现代独立GPU支持统一内存空间和实时请求取页。这使得CPU内存和GPU内存之间的数据移动可由驱动和硬件自动管理，因此，软件开发人员的工作量大大减少。然而，当应用程序的工作数据集超过GPU内存的物理容量时，由此产生的数据移动开销可能会导致严重的性能损失。

本文提出了一种提升内存超额配置情况下程序性能的ETC内存管理框架，该框架可以对应用程序不感知地掩藏逐出请求延迟，缓解内存抖动问题并可以提高有效的内存空间。 ETC主要包括三项技术：1. 主动逐出页技术（E）：通过主动逐出作废的页，提前为新写入的数据创建空间，可以隐藏页逐出延迟。2. 内存感知的并行性控制（T）：当页缺失频率变高时，可以动态地降低GPU并行性来减少缺页中断，缓解内存抖动现象；3. 内存压缩技术（C）：可以在不增加实际物理内存容量的情况下存放更大的工作集。

我们发现没有一种单一的技术适合所有应用程序，因此，ETC将主动逐出页技术，内存感知并行度控制技术和内存压缩技术集成到ETC框架中，该框架对程序员不感知地动态选择最有效的技术组合。ETC将应用程序分为三类：内核之间没有数据共享的常规应用程序，跨内核的数据共享的常规应用程序以及不规则的应用程序。我们的评估表明，ETC完全缓解了没有数据共享的常规应用程序的超额预订开销，并提供了与理想的无限GPU内存基线类似的性能。我们还表明，对于具有数据共享和不规则的常规应用，ETC优于最先进的基线60.4％和270％应用程序

引言：

随着面向高性能计算的应用程序对密集型计算需求和程序员对高可编程性需求的不断增加，如今图像处理器已经成为高性能计算领域首选的计算平台。然而，想要使得应用程序发挥出最大的性能，依然要求程序员手动调整代码来适应GPU的系统结构和内存容量的要求。而随着GPGPU应用程序的工作数据集不断增大，有限地GPU内存容量已经成为影响硬件设计和程序性能的严重瓶颈。

如今内存虚拟化支持不断演化，GPGPU程序可以更加容易地扩展工作数据集，使之超出GPU内存物理容量的限制。现代GPU配备了统一内存空间和实时请求取页功能，这些新的特性使得开发者不再需要手动地管理数据在CPU内存和GPU内存之间移动。但是当一个GPU应用程序的工作数据集超出了内存的物理空间时，即内存超额配置时，旧数据必须被逐出，为新数据腾出空间。我们在一个真实GPU系统的测试显示，当内存超额配置时，真实GPGPU应用程序会遭受巨大的性能损失，有时甚至会发生宕机。

部分内存超额配置引发的性能损失可以通过程序员的软件优化来缓解。例如，程序员可以将只读数据从CPU内存端复制到GPU内存，当需要更多空间存放新数据，这些只读数据不再需要使用时，可以直接丢弃这些只读数据，而无需逐出写回到CPU的内存；程序员可以将预取请求和逐出请求重叠起来，新请求的数据无需等待内存逐出操作，有效地减少了延迟。但是，采用软件优化方法来降低内存超额配置引发的性能损失有几个明显缺陷。首先，这要求程序员能够直接分清数据是否为只读数据；其次，程序员必须理解并利用好成千上万条线程的局部性原理，直接将不同地数据页映射到CPU或GPU的内存。最后，程序员需要手动地在CPU和GPU内存之间搬运数据。这些缺陷在云计算环境里会变得更加明显，因为共享一个GPU的虚拟机用户无法知道其他用户的应用程序的工作数据集大小，难以对内存超额配置情况进行软件优化。因此，设计一种能够降低内存超额配置开销并且对程序员不感知的机制的需求非常迫切。

我们观察到利用当前GPGPU应用程序的两个关键特性可以更好地管理好超额配置的内存。首先，由于内存超额配置带来的性能损失大小取决于不同应用程序的不同访存特性。基于GPU内存的页访问是否具有可预测性，我们大致将应用程序分为规则应用程序和不规则应用程序；第二，内存超额配置开销的来源与程序类型相关。内存抖动，即重复地在CPU和GPU内存之间进行页来回数据迁移，主导了不规则程序的内存超额配置开销的原因。而较高的逐出页延迟则是规则程序的内存超额配置开销的主要来源。我们还发现同一个应用程序不同的GPU内核函数若存在数据共享，则会进一步影响GPU内存超额配置的性能。

基于这些特性，我们提出了一种叫ETC的内存超额配置管理框架。该框架以一种对程序员不感知地方式来降低内存超额配置开销。ETC首先高效并自动地将应用程序划分为数据不共享的规则应用程序；数据共享的规则应用程序以及不规则的应用程序；ETC之后会基于应用程序的类型选择最有效的策略组合来减缓内存超额配置开销。ETC集成了多个模块，它们协同工作可以有效掩藏或减少内存超额配置带来的开销。ETC主要包括（1）一个应用划分器，能够通过测量的访存合并系数来检测并判断应用程序的类型；（2）一种策略引擎，能够基于应用程序类型选择并采用最合适的策略；（3）一种主动页逐出技术，针对规则应用程序，主动提前逐出无用页，为之后要请求的页腾出物理空间；（4）一种内存感知的并行控制策略，针对不规则应用程序，通过降低线程级并行来减少有效在线工作数据集；（5）一种主存压缩引擎，能够为应用程序有效提高可用存储空间。

