路径。由于不同路径的延迟存在差异，Themis 几乎可以消除所有持久化屏障，从而提升持久化应用程序的性能。PM-spec [28]也为持久性内存提出了一种多数据路径架构。与 Themis 不同，PM-spec 允许持久内存控制器分别通过独立的数据路径接收持久内存加载和存储操作。具体而言，持久内存加载使用时间性数据路径，而持久内存存储操作则通过非时间数据路径进行。BUCL [36] 是一种针对未合并加载的 GPU 缓存绕过方案。如果未合并内存访问的数量大于合并度阈值，则选择非时间数据路径；否则，使用时间性数据路径。

根据图 7，使用不同撤销日志记录策略时的平均归一化执行时间分别为 $nt\text{-}store$：203.39%、 $store\text{+}clwb$：198.43%、Themis：165.22%、PM-spec：207.29%、BUCL：185.60%和AGPM：140.85%。可以看出，即使与静态策略的实验结果相比（$nt\text{-}store$、 $store\text{+}clwb$），PM-spec 的归一化执行时间也是最长的。这是因为由持久内存日志更新产生的大多数内存访问都是 $store$ 操作。在这种情况下，按照 $load$和 $store$ 分离数据路径并不适用于撤销日志记录。因此，结果是使用 PM-spec 的性能（207.29%）与 $nt\text{-}store$ （203.39%）接近。除了静态策略和 PM-spec 之外，BUCL 的归一化执行时间排名第二长。原因之一是，BUCL 的旁路逻辑主要依赖于 GPU 工作负载的合并度。根据图4和表 I，我们可以看到，GPU 工作负载的合并度并不总是与其对应的类型相符。例如，根据 BUCL 的标准以及图4所示的合并度，SAD1、Stencil1、GRID1、GRID2、2DCONV、Backprop1、Backprop2、Pathfinder、ATAX1、MVT1 和 GESUMMV 属于内存访问未合并应用。然而，SAD1、GRID1、GRID2、Backprop1 和 Pathfinder 属于 I 型。Stencil1 和 2DCONV 属于 II 型。Backprop2、ATAX1、MVT1 和 GESUMMV 属于 III 型。在这种情况下，使用 BUCL 进行数据路径选择，在用于 GPU 缓存绕过时，无法为持久内存日志更新带来同等程度的性能提升。此外，BUCL 并非针对持久性内存应用而设计，这可能导致其与撤销日志事务流程不兼容，因为该流程要求日志更新先于对应的数据更新达到持久化状态。另一方面，BUCL 在部分基准测试（StreamTriad、SSSP1）中取得了最佳性能。这是因为，当内存请求访问易失性缓存后发生预留失败时，BUCL 会将该请求发送至缓存绕过逻辑。这一特性可防止互连网络和 GPU 内存层次结构因缓存拥塞或停滞而未得到充分利用。在所有策略中，Themis 的性能排名第二。Themis 根据 $store$ 的时间局部性做出数据路径选择决策，这与 AGPM 类似。在这种情况下，除 Backprop2、ATAX1、ATAX2、MVT1 和 GESUMMV 外，Themis 在大多数基准测试中都取得了相对较短的归一化执行时间。这些基准测试属于表III中描述的原因 b，而 Themis 无法察觉数据局部性过强可能导致的缓存颠簸。在这种情况下，时间数据路径和非时间数据路径无法区分，从而导致使用 Themis 时性能受到负面影响。请注意，Themis 的一项关键优势是

消除持久性模型中几乎所有的持久化屏障。由于 $sfence$ 指令的排序语义会阻止内存访问的缓冲和重排，以及乱序执行，因此这一特性能够显著提升 CPU 工作负载的性能。然而，对于 GPU 而言，线程高度重叠，且其内存延迟可通过大规模多线程加以隐藏，因此在 GPU 内核中移除持久化屏障所产生的影响，不如在 CPU 工作负载中显著（平均为 1.5%[31]）。

VI. 相关工作

英特尔提供 PMDK [8], 库，以利用 Optane NVM[14]。Haria 等人 [17] 提出了一个 C++ 库，使对一个或多个数据结构的多次更新能够相对于故障实现原子性。Hsu 等人 [18] 提出了一种编程模型和运行时，为现有的多线程 C/C++ 程序添加持久性。Liu 等人 [43] 提出了 Janus，这是一种软硬件协同设计，可在 NVM 系统中并行化并预执行后端内存操作。Mnemosyne[6], 、NV-Heaps [16], 和 NVRAM 事务 [26]探索了与持久性内存进行事务性交互的技术。检查点通常用于容错。以往关于 NVM 检查点的研究[22, 23]主要致力于最小化检查点延迟和带宽开销。

