一种采用检查点机制的动态主动GPU抢占机制

介绍

GPU由于其大规模并行处理能力，已经在高性能计算，机器学习和科学计算等领域得到广泛应用。这些方面的计算如今以服务的形式存在于数据中心或云上，而GPU则可作为共享基础硬件资源提供给不同的用户。多任务在GPU中为支持并行服务和任务已经变得必不可少。已经有些主要的硬件特性支持多任务处理，例如Nvidia Kepler体系结构提供的HYPER-Q, AMD支持的命令处理机制。虽然已经有这些机制支持多任务处理，但还需要更多的工作来支持真正的多任务处理机制。

上下文切换是一种在CPU中常用的支持并行性的技术，如今已经被应用到了GPU来支持多任务。CPU的进程相对来说非常轻量化，所以在上下文切换和任务时分应用上非常快速高效。但是，一个CUDA的上下文相比于CPU却是非常巨大的。比如说在NVIDIA GTX980 GPU上，上下文包括每个流多核处理器的256KB的寄存器和96KB的共享内存，对于一个含16个流多核处理器的GPU，其上下文大小达到了5664KB，存储这样大的上下文需要花费大量的存储带宽，并带来严重的性能损失。

之前已经有许多尝试来降低GPU的上下文切换的开销。最早出现的技术是让上下文切换仅在一部分SM上进行，这样其他的SM就能保持执行。被切换的SM回完全停止执行指令，以完成上下文的存取，这些操作对于存储带宽的要求依然非常高。之后出现的方法是，让一部分线程块继续执行直到完成，而只上下文切换一部分线程块。这样可以最大化的利用程序访存和上下文访存的重叠并女性。这个技术被进一步加强扩展，允许在需要抢占时每一个SM内部的不同线程块同时进行执行（直至线程块完成）、丢弃（满足幂等性）、和上下文切换。这些选择均取决于每个线程块对于抢占的终止时间的要求。除了这些工作，一种轻量级的上下文切换技术被设计来降低需要保存在片外存储器的上下文大小。所有这些方法都是通过被动的方法来实现抢占，即只有当抢占请求到来以后才激活所有的操作。因此，如果没有丢弃操作，抢占延迟依然是对性能的一大挑战。

在本章，我们提出了一种动态主动的抢占机制，PEP。这种抢占机制能够大大降低抢占延迟和开销。通过观察kernel从CPU到GPU的发射过程，我们发现kernel的实际执行总是在kernel发射之后。从一个kernel在CPU发射到开始在GPU执行，大约需要几十个毫秒的数量级，我们可以通过预计抢占请求的到达时间来主动准备上下文切换。当抢占的kernel真正到达的时候，需要等待完成的上下文切换工作将变得非常小。因此，需要等待的抢占时间将变得非常短。准备上下文切换的工作我们采用了检查点（checkpoint）的概念。第一个检查点，我们在抢占被预计发生时备份当前的上下文。当真正的抢占请求到达GPU后，仅需再备份变化的上下文。备份变化的上下文相比于完整的kernel的上下文能够节省大量的时间，减少抢占kernel的等待时间。平均来看，总的需要备份的上下文不大于所有的上下文。我们还观察到分配的上下文在线程块的生命周期里并不是完全激活的。所以，我们为寄存器设置脏位，来表明该寄存器是否是激活的。只有有效的寄存器才会被备份，这大大减少了需要存储的上下文大小。此外，我们设计了一个动态实时调度策略来确定抢占方法。小的kernel将要继续执行直到结束，而大的kernel需要采用checkpoint（上下文切换）来进行抢占。这个算法可以达到最小延迟和开销。

我们的贡献主要包括：

1. 我们研究了kernel发射的过程，观察到抢占的事件可以被预测。
2. 我们引入了一种主动的抢占技术来减少抢占的kernel等待上下文切换的时间。采用主动checkpoint技术，当真正的抢占请求到来时，只有一小部分的脏上下文需要被存储。
3. 我们使用了一种相对简单的脏数据存储技术来减少上下文大小，这可以减少不必要的上下文存储。
4. 我们开发了一种相对更加精确的线程块执行时间和上下文切换时间估算方法，设计了实时动态选择算法来确定采用的抢占方法。我们可以完成小kernel和大kernel的抢占，并使之达到最短延迟和最小开销。

我们实验评测了PEP，并与之前最好的抢占工作Chimera在集中不同类型的测试集里比较。我们的实验结果显示，相比之前的工作Chimera，我们可以将平均抢占延迟从8.9us降低到3.6us。我们采用的简单的上下文大小减少技术，将需要存储的上下文从完整的上下文大小减少了16.1%。PEP的总开销，即平均线程块切换延迟相比Chimera减少了6.3%。