我们的ETC是一种软硬件协同机制。我们实验评估了来自多个基准测试集的15个应用程序。性能测试结果显示，ETC在降低内存超额配置开销上非常有效。对于数据不共享的规则应用程序，ETC几乎能够消除内存超额配置带来的性能损失，使之与无内存超额配置的应用程序性能相当。对于数据共享的规则应用程序和不规则应用程序，ETC相对于当前技术，性能分别提高了60.4%和270%。

本章主要包括以下贡献：

据我们所知，这是第一篇对GPU内存超额配置下的性能开销做了深度分析，并且基于不同的程序类型分析出产生GPU内存超额配置下的性能损失的原因。

我们提出了一种软硬件结合的解决方案，显著降低内存超额配置下的性能损失。我们的方法对程序员不感知，不要求任何代码的修改。

我们开发了三种内存超额配置优化策略。我们发现并没有任何一种单一方法能够适用于所有类型的应用程序。从这个角度出发，ETC根据访存特性划分应用程序的类别，并为不同类别的程序采用不同的策略组合来优化性能。

背景

本节主要介绍了内存超额配置的背景和应用程序不感知的优化方法的实现动机。2.1节主要介绍了GPU工作模型和统一内存的背景。2.2节主要分析了真实gpu系统中的内存超额配置开销。2.3节描述了先前避免内存超额配置的方法及为何驱动了我们设计新的应用程序不感知的框架。

GPU工作模型

GPU通过单指令多线程模型实现了超高的吞吐率。在每个时钟周期，一个GPU核（有时又被成为SM）执行一组线程，该组线程一般被称为warp或wavefront。一个warp中所有的线程以整体来执行。GPU能够通过细粒度多线程掩藏超长的访问延迟，即在每一个时钟周期，新的warp指令会被取出并执行，因此并没有来自同一个warp的两个指令会同时出现在流水线中。当没有一个有效的warp能被执行时，这个GPU核将暂停。每个执行的线程可以访问独立的存储地址，因此GPU会潜在地产生大量同时发生的存储访问请求。

统一虚拟地址空间。现代GPU支持CPU和GPU的统一虚拟地址空间。这允许CPU可以通过GPU程序的指针访问并管理GPU物理内存中的数据。这一功能极大地提升了GPU的可编程性，因为只需要一套虚拟地址映射，开发者可以便捷地管理CPU和GPU的存储空间。

统一内存。出现统一虚拟地址空间后，开发者仍然需要通过编程在GPU内存分配空间，并在GPU核函数开始执行之前，从CPU内存拷贝数据到GPU内存。GPU统一内存技术则将CPU内存和GPU内存看作一个整体，所有的CPU程序访问或GPU核函数的访问都可以抽象。因此，当GPU核函数开始执行时，内存可以自动地按照需要移动数据，需要访问某一块数据时，则产生页缺失，从CPU内存直接调到GPU内存。

GPU内存超额配置

虽然统一虚拟内存可以大大地提高可编程性，然而这并不是万精油。首先，虚实地址转换的硬件结构会引入性能开销，降低GPU的吞吐率。第二，按页表从CPU内存往GPU内存取数据要求频繁地高延迟数据搬移。

许多先前的工作都以提升GPU虚实地址转换的性能为目标。（如并行地查页表，更高的TLB映射率以及更低的查页表延迟）。 然而这些方法并没有尝试去直接解决实时按需取页的高开销问题。之前的工作发现预取在掩盖开销方面具有一定的价值，但并没有考虑如何优化内存超额配置时的性能。

图1显示了内存超额配置带来的性能损失。我们观察了了来自CUDA SDK和Polybench测试集的五个不同的GPGPU应用程序。这些应用程序在一个拥有2GB可用内存的NVIDIA GTX 1060 GPU。为引入内存超额配置，我们手动地修改可用内存，使之分别是每个GPU内核函数所需内存容量的大约50%和大约75%。从图中结果我们有三点观察。第一，所有的应用程序在内存超额配置的情况下均出现了显著地性能下降：内存超额配置率越高，则性能下降约大。第二，2DCONV，3DCONV和RED三个应用程序性能平均下降了17%。我们发现这个性能损失是由于GPU在等待逐出GPU物理页，为新需要的数据页腾出空间。这也是规则应用程序的内存超额配置主要开销来源。第三，当有效内存为ATAX和MVT的内核函数所需存储空间的75%时，两个应用程序的性能下降超过了1000倍，而当GPU的有效内存仅能满足应用程序所需内存的50%时，会出现内存抖动的现象，使得数据页不断在CPU和GPU内存之间来回搬运，该现象产生过多缺页中断直接导致系统宕机。