Lin 等人 [31] 将现有的 CPU 持久性模型适配到 GPU。DRAGON [44] 通过统一内存利用 NVM 为 GPU 提供的能力。GPM [21] 利用 UVA 将持久性内存映射到 GPU 的地址空间，从而在无需 CPU 帮助的情况下，为 GPU 内核提供细粒度持久性。

Nalli 等人 [13] 提出了 WHISPER（一套 PM 基准测试套件）和 HOPS（一种非干预式持久化系统）。HOPS 在硬件中跟踪对 PM 的更新，并提供高级 ISA 原语，使应用程序能够分别表达持久性和排序约束。Themis [27] 利用独立的数据路径，在不使用干预性栅栏指令的情况下提供隐式排序保证。DPO [45] 和 PMEM-spec [28] 利用专用的持久化路径处理 PM $store$s，并通过常规路径处理 PM $load$s。

VII. 结论

本文提出了一种自适应方法（AGPM），用于选择 GPU 中持久性内存日志更新的数据路径。我们首先介绍了对采用静态数据路径选择策略的 GPU 内核进行实验比较后得出的观察结果。我们发现，对于撤销日志记录而言，使用非时序路径更有利这一假设并不适用于 GPU 工作负载。此外，我们还发现，与后续数据更新具有较强局部性的日志更新，并不总是适合采用时间性数据路径。我们分析了 GPU 内核中的日志更新与后续数据更新之间的数据局部性，以及独占日志更新之间的数据局部性，考察范围涵盖 GPU 的 L1D 和 L2 缓存。我们将这些分析归纳为 8 个原因，并以此指导所提方案中的数据路径选择。我们实现了 L1 级和 L2 级 AGPM 缓冲区，用于记录 GPU 内核运行时执行过程中的统计信息；随后，我们将这些统计信息输入每个流式多处理器中的 AGPM 路径选择器，以匹配归纳出的原因。根据匹配结果，AGPM 会自适应地改变日志更新写入持久性内存的数据路径。实验结果表明，对于不同类型的 GPU 工作负载，AGPM 的性能优于最先进的持久性内存多数据路径架构。

致谢

TBD

参考文献

[1]J. Arulraj、A. Pavlo、S. R. Dulloor，《谈谈非易失性内存数据库系统的存储与恢复方法》，载于：2015 年 ACM SIGMOD 数据管理国际会议论文集，2015 年，第 707–722 页。[2] J. Condit、E. B. Nightingale、C. Frost、E. Ipek、B. Lee、D. Burger、D. Coetzee，《通过字节寻址持久内存改进 I/O》，载于：第 22 届 ACM SIGOPS 操作系统原理研讨会论文集，2009 年，第 133–146 页。[3] S. R. Dulloor、S. Kumar、A. Keshavamurthy、P. Lantz、D. Reddy、R. Sankaran、J. Jackson，《持久内存系统软件》，载于：第九届欧洲计算机系统会议论文集，2014 年，第 1–15 页。[4] A. Joshi、V. Nagarajan、M. Cintra、S. Viglas，《面向多核处理器的高效持久化屏障》，载于：第 48 届国际微体系结构研讨会论文集，2015 年，第 660–671 页。[5] J. Ren、J. Zhao、S. Khan、J. Choi、Y. Wu、O. Mutlu，《Thynvm：在持久内存系统中实现对软件透明的崩溃一致性》，载于：第 48 届国际微体系结构研讨会论文集，2015 年，第 672–685 页。[6]H. Volos、A. J. Tack、M. M. Swift，《Mnemosyne：轻量级持久内存》，《ACM SIGARCH 计算机体系结构新闻》39（2011 年），第 91–104 页。[7] J. Zhao、S. Li、D. H. Yoon、Y. Xie、N. P. Jouppi，《Kiln：缩小具备与不具备持久化支持的系统之间的性能差距》，载于：第 46 届 IEEE/ACM 微体系结构国际年会论文集，2013 年，第 421–432 页。[8] 英特尔 N. L. T.，2023，持久内存编程博客，网址：pmem.io。[9] H. P. Enterprise，2023，持久内存，网址：hpe. com/us/en/servers/persistent-memory.html。[10]I. Newsroom，2015，英特尔和美光推出突破性存储技术，网址：newsroom. intel.com/community/intel newsroom/blog/2015/07/28/
i
n
t
el-
and-micron-produce-breakthrough-memory-technolo
gy.
[11] SNIA，2023年，NVM 编程技术工作组，网址：snia.org/forums/sssi/nvmp。[12]I. B. Peng、M. B. Gokhale、E. W. Green，英特尔傲腾字节可寻址 NVM 的系统评估，载于：