背景和动机

在这一节，我们首先简单介绍了GPU的基本结构及其工作模式。我们的基准结构模拟的是一款NVIDIA的GPU体系结构。因此，我们在本章主要使用NVIDIA/CUDA的术语。但是，本章的想法也可以应用到其他厂商的GPU。我们还介绍了checkpointing的方法，该方法在我们的设计中起到了关键作用。

基准结构

1. GPU程序执行：典型的GPU程序包括两个部分的代码：在CPU上运行的主机部分的代码，以及在GPU上运行的设备代码（kernels）。Kernels是以SIMT的模式执行（单条指令，多条线程）。一个kernel的执行意味着无数线程同时在GPU上并行执行。线程会被程序员组合成线程块。

NVIDIA GPU的CUDA编程模型以CUDA C的形式展示给程序员，即从C语言和实时库中扩展而来。图1是一段CUDA程序的例子。一段典型的CUDA C程序的操作序列包括：

1. 声明和分配主机和设备的内存（8-13行）
2. 从主机内存向设备内存迁移数据（14行）
3. 发射kernel。在这个例子里，程序员发射N/256个线程块，每个线程块包括了256条线程（15行）
4. 从设备内存向主机内存迁移数据（16行）
5. 释放内存空间（18-19行）

线程块之间是互相独立的，他们被分别发送到SM上。每个SM上能够并行的线程块数量受限于设备的资源（包括寄存器，共享内存和线程的数量），这个信息在编译时可以知道。大多数之前的抢占策略设计的工作是以线程块的粒度完成的，也会采用可用资源的信息帮助抢占策略的选择。

1. GPU体系结构：图2是GPU基准体系结构，我们本章所描述的GPU体系结构均基于此。当一个GPU程序收到主机CPU执行时发送的操作指令，用户空间实时引擎将API调用指令发送来控制数据操作和kernel的发射。GPU设备驱动发送这些操作指令到流控制管理器的队列里。流控制管理器通过软件队列来管理多条不同的流；每一条流里的指令将被串行执行。一般来说，CPU会先声明并分配存储空间，然后调用cudaMalloc来分配GPU上的全局内存。之后，一个cudaMemcpy（H2D）API的调用将数据从主机内存移动到设备内存。一旦所有的数据被传输完成，流控制管理器可以发射内核，即传输kernel信息（例如维度配置和每个条目的PC地址）到内核函数管理器单元（KMU）。当所有的信息准备完毕，kernel会请求SM资源。如果SM没有足够的资源，kernel需要等待kernel等待池。如果正在等待的kernel的优先级高于正在执行的kernels，则等待kernel有可能需要抢占SM中正在占用资源的线程块，否则，需要等待之前的kernel执行完毕再接着执行。

一旦kernel准备好开始执行，它将被传输到kernel分发单元（KDU）。线程块调度器将相应的线程块分发到不同的SM。每个SM能够处理的线程块的最大数量取决于资源限制，包括可以执行的线程块数量、线程数量、寄存器数量和共享内存空间。在每个SM中的kernel执行过程中，线程块将被分成warps，每个warps包含最多32个线程。每个SM包含一个或多个warp调度器，来选择发出哪一个warp。在nvidia gtx980 gpu体系结构中，每个warp调度器控制32个流处理单元（SP），每个流处理单元处理一个线程。当一个warp因为访存或其他耗时较长的操作而停滞时，调度器会切换执行其他的warp。切换warp没有任何开销，因为所有warps的上下文已经被放入寄存器和共享内存中。因此，GPU通过掩藏停滞warp的延迟，大大提升了并行性能。

B. 先前的抢占方法

当抢占发生时，每个SM的操作可以是独立的，这意味着有一些SM可能是执行抢占，而另一些SM可以继续执行直到结束。被抢占的SM需要将SM内的上下文存储备份到全局内存。一个SM的上下文就是其执行的状态，包括SIMT栈、寄存器和共享内存。SIMT栈存储的是线程执行信息，例如程序计数器和激活掩码（用于分支处理）。相比于寄存器和共享内存的大小，SIMT栈的大小可以忽略不计，因此在本章我们暂不考虑SIMT栈。一个线程块在其执行的时候占用SM的资源；线程块保持活动状态直到其执行完毕。但是，在其执行的时候，有可能有新的kernel会被发射。当该新的kernel需要满足严格的延迟要求，如果等待上一个kernel执行完毕再开始执行会打破该延迟要求。因此我们需要去抢占一些活动的线程块来为新的kernel的线程块提供资源。但是，线程块的上下文相对来说非常大，将他们存到全局内存将引入相当大的开销，抢占延迟会非常大。如表1所示，抢占延迟（平均上下文切换时间）有可能超过20us。这些延迟给需要满足延迟要求的新kernel造成很大隐患。