一种应用程序不感知的框架

先前避免内存超额配置的方法。有许多技术都可以用来管理内存超额配置。例如增加存储容量是一种避免内存超额配置的最有效方法。例如封装内的三维堆叠存储器HBM和HMC已经被广泛应用到了NVIDIA P100和V100 GPU, AMD Radeon R9系列GPU以及谷歌的TPUv2上。然而，增加封装内的三维堆叠存储器的存储容量依然面临着三大主要挑战。首先，堆叠的层数受到了半导体制造技术的限制。其次，在硅中介层水平方向堆叠更多的存储器受到硅中介层引线复杂度和处理器引脚数目的限制。第三，随着GPU应用程序数据集持续增长，应用程序开发者仍然需要考虑如何使用有限的存储容量。除此之外，将任务划分分配到多个GPU，或者将任务划分成更小的粒度，使得每个任务所需要的存储空间小一点也是可以采用的方法。但是，这需要程序员花费很大的精力将一个复杂的大GPU任务分解，对程序员的要求非常高。将任务分配到多GPU系统也将引入更多的GPU之间的通信，以及CPU和GPU之间的通信，这也将带来额外的开销。

原始设计。为了减少内存超额配置开销，我们尝试了多种不同的方案策略进行设计空间探索。这些策略的主要目标是先降低缺页中断开销。我们测试了不同的warp调度策略。给定发出缺页中断请求的warp和没发出缺页中断请求的warp不同的优先级。非缺页中断请求的warp将拥有较高优先级，因此该warp已在内存中的数据能被继续处理。但是，我们发现这种warp调度策略并不能减少缺页中断，只是将缺页中断的分布打散。最终，所有的线程都会在等待缺页中断处理的完成而停滞。由此我们给定结论，虽然warp级别的调度策略是有效掩藏访存延迟的方法，但这还是难以掩藏缺页中断处理延迟，因为缺页中断处理的延迟比访存延迟长几个数量级。

我们也试验了集中不同的页替换策略来强化局部性原理并将内存抖动最小化。传统想法认为理想的LRU策略是页替换策略可以达到的性能上限，但是LRU策略在GPU内存上实现太过昂贵。基于时序的LRU策略只需要一个列表存储每一个页从CPU内存移入GPU内存的时间，相对易于管理维护。我们的实验显示，基于时序的LRU策略对于流访问模式的应用程序效果非常显著，这种访问模式的应用程序即为我们定义的具有强序列局部性的规则应用程序。然而对于不规则应用程序，也就是随机访问模式的应用程序，基于时序的LRU策略并不奏效。在这种应用程序中，我们观察到内存超额配置会引起严重的内存抖动。因此，没有任何一种页替换策略能够有效减少内存抖动。

目标。我们的目标主要有三层，首先，我们的设计目标是尽最大可能恢复内存超额配置下的程序性能到应用程序拥有足够的可用内存时的水平。其次，我们设计的框架必须对程序员不感知，因为我们不希望程序员手动管理控制物理内存。第三，我们的设计必须能够满足不同类型不同特点的应用程序的需求。

3.GPGPU应用程序的访存特性研究

设计一个有效的内存管理框架要求对应用程序访存特性有深入的分析理解，也就是应用程序的特性。从这个角度出发，我们首先分析检测各种不同应用程序的访存trace，抽取其中最具有代表性的访问模式。我们发现，应用程序一般可以被划分为规则访问模式和不规则访问模式。图2（a）-(b)显示了两个代表性应用程序（3DCONV和atax）.在线程块与页访问的关系中，3DCONV呈现出典型的流访问模式，而ATAX则呈现出一种相对随机的访问模式。在任一时间点，3DCONV只访问非常少数量的页。如图3所示，几乎所有的线程块在一个时间点都在访问少量的几块页面。相反地，无论任一时间点，ATAX都在同时访问大量的页面。如图3b所示，ATAX的不同线程块访问的是不同的页面。我们还发现有许多应用程序都表现出和3DCONV类似的内存访问特征。这一类应用程序具有相当较小的在线工作数据集，也就是说在一段时间内访问的数据量相对较小。相反地，像ATAX这一类不规则应用程序的在线工作数据集相对较大，因为每个线程都有可能访问一个新的不同的页面。这将导致在给定的时间内大量不同的页面被访问。此外，我们发现规则应用程序的访存是可以被预测的，如流访问模式，而不规则应用程序的访问模式是无法被预测的。

我们观察到对于规则应用程序，流访问模式的可预测性表现在被逐出的页通常都不会在将来被重新请求，这很自然地避免了内存抖动现象。但是，对于非规则的应用程序，预测访问模式非常困难，一旦在线工作数据集超出了有效的内存容量，任何被逐出的页都有可能短时间内再次被请求，导致内存抖动的发生。

GPU内核函数间数据共享。 另一种可能的情况是某些规则应用程序，不同的GPU内核函数将访问处理同一块数据。如图4所示，在LUD中，每个内核函数的访存模式均为流访问模式且访问较小的数据集。但是，当一个应用程序的多个内核函数共享一块数据时，这块数据将被重复的访问。当这块数据大于物理内存的容量时，这块数据将多次在CPU和GPU之间来回搬运，导致较低的性能。

基于以上发现，我们有三点结论。首先，逐出开销是规则应用程序内存超额配置的最主要来源。第二，内存抖动主导了非规则应用程序的内存超额配置的性能损失。第三，数据共享引入了额外的数据迁移，导致更低的性能。这三点结论将指引我们框架的设计。