《国际存储系统研讨会论文集》，2019，页码 304–315。[13] S. Nalli、S. Haria、M. D. Hill、M. M. Swift、H. Volos、K. Keeton，使用 WHISPER 的持久内存应用分析，《ACM SIGPLAN Notices》52（2017）135–148。[14]英特尔，2023年，英特尔傲腾持久内存，网址：
https://www.intel.in/content/www/in/en/architecture-and-technology/optane-dc-persistent-memory.html。[15]V. Chidambaram、T. S. Pillai、A. C. Arpaci-Dusseau、R. H. Arpaci-Dusseau，乐观崩溃一致性，载于：《第二十四届 ACM 操作系统原理研讨会论文集》，2013，页码 228–243。[16]J. Coburn、A. M. Caulfield、A. Akel、L. M. Grupp、R. K. Gupta、R. Jhala、S. Swanson，NV-Heaps：利用新一代非易失性内存实现快速、安全的持久对象，《ACM SIGARCH 计算机体系结构新闻》39（2011）105–118。[17] S. Haria、M. D. Hill、M. M. Swift，MOD：面向持久内存的最小有序持久数据结构，载于：《第二十五届编程语言与操作系统体系结构支持国际会议论文集》，2020年，页码 775–788。[18] T. C.-H. Hsu、H. Br¨ugner、I. Roy、K. Keeton、P. Eug-ster，NVthreads：面向多线程应用程序的实用持久化，载于：《第十二届欧洲计算机系统会议论文集》，2017年，页码 468–482。[19] R. Kadekodi、S. K. Lee、S. Kashyap、T. Kim、A. Kolli、V. Chidambaram，SplitFS：降低持久内存文件系统的软件开销，载于：《第 27 届 ACM 操作系统原理研讨会论文集》，2019，页码 494–508。[20]S. K. Lee、J. Mohan、S. Kashyap、T. Kim、V. Chidambaram，Recipe：将并发动态随机存取存储器索引转换为持久内存索引，载于：《第 27 届 ACM 操作系统原理研讨会论文集》，2019，页码 462–477。[21] S. Pandey、A. K. Kamath、A. Basu，GPM：利用 GPU 的持久内存，载于：《第 27 届 ACM 编程语言与操作系统体系结构支持国际会议论文集》，2022，页码 142–156。[22] H. Elnawawy、M. Alshboul、J. Tuck、Y. Solihin，面向非易失性主存的基于循环代码的高效检查点，载于：2017 年第 26 届并行体系结构与编译技术国际会议（PACT），IEEE，2017，页码 318–329。[23] M. Alshboul、J. Tuck、Y. Solihin，惰性持久化：一种高性能、写入高效的软件持久化技术，载于：2018 年 ACM/IEEE 第 45 届计算机体系结构年度国际研讨会（ISCA），IEEE，2018，页码 439–451。[24] R.-S. Liu、D.-Y. Shen、C.-L. Yang、S.-C. Yu、C.-Y. M. Wang，NVM Duet：统一工作内存与持久存储的体系结构，《ACM SIGARCH 计算机体系结构新闻》42（2014）455–470。[25] J. Ren、J. Zhao、S. Khan、J. Choi、Y. Wu、O. Mutlu，《Thynvm：在持久内存系统中实现软件透明的崩溃一致性》，载于：第 48 届国际微体系结构研讨会论文集，2015年，第672–685页。[26] A. Kolli、S. Pelley、A. Saidi、P. M. Chen、T. F. Wenisch，《面向持久内存的高性能事务》，载于：第二十一届编程语言与操作系统体系结构支持国际会议论文集，2016年，第399–411页。[27]S. M. Shahri、S. A. V. Ghahani、A. Kolli，《（几乎）无栅栏持久化顺序》，载于：2020年IEEE/ACM第53届国际微体系结构年会（MICRO），IEEE，2020年，第539–554页。[28]J. Jeong、C. Jung，《Pmem-spec：持久内存推测（严格持久性可以胜过宽松持久性）》，载于：第26届ACM编程语言与操作系统体系结构支持国际会议论文集，2021年，第517–529页。[29]A. Joshi、V. Nagarajan、S. Viglas、M. Cintra，《Atom：通过硬件日志记录实现非易失性内存中的原子持久性》，载于：2017年IEEE高性能计算机体系结构国际研讨会（HPCA），IEEE，2017年，第361–372页。[30] S. Pelley、P. M. Chen、T. F. Wenisch，《内存持久性：面向字节寻址非易失性内存技术的语义》，IEEE Micro 35（2015年），第125–131页。[31] Z. Lin、M. Alshboul、Y. Solihin、H. Zhou，《探索GPU的内存持久性模型》，载于：2019年第28届并行体系结构与编译技术国际会议（PACT），IEEE，2019年，第311–323页。[32]C. webinar，2011年，《点对点与统一虚拟寻址》，URL：https://developer.download.nvidia.com/CUDA/training/cuda webinars GPUDirect uva.pdf。[33] A. M. R. [无日期。], 2016, 弃用pcommit指令，URL：https://software.intel.com/blogs/2016/09/12/deprecate-pcommit-instruction。[34]英特尔，2021年，《eADR：持久内存应用的新机遇》，URL：
https://software.intel.com/content/www/us/en/develop/articles/eadr-new-opportunities-for-persistent-memory-application
s.html.[35]英特尔，2021，《持久性内存深入了解系列》第 2 部分，网址：https://software.intel.com/content/www/us/en/developer/articles/training/pmem-learn-more-series-part-2.html。[36] C. Zhao、F. Wang、Z. Lin、H. Zhou、N. Zheng，《针对未合并加载选择性绕过 GPU 缓存》，载于：2016 年 IEEE 第 22 届并行与分布式系统国际会议（ICPADS），IEEE，2016 年，第 908–915 页。[37] X. Xie、Y. Liang、Y. Wang、G. Sun、T. Wang，《面向 GPU 的协同静态与动态缓存绕过》，载于：2015 年 IEEE 第 21 届高性能计算机体系结构国际研讨会（HPCA），IEEE，2015 年，第