为达到较低的抢占延迟的目标，Park等人提出了一种方法，可以简单的丢弃线程块。在这个方法里，SM直接丢弃掉线程块的上下文，并不备份相关上下文。之后直接指向来自更高优先级kernel的新线程块。在这个kernel运行结束以后，SM会重新开始执行被丢弃的线程块。丢弃操作几乎没有抢占延迟开销。但是，并不是所有的kernel都能在任意时间执行丢弃操作。丢弃操作要求kernel是幂等的（idempotent），这意味着该kernel无论执行多少次，运行结果都是相同且独立的，即无原子操作，在丢弃操作发生之前无全局内存的写操作。大多数应用程序并不是幂等的（大约30%的Rodinia应用程序）。幂等性可以变得相对宽松，但记录幂等性的开销非常大。因此，丢弃操作的开销也可能非常大，与需要重复执行的指令成比例。

为了达到较低的抢占开销，SM排泄执行操作被提出。该方法要求在新的kernel的线程块开始执行之前，当前的线程块继续执行，直到结束。这种方法不要求上下文的存储备份，因此抢占开销能够最小化。但是，这种情况的抢占延迟也可能非常高，这是因为执行kernel可能非常大，执行时间会非常长。这很可能导致抢占kernel无法满足其延迟要求。表1包含了我们测量的不同的kernel的单个线程块运行时间。我们可以看到，有的线程块（如Kmeans）的执行时间达到接近1ms。因此，SM排泄执行的方法最适合较短延迟的线程块。

Lin等人提出一种轻量化的上下文切换方法来减少需要拷贝到片外的上下文的大小。这些技术主要包括本地上下文切换，将上下文存到未使用的寄存器或共享内存；清除废弃的寄存器，即降低上下文大小；还有寄存器压缩技术，我们在PEP中也采用了本地上下文切换技术。但是，采用活跃度（liveness）信息要求为每一条指令的每一个寄存器提供一个活跃位（liveness bit），这将引入一个非常大的活跃度表存储在硬件里。为降低这种较大的开销，抢占操作只能在特点的时间点执行，才能尽可能的使用较少的存储单元来存储活跃度信息。而寄存器压缩技术也需要额外的硬件开销，因此我们也不在PEP中采用。

C. GPU中的检查点技术

检查点技术即是将一个运行的进程的状态备份的方法，目的是当出现错误时，能够从检查点恢复操作。GPU检查点技术已经在软件上被实现。虽然检查点技术能够恢复一个进程，但该技术的目的是容错，并不适合于抢占操作。运行进程的设备有可能出现错误，所以有必要将运行状态存储到另一个设备。这是一个非常高延迟的操作，但是相比于错误导致的工作进程丢失，这是非常必要的。对于抢占操作，我们的目标是为抢占kernel达到一个理想的反应时间，因为抢占kernel需要满足一个延迟要求。因此，我们需要存储上下文到设备的全局内存。检查点技术被采用来降低未来的上下文切换的延迟。为引入检查点技术到抢占操作中，限制检查点次数尤为重要，因为我们另一个目标是降低抢占操作的开销。

D.动机

Chimera采用了一个选择算法在抢占请求到来时为不同的线程块选择不同的抢占方法；这个选择是基于对上面介绍的三种方法的选择和平衡。Chimera估计了每一个技术的抢占延迟和开销来选择最有效哦的抢占方法。因此，SM中不同的线程块会被不同的抢占方法被抢占。

但是我们发现排泄执行方法和上下文切换方法会竞争全局内存的带宽。举例来看，在图3中，访存密集型的应用程序LBM会遇到这样的一种冲突。LBM在每个SM中可以执行9个线程块；我们展示了所有10种切换和排泄执行的可能性组合。如果所有的线程块被一个又一个地切换，则不会出现切换和排泄执行的竞争。而另一种情况，当一个SM在上下文切换时，其他SM都在排泄执行时，排泄执行的时间和上下文切换时间都将远长于8个线程块在做上下文切换操作，1个线程块在排泄执行的情况。因此，我们发现带宽竞争会导致Chimera的估计不准确。在这种情况下，当1个线程块在排泄执行时，其他线程块一个接一个被切换时带宽竞争非常小。另一方面，这个排泄执行的线程块不会与其他线程块竞争执行单元。因此，IPC受到排泄执行的线程块的数量影响。