ETC框架

ETC的关键准则是为不同类型的应用程序采用不同的内存管理技术。应用程序主要有三类：无数据共享的规则应用程序，数据贡献的规则应用程序以及不规则应用程序。基于这三类应用程序，ETC主要包含四个主要技术：应用划分技术，主动逐出技术，内存感知的并行控制技术，以及内存容量压缩技术。

一旦检测到内存超额配置，ETC首先对应用程序类别进行判定。根据程序的类型，以及数据是否在多个内核函数间共享，ETC选择主动逐出，内存感知的并行性控制以及内存容量压缩技术中的一种或多种进行性能优化，对于规则应用程序，ETC会采用主动逐出技术，对于无数据共享的规则应用程序，ETC在采用主动逐出技术的同时还会采用内存容量压缩技术。对于非规则应用程序，ETC会采用减少一部分运行的SM来控制并行性，同时采用内存容量压缩技术来优化性能。

4.1 应用程序划分

在ETC能够选择为哪种应用程序采用哪一种技术之前，它首先检测应用程序的类型以及是否有不同内核函数之间的数据共享。为检测每个SM上运行的应用程序的类型，ETC采用访存合并系数作为判断指标，这也是在GPU等SIMT体系结构的系统中被广泛使用来剖析应用程序。当来自于同一个warp的方寸请求访问同一条缓存行时，访存合并单元会合并请求，避免冗余访问，同时也减少了带宽消耗。对于规则应用程序，由于其访存局部性非常高，访存合并率非常高。但是对于不规则应用程序，由于非常差的局部性，更多的呈现出来的低合并率。基于这些观察，ETC在每个SM的存取单元采用了一个计数器来取样合并后的访存数。如果合并后的访存数低于一定的阈值，则被仍未该SM上运行的是规则应用程序，否则则将该应用划分为非规则应用程序。

为检测多个内核函数间的数据共享，ETC依赖于编译技术。当编译器检测到来自多个内核函数的访存指针指向接近的数据地址，则表面该应用程序的多个内核函数存在数据共享。

4.2主动逐出技术

主动逐出的关键想法是预先地在GPU用完所有的物理内存之前就逐出页。这将允许页逐出产生的数据迁移与缺页中断产生的数据迁移同时发生。图5给出了主动逐出技术如何工作的例子。当页面在GPU内存缺失时，失败的虚实地址转换会产生缺页中断，MMU则会向CPU内存取新的页如图5（a）。如果应用程序用尽了物理内存，对新数据的访问需等到其他GPU内存中的页被逐出回CPU内存后才能执行。在目前的产品系统中，逐出只能被缺页中断触发。从CPU到GPU的内存迁移只能在GPU到CPU的页逐出内存迁移完成之后才能开始，如图5a所示。如图5b所示，我们观察到可以通过重叠页逐出和缺页中断处理来掩盖逐出开销，实现降低内存超额配置开销的目的。

当前GPU支持双DMA引擎，这使得缺页中断和页逐出产生的数据迁移理论上能够同时发生。应用程序开发者可以通过手动并行预取和逐出操作来优化应用程序。但这仍旧是给了程序员巨大的负担，并且这也违背了在线按需取页的关键目标：减轻程序员负担。为了自动的并行页逐出和缺页中断的数据迁移，我们修改GPU的驱动来迫使应用程序消耗完GPU所有物理内存之前就逐出页。这样允许缺页中断的处理和页逐出的处理同时发生。

但是如何在最合适的时机进行主动页逐出是重要的设计挑战。第一，如果太早从GPU内存中逐出页会导致被逐出的页还在使用中，很可能会被迅速再次请求。另一方面，如果太晚逐出页很可能导致主动逐出失效。第二，GPU驱动需要确定一次有多少页需要被逐出。主动逐出过多的页可能可以减少GPU内存中写不再使用的页，腾出更多的空间给新数据页。但是，如果主动逐出过多的页会导致仍在使用的页被逐出，造成额外的开销。我们开发出一种机制来在这之中达到一种平衡。

避免过早逐出。为确定主动逐出的正确时机，我们剖析在NVIDIA GTX 1060 GPU上运行的几个不同的GPGPU应用程序，观察每个应用的内存访问轨迹，即数据页从CPU迁移到GPU的量是如何增长的。图6显示了五个GPGPU应用程序从CPU内存移动数据页到GPU内存的轨迹图。从数据中共我们得出四点观察。首先，内存占用是随着时间线性增长的。其次，有可能出现多个增长的阶段（如ATAX的内存占用（图6中的蓝色点线）），但每个阶段内存占用增长的趋势仍然会线性的。第三，GPU的SIMT工作模型显示，不同的warp执行相同的指令，但访问不同的数据。当所有这些warp并行地执行，共享全局内存带宽，则内存占用会增加直到这个阶段所有的数据被取回。这也能解释内存占用的增长为什么是线性的。第四，缺页中断之间的时间间隔是相对恒定的。基于这些观察，GPU可以预测当一个缺页中断发生时，一系列的缺页中断在短时间内相继稳定发生。我们也可以触发多个页逐出请求，来为新需要的数据页腾出更多的空间。