76–88。[38] W. Jia、K. A. Shaw、M. Martonosi，《GPU 中需求获取缓存使用情况的特征分析与改进》，载于：第 26 届 ACM 超级计算国际会议论文集，2012 年，第 15–24 页。[39] S. Che、M. Boyer、J. Meng、D. Tarjan、J. W. Sheaffer、S.-H. Lee、K. Skadron，《Rodinia：面向异构计算的基准测试套件》，载于：2009 年 IEEE 国际工作负载特征分析研讨会（IISWC），IEEE，2009 年，第 44–54 页。[40]M. Kulkarni、M. Burtscher、C. Casc.aval、K. Pingali，《Lonestar：并行非规则程序套件》，载于：2009 年 IEEE 系统与软件性能分析国际研讨会，IEEE，2009 年，第 65–76 页。[41]J. Karimov、T. Rabl、V. Markl，《Polybench：首个面向多模态存储系统的基准测试》，载于：性能评估与基准测试技术会议，施普林格，2019 年，第 24–41 页。[42] J. A. Stratton、C. Rodrigues、I.-J. Sung、N. Obeid、L.-W. Chang、N. Anssari、G. D. Liu、W.-m. W. Hwu，《Parboil：面向科学与商业吞吐量计算的修订版基准测试套件》，可靠计算与高性能计算中心 127（2012）27。[43]S. Liu、K. Seemakhupt、G. Pekhimenko、A. Kolli、S. Khan，《Janus：优化非易失性内存系统的内存与存储支持》，载于：2019 年 ACM/IEEE 第 46 届计算机体系结构年度国际研讨会（ISCA），IEEE，2019 年，第 143–156 页。[44]P. Markthub、M. E. Belviranli、S. Lee、J. S. Vetter、S. Matsuoka，《DRAGON：通过直接访问非易失性内存突破 GPU 内存容量限制》，载于：SC18：高性能计算、网络、存储与分析国际会议，IEEE，2018 年，第 414–426 页。[45] A. Kolli、J. Rosen、S. Diestelhorst、A. Saidi、S. Pelley、S. Liu、P. M. Chen、T. F. Wenisch，《委托持久化排序》，载于：2016 年第 49 届 IEEE/ACM 微体系结构年度国际研讨会（MICRO），IEEE，2016 年，第 1–13 页。