我们也观察到在每个SM中所有的线程块通常选择一种抢占方法。表1展示了线程块执行时间的范围远大于上下文切换的时间范围。因此，对于大kernel，排泄执行时间和上下文切换时间相差巨大。因此，对于小的抢占kernel来说，最佳方法是排泄执行SM内所有的线程块，这可以满足延迟要求的同时接近零抢占开销。相反，对于大kernel来说，如果不满足幂等性要求，则必须做上下文切换。

当前GPU的总上下文大小是每个SM有352KB（寄存器包含256KB，而共享内存包含96KB）。为了传输所有上下文到全局内存，假设带宽被完全利用需要大约15us。之前所有的技术都是被动的，他们至少需要这么长时间来完成上下文切换。为了进一步降低上下文切换的抢占延迟，我们不仅需要减少上下文的大小，还需要一种主动的抢占方法。

检查点技术即是一种主动机制广泛应用于容错处理中；该方法周期性地存储当前进程的执行状态到下一级存储中。类似地，我们可以为了抢占操作，将正在运行的线程块的上下文存储到全局内存中。我们介绍一种新的检查点方法，PEP。我们可以在抢占之前将当前检查点的上下文备份，当真正的抢占操作开始后，我们只需要被动地存储这段时间更新的上下文，这大大减少了抢占等待时间。

3. 设计

在本节中，我们首先给出了主动抢占设计的全局概要图，然后我们将证明预测kernel发射时间和估计抢占时间的可行性。最后，我们提出了基于检查点方法的设计和在线选择算法。

1. 全局设计

我们的方法是基于一个观察：只要延迟和开销是可以接受的，上下文切换可以在线程块执行的任何阶段发生。为了降低延迟和开销，我们将减上下文的大小。为了减少抢占延迟，我们可以提前做上下文切换。在合适的时机我们采用了排泄执行的方法，因为这个方法几乎没有开销。

为了减少上下文的大小，我们采用了脏位来指明一个寄存器活跃与否。因此，我们不会存储未使用或者已经被释放的上下文。我们还会采用Lin等人提出的本地上下文备份的方法，该方法支持让上下文存入空闲的本地内存。在这个方法中，不需要通过互联网络将数据传输到全局内存，因此不占用存储带宽。

我们采用检查点技术来实现主动上下文切换。我们的算法支持在抢占之前的某个检查点存储数据到全局内存。接着继续执行当前kernel，直到抢占请求到来。此时，我们只需要将这段时间相对于上一个检查点的上下文更新存储到全局内存中。如果一个线程块在第一个检查点和抢占发生直接执行完成，则释放第一个检查点存储的上下文。这个方法可以取得远小于一次完整上下文存储的开销。

为了在抢占中实现检查点技术，我们必须限制检查点存储发生的次数；如果我们备份太多次检查点状态，开销很可能无法接受。另一方面，如果一个线程块在多个检查点之后，抢占之前执行完毕，则之前的检查点备份均浪费掉了，同时造成了不少开销。因此，我们有必要预测线程块是否能在抢占点发生的时候依然在执行。此时，我们只需要为在抢占发生时依然在执行的线程块执行检查点技术。我们知道kernel在GPU的执行是在CUDA API调用cudaLaunch之后发生的。在这个API调用之后，一个kernel发射的命令将被发送到流控制管理器中。如果这个命令进入流队列的队列头，kernel的相关信息将被发送到KMU，开始请求SM资源。因此，我们发现kernel发射的时间是可以被预测的。

我们的检查点技术非常适合打kernel。对于小kernel，我们还是会采用排泄执行的方法代替上下文切换。为了采用这两种抢占技术，我们需要估计排泄执行和上下文切换的时机以在线选择合适的抢占方法。

1. 预测和估计

Kernel发射时间的预测和排泄执行时间和上下文切换时间的估计是PEP的关键部分。通过我们对一系列应用程序的研究分析，我们发现有三个时间对我们预测策略的正确性特别关键，分别是kernel发射时间、上下文切换时间和线程块执行时间。Kernel发射时间可以用来预测什么时候抢占请求会真正发射。上下文切换时间和线程块执行时间可以用来确定采用基于检查点的上下文切换技术还是排泄执行技术。表1展示了我们测量的三种延迟。

1. 预测：从对表1的研究我们有两点重要的观察。第一个重要的观察是我们发现kernel发射的时间和上下文切换的时间（接近检查点技术的延迟）是在同一个数量级。这意味着如果我们在开始在预测的时间进行一次检查点备份时，很可能在抢占请求发生时，我们刚好完成有效上下文的存储备份。第二个观察是无论是上下文切换时间还是kernel发射的时间，在大多数情况都远小于线程块的执行时间。对于大kernel的线程块，不太精确的预测不会影响我们是做检查点备份还是排泄执行。因此，一次错误的预测不会对最终的延迟性能和开销造成巨大影响。我们将在本节详细讨论这些延时信息。