避免过晚逐出。我们发现CPU到GPU的数据传输速度并不总是和GPU到CPU的数据传输速度相同。通过在nvidia gtx 1060 GPU的测试，我们发现从GPU向CPU的传输速度要显著快于CPU到GPU的传输速度。因此，传输同样数量的页，从GPU传输回CPU的速度相对于从CPU传输到GPU速度更快。基于这些发现，我们可以在发生缺页中断的同时开始页逐出避免过晚地逐出页。

不规则应用程序在短时间内访问大量的页面。因此，主动逐出技术将变得非常低效。我们发现潜在地缺点是主动逐出技术很可能会加重内存抖动的发生。

实现。为了实现主动逐出，ETC修改GPU runtime中的虚拟内存管理单元，来包含主动逐出单元（PEU）。当缺页中断发生时，PEU中断GPU驱动，使之移动缺失的页到GPU内存。当GPU驱动可成功分配一块新页到GPU内存时，PEU首先开始检查应用程序划分逻辑，然后，PEU检查内存分配空间大小，并与有效内存容量比较来预测是否内存超额配置。主动逐出单元在以下情况发生时会触发：1.内存超额配置 2.有效内存小于一定的阈值（经验设置小于2MB）。

4.3内存感知的并行度控制

如2.2节讨论的，页面层次的内存抖动可以大大降低非规则GPGPU应用程序的性能。如图2b和3b所示，通常情况下，非规则应用程序的一个页仅被少量的线程块访问。当GPU许多非规则应用程序的线程块同时执行时，在线工作数据集快速增长，产生严重的内存抖动。这是传统页替换策略无法解决的。为了避免内存抖动，我们的想法是限制同时访问内存取页的数量。从这个角度出发，我们采用了内存感知的并行度控制技术，其目标是通过限制并行的线程数量，来减少非规则应用程序的有效在线工作数据集。GPU并行度控制技术可以有两种实现方式，第一是线程块并行度控制，第二是SM并行度控制。线程块并行度控制降低每个SM内部的线程块的并行度。而SM并行度控制则降低每个GPU中SM的并行度。我们实验了这两种方法并发现线程块并行度控制方法引入了相对过长的时间来达到最少的内存抖动。与线程块并行度控制相比，SM并行度控制相对更快限制并行度使之达到最合适的在线工作数据集大小。因此，ETC采用了SM并行度控制的方法来减少GPU中的内存抖动现象。

实现。当非规则应用程序被检测到且发现内存时超额配置的，ETC则会触发我们基于周期的SM并行度控制技术。当并行度控制被触发，ETC首先通过暂停一半SM的取指功能，已经在流水线中的指令将继续执行直到完成。在这个初始阶段，ETC很可能停掉了过多的SM，导致较低的利用率；另一种情况是停掉的SM数量太少，仍然导致内存抖动。因此在初始阶段之后，ETC需要基于观察到的内存利用率自适应地动态调整可执行的SM的数量。如图7所示，内存感知的并行度控制将GPU的执行划分为两个阶段：检测阶段和执行阶段。在检测阶段，ETC首先检测是否出现了逐出请求或缺页中断请求来判断初始阶段ETC的SM并行度控制是否过度。如果一个缺页中断请求被检测到，或者检测阶段的运行时间结束，检测阶段将停止。一旦检测阶段停止，执行阶段开始，内存感知的并行性控制技术完成自适应地调整可运行的SM数量，不再调整，继续执行直到本阶段结束。

如果检测阶段的结束是因为检测阶段的时间完成，并没有发生缺页中断。这显示当前的有效物理容量是足够的。GPU还能够同时执行更多的线程，以更加高的内存利用率运行而无内存抖动。在这种情况下，ETC增加一个可运行的SM.完成这一操作，ETC使上一个被停止取指的SM重新开始取指来实现SM开启。

如果检测阶段的结束是因为一个缺页中断且无页逐出发生，这说明GPU依然有足够的有效内存空间。

如果检测阶段的结束时因为一个缺页中断，且发生至少一次页逐出，这说明GPU的内存容量已无法满足应用程序的有效工作数据集。ETC需要限制更多SM的执行，以减少有效工作数据集，使得内存容量足以容纳现有的工作数据集。在这种情况，只要缺页中断处理完成，ETC将停止产生该缺页中断的SM的取指功能，终止其运行。因为这个SM最有可能是刚开始执行处理数据。

在每次调整之后，GPU会不受打扰地执行一段时间。当所有可运行的SM执行完执行阶段的时间后，GPU重新回到检测阶段，再次检测并调整可运行SM的数量。

通过内存感知的并行性控制方法，非规则应用程序的并行度被调整到有效工作数据集大小能完整地在内存中运行。虽然并行度控制技术在一定程度上降低了线程级的并行度，我们发现它能大大减少CPU内存和GPU内存之间的数据搬运，同时大大恢复因为内存超额配置造成的性能下降。同时，这种并行度下降产生的性能损失，也可以通过结合容量压缩方法来挽回。这将在4.4节中具体介绍。

4.4 容量压缩

通过之前的介绍可以知道，在内存超额配置的情况下，主动页逐出技术可以提高规则应用程序的性能，而内存感知的并行度控制技术可以提升非规则应用程序的性能，但依然可能出现这两种应用程序之外的情况，即以上两种方法并不足以独立提升足够的性能。首先，ETC的主动逐出技术可以掩藏页逐出延迟，但是在多个内核函数数据共享的情况下，不能减少页搬运的次数。第二，ETC的内存感知的并行度控制技术虽然对避免非规则应用程序的内存抖动情况非常有效，但它会带来线程并行度下降的问题。

为了进一步降低内存超额配置的开销，我们的目标是提高主存的有效容量。从这个角度出发，我们开发了一种内存压缩技术。ETC内存压缩技术的和新想法是，选择性地使用内存压缩来提升性能。以前已经有好几种主存压缩技术被移除， 他们都能有效提高内存容量。在本章中，我们采用了线性压缩页技术（LCP）来压缩主存中的数据。

LCP是一种低延迟的主存压缩框架，之前在CPU的应用已显示其在提升主存容量方面非常有效。我们发现LCP在GPU中性能会有严重影响，因为它需要额外的一次访存来获取存储在主存中的压缩相关元数据。如图8所示，额外的一次LCP元数据访问可以导致额外的带宽需求，实验显示在无内存超额配置的情况下，GPU的性能下降了13%。

因此，对于ETC来说确定何时采用LCP压缩技术非常重要。内存压缩技术主要应用在两种类型的应用程序中，包括数据共享的规则应用程序，和不规则应用程序。因为来自这两种类别的应用程序的线程块会访问大量的数据，而内存压缩技术允许更多的数据存储在主存中。更重要的是，有了内存压缩技术，内存感知的并行度控制能够一定程度上提升并行度，这样比较单独并行度控制，更多的线程可以同时执行。

实现。由于当前的GPU已经在内存控制器和PCIe上应用了带宽压缩技术，内存控制器和DMA单元已经配置了压缩和解压缩硬件模块。为实现LCP，ETC采用了一个具有512个存储单元的缓存在内存控制器中以加速压缩元数据的查询，页能够降低LCP的性能开销。一旦应用程序划分单元确定当前执行的应用程序是数据共享的规则应用程序或者是非规则应用程序，且内存处于超额配置状态，ETC开始进行容量压缩操作。所有写入到GPU内存的数据将采用BDI压缩算法进行压缩，并通过LCP框架存入内存，非常简单高效。

4.5 ETC设计总结

图9是ETC的顶层设计方案，主要包括应用划分单元，主动页逐出单元，内存感知的并行度控制单元，以及主存压缩单元。

当待分配的总数据量大于GPU的物理内存时，ETC将被激活，应用程序划分器将开始根据GPU的硬件信息和编译信息开始识别应用程序类型。一旦应用程序被识别为规则应用程序，ETC将在GPU驱动的虚存管理单元开启主动页逐出功能。如果应用程序有数据共享，则同时开启内存压缩技术。如果应用程序被检测为不规则应用程序，ETC将同时开启内存感知的并行度控制技术和内存压缩技术，避免内存抖动现象的同时提升有效内存容量。

实验方法学

我们修改了基于GPGPU-sim 3.2.2的Mosaic模拟器来评估ETC。GPU核心和存储系统的实验配置如表1所示。

在线按需取页和内存超额配置。我们详尽地模拟了在CPU内存和GPU内存中按需取页搬运数据的过程，符合CUDA8.0的描述。当一个内核函数第一次要访问一块页时，TLB缺失会触发页表查询。如果该页并不在GPU内存中时，也表查询失败，产生一个缺失页。IOMMU会中断CPU来处理这个缺页错误。我们模拟了大约20us的缺页中断处理延迟，并应用了最新的硬件页预取器来降低缺页中断开销。当GPU内存完全被用满之后，GPU驱动会通过基于时序的LRU页替换策略逐出旧的页以足够的空间拷贝新的页。我们在实验中为每个应用程序配置完全足够的内存空间，应用程序所需内存的75%和50%。

应用程序。我们随机广泛地从CUDA SDK，Rodinia，Parboil和Polybench选择了15个应用程序。我们将应用程序划分为无数据共享的规则应用程序（2DCONV，3DCONV，SAD，CORR，COVAR，FDTD和LPS），数据共享的规则应用程序（LUD，SRAD，CONS和SCAN）以及非规则应用程序(ATAX,BICG,GESUMMV和MVT)。这些应用所需的内存占用从7.28MB到22.5MB不等，平均值为22.5MB。我们没有模拟更大的内存占用是因为模拟过大的内存占用会导致不现实的模拟延迟。

设计参数。ETC采用了不同的设计参数。我们设置内存合并参数阈值为10个缓存行来划分规则应用程序和非规则应用程序。我们设置2MB作为剩余GPU内存空间的阈值来触发主动页逐出技术。我们设置并行度下降和并行度上升的程度为一次一个SM，我们实验发现这个值能取得最高的性能。

6. 实验评估

我们评估ETC,并将其与1）当代采用页预取的真实情况（BL）以及2）一个有无限内存的理想情况作比较。

6.1 性能

图10显示了对于不同类别应用程序以无限内存的情况为性能标准的实验结果。我们做出了三点结论。首先，ETC在降低内存超额配置开销上作用显著。对于无数据共享的规则应用程序，其性能接近无限内存的情况，这是因为通过我们的主动页逐出技术，这一类应用程序的最主要开销来源，页逐出延迟可以被完全掩盖。第二，我们发现数据共享的规则应用程序由于不同内核函数之间的同步，导致的页面数据移动不能完全避免。但是ETC相比于当前最新技术依然提升了60.4%的性能。第三，对于非规则的应用程序，ETC相比当前最新的技术BL依然提升了2.7倍。我们给出结论，ETC框架对于内存超额配置非常有效。

6.2 技术和应用程序细节分析

我们提供了关于每种ETC采用的技术对于每种类型应用程序影响的深度分析。

无数据共享的规则应用程序。图11显示了主动页逐出技术，容量压缩技术对于无数据共享的规则应用程序的影响。我们有三点观察发现，第一，当主动页逐出技术被触发，页逐出延迟几乎可以完全被掩藏。从实验中我们可以发现，对于无数据共享的规则应用程序中仅LPS因为逐出过多的页而导致性能离理想情况稍有下降。

第二，由于页面限制图11并没有完全展现，无数据共享的规则应用程序并不能从SM并行度控制技术对这一类应用并无效果。因为SM并行度控制不能掩藏页逐出延迟。事实上，SM并行性控制技术使得线程级并行下降的同时，使得延迟掩藏能力也下降。

第三，无数据共享的规则应用程序在采用了内存压缩技术后，性能比BL更差。这是应为额外的压缩元数据访问所带来的开销，如4.4节介绍的。

数据共享的规则应用程序。图12显示了数据共享的规则因公程序在主动逐出技术，容量压缩技术和二者一起使用时的性能。我们有四点观察，第一，与无线内存的情况相比，当前技术在75%和50%的数据集能被装进物理内存的情况下性能分别下降了52.2%和74.1%。第二，仅采用主动页逐出技术，性能相比于当前技术仅提升了9.3%。这是因为由于数据共享导致的数据移动在这种情况下主导了内存超额配置的开销。第三，仅使用内存压缩技术可以获得52.8%的性能提升。这是因为它增加了有效内存容量。第四，当主动页逐出技术和内存压缩技术同时使用时，可以获得平均60.4%的性能提升。

我们得出以下结论，ETC提升了规则应用程序的性能，无论数据是否在多个内核函数之间共享。

不规则应用程序。图13和图14显示了每个不规则应用程序在ETC各个技术下的性能和总的页逐出数。

为评估ETC的并行度控制策略（MT），我们将它和一个简单的并行度控制策略进行比较，该策略静态地在开始执行时减少一半的SM数量（如图13和14所示）。我们有三点观察，首先，简单的并行度控制方法在内存能容纳75%的数据集的时候性能提升57.7%。第二，当内存仅能容纳50%的数据集的时候，简单的并行度控制方法变得不那么有效，降低了10.5%的性能。相反地，我们的内存感知策略，可以动态地调整可运行的SM的数量，相比于当前技术，提升了436%的性能。第三，我们的自适应调整并行度策略在bicg和GESUMMV两个应用程序，且内存仅能容纳75%的数据集的情况下性能不及简单的并行度控制方法，这是因为自适应调整需要一段时间才能达到最佳并行级别，而简单的并行度控制方法在这种情况恰好达到了。

图15显示了1000万个时钟周期里非规则应用程序ATAX的缺页中断率。当内存超额配置且75%的数据集能存放于内存中，会发生内存抖动，且频繁地发生缺页中断。相反地，当内存感知的并行度控制方法被激活时，缺页中断并不频繁。这说明我们的内存感知的并行度控制犯法在降低在线有效工作数据集的大小非常有效。同时，图13和14的数据页表明，内存感知的并行度控制方法可以减少页逐出的发生。

内存压缩策略对于非规则应用程序是否有效取决于压缩率和GPU相对于应用程序的内存占用的比例。当一个应用程序完整的内存占用能够在压缩之后被物理内存满足，缺页中断将 不再发生。图13中BICG和MVT大大减少的页逐出数说明这两个应用程序的压缩率足够高，使得内存能够完全满足应用程序的内存占用。相比于当前最新技术，内存压缩技术分别提升了51.8%和203.6%。相比于无限内存的情况，BICG和MVT分别恢复了85.7%和88.7%的性能。图14显示了GPU的内存仅能满足50%的数据集的性能情况。即使使用了内存压缩技术,所有的应用程序都出现了内存抖动。我们的内存感知的并行度控制技术在和内存压缩技术同时使用时，相比于当前技术性能提升了436%。我们得出以下结论，采用内存压缩技术有助于提升内存超额配置下非规则应用程序的性能，然而缺页中断和内存抖动仍然限制应用程序的性能。因此，需要内存压缩技术和内存感知的并行度控制一起工作以在内存超额配置的情况下达到较好的性能。

我们观察到主动页逐出技术相比于被动页逐出技术性能下降了29.7%。这是因为数据页被过早地逐出GPU物理内存。因此，主动页逐出技术并不适合非规则应用程序。

总的来说，内存感知的并行度控制技术和内存压缩技术对于提升非规则应用程序的性能非常有效。因此，我们的ETC框架采用这两种技术。如图13和图14所示，采用了这两种技术的ETC能够提升270%的非规则应用程序的性能。虽然并行度控制技术在一定程度上降低了线程级并行，但它能够有效降低内存超额配置的开销以及缓解内存抖动。

6.3 应用程序类别划分精确度分析

如6.2节所讨论的，ETC依赖于正确的应用程序类别划分，只有确定正确的应用程序类别才能选择最合适的策略（见4.1节）。图16比较了从5万个时钟周期取样的平均访存合并系数与每个应用的真实平均访存合并系数。我们可以观察到规则应用程序和非规则应用程序的访存合并系数相差非常大。我们发现，将访存合并系数阈值设置在5到10之间都能使得ETC的应用程序类别划分准确读达到100%。因此我们将阈值设置为10。

图17显示了每个warp指令中32个线程访问的平均页数。我们可以发现非规则应用程序的warp一般会访问多个页，而几乎大多数的规则应用程序一般只访问一个页。因此，采用页面级别的访存合并系数与缓存行级别的访存合并系数来判定不同的应用程序类别均有非常高的准确率。

6.4敏感度分析

在这一节，我们测试了不同敏感度下ETC的性能。主要包括内存感知的并行度控制策略调节度大小，缺页中断处理延迟大小和内存大小对GPU内存超额配置下的性能的影响。

SM并行度控制度。每个阶段并行度提高或降低所变化的SM的数量称为SM并行度控制的度。它能够直接影响应用程序的性能。图18显示了每个阶段调节不同的SM数（每次增加或减少的SM数量）的相对性能。基于图18，我们得出两点观察。第一，我们的内存感知的并行度控制策略在并行度下降和上升度均为1时得到最高的性能，说明细粒度的调整效果最好。第二，我们观察到ETC的性能相比于并行度上升，对于并行度下降更加敏感，因为缺页中断产生的开销相比于线程级并行的下降产生的开销更大。

缺页中断延迟。图19显示了GPGPU的应用程序在不同的缺页中断处理延迟下的性能。我们的实验设置缺页中断延迟从20us到50us不等。实验结果显示的是相对于缺页中断延迟为20us的性能。我们观察到当缺页中断延迟从20us提高到50us，平均性能下降了31.2%。这些数据说明隐藏页逐出开销对于恢复内存超额配置下的性能非常重要，因为缺页中断延迟是一个主要的性能瓶颈。

压缩率。每个应用程序的内存占用压缩影响到GPU主存能够容纳的页的数目。图20显示了所有应用程序采用不同的合成压缩率下的性能，相对于每个没有采用压缩技术的应用程序的性能。该实验在物理内存仅能容纳下50%的内存占用的内存超额配置下进行。实验结果显示GPU性能随着内存压缩率的增加而线性提高。当所有的内存占用能够被GPU内存容纳时，性能可以大幅度提升（压缩率为2）。

6.5硬件开销

我们分析为支持ETC的每个部分所需要的硬件开销。主动页逐出技术不需要任何硬件开销，可以直接在GPU驱动中实现。我们修改了驱动来检测可用的内存大小以触发主动页逐出技术。为实现内存感知的并行度控制技术，IOMMU必须扩展以支持我们的设计。增加两个32位的计数器来统计每个阶段的时钟周期。暂停取指功能需要增加控制逻辑。为实现容量压缩技术，我们增加的硬件开销和为CPU设计的LCP框类似，主要包括一个有512个条目的压缩元数据缓存。内存压缩技术不需要额外的硬件开销，因为当前GPU已经具备了内存压缩功能。压缩与解压缩单元已经存在于存储控制器中。我们扩展页表项，每个页表项增加9位来包含液压锁信息。最后，应用程序划分单元需要：（1）每个存取单元增加一个32位的访存合并计数器；（2）接到取指单元，压缩单元以及IOMMU的信号。

总的来说，我们的设计增加的硬件开销是有限的。除了逻辑开销外，32KB的元数据缓存和482个32位计数器（30个SM中每个SM包含16个计数器，IOMMU有两个计数器），总存储开销小于2KB。

总结

我们介绍了ETC，一种在GPU中能够有效减少内存超额配置下的开销的框架，该框架是对应用程序透明的。当内存超额配置情况下，规则因公程序和不规则应用程序呈现出不同的应用特性。规则应用程序的性能受到页逐出延迟的影响非常大，而非规则应用程序的性能主要受到内存抖动的影响。ETC划分应用程序为规则和非规则的应用程序，我们采用（1）主动页逐出技术来掩藏页逐出延迟，（2）内存感知的并行度控制技术来缓解内存抖动，3）容量压缩技术增加了有效内存容量。对于无内存共享的规则应用程序，ETC消除了内存超额配置带来的性能损失，性能与无内存超额配置下类似。对于数据共享的规则应用程序和非规则应用程序，ETC相比于当前技术，性能分别提升了60.4%和270%。我们做出总结，ETC是一种高效低开销的框架，对于GPU内存超额配置开销最小化具有重要作用