前面已经提到，我们必须预测何时抢占请求会出现以避免检查点备份作废。一个cuda应用程序一般包括5步，我们将其概括为5个重要的CUDA API 调用：cudaMalloc, cudaMemcpy(H2D)，cudaLaunch，cudaMemcpy（D2H）和cudaFree。CudaLaunch会触发kernel发射的操作。该操作会将kernel的相关信息传输到GPU，包括线程块的组织信息（网格和块的维度），指针以及共享内存的分配等信息。我们测试了大量的应用程序，观察到kernel发射的时间是在几十微秒的数量级。表1展示了一系列我们测试的应用程序，可以发现kernel发射时间从3.3us到33.3us不等。这个时间是从cudaLaunch被调用到kernel信息被传输到KMU，假设没有在流控制处理器中排队。这种较高的kernel发射时间包括软件api调用和参数数据拷贝到GPU的kernel等待池（pending kernel pool）的过程。因此，这些开销会有很大差异。

此外，每个线程块的平均上下文切换时间从1us到20us不等，取决于每个线程块的上下文的大小。从表1中我们知道上下文切换的时间和kernel发射的时间在同一个数量级。上下文切换的时间大致与kernel发射的时间接近。这意味着如果我们完成第一个检查点备份，抢占请求可能已经发射。在这种情况，我们可以立即释放资源，为新的kernel腾出空间。我们的设计不要求对kernel发射时间的精确预测。这是因为如果一个检查点备份完成之后，真正的抢占请求还未到来，SM可以继续执行线程块知道抢占开始，在抢占时仅需存储更新的上下文。

如表1所示，我们也发现平均线程块的执行时间再1.5us到超过900us不等。这种变化不定的线程块执行时间完全取决于kernel的大小。小kernel的线程块将被排泄执行，这种方法几乎没有开销，同时满足抢占kernel的延迟要求，因为排泄执行的时间非常短。只有大kernel的线程块需要进行上下文切换的操作。因为那些大的线程块的执行时间可以达到几百微秒，因此在kernel发射的api被调用时粗略地预测一个线程块是否会被抢占并不难。为了实现预测的目的，我们设置kernel发射的时间为20us。当一个cudaLaunch被调用，我们回比较预测的kernel发射时间和每个线程块的剩余执行时间。如果预测的kernel发射时间小于线程块的执行时间，则我们立刻采用检查点备份技术。否侧，我们将采用排泄执行该线程块。注意到与线程块执行时间相比，kernel发射的时间相对变化较小。因此，即使一个kernel的真正发射时间不是20us，页不太可能会引起我们选择其他的抢占方法。

现实情况中，kernel发射的时间可能会因为在流控制处理器中的队列等待而延迟。例如，之前的一个长的内存拷贝操作仍然没有结束，堵住了流控制处理器的队列。但是这个延迟在我们的算法中并不是个问题。在平均线程块执行时间远大于kernel发射时间的情况，例如对于CUTCP，这个延迟不太可能比线程块完成执行的时间还快，所以我们的检查点策略不会被浪费。而在平均线程块执行时间小于kernel发射时间的情况，我们会选择排泄执行，所以这个延迟页确定不会影响检查点备份的开销。

我们的预测方法将确定检查点备份的总次数不会超过两次。第一次检查点备份被cudaLaunch调用来触发，而第二次检查点备份被抢占请求触发。因此，检查点备份的开销是可以被控制的。

1. 时间估计：我们可以估计上下文切换的时间和排泄执行的时间来选择不同的抢占策略，即基于检查点技术的上下文切换和排泄执行。我们还需要时间估计来预测抢占是否在当前线程块执行的时候发生。Chimera采用时间估计的方法来比较不同的抢占方法的吞吐量开销。Chimera估计每个线程块的排泄执行时间为线程块剩余的指令数乘以该线程块之前的CPI；上下文切换时间为每个线程块的上下文大小除以全局内存被每个SM分享的带宽。但是这种基于线程块的方法在某些情况下并不准确。而当排泄执行和上下文切换同时进行的时候，这个时间变得无法估计。如图4所示，当一般的线程块在排泄执行，而其他的线程块在做上下文切换时，估计的时间与实际的时间相差巨大。这是因为较少的排泄执行的线程块意味着在流处理器（SP）较小的冲突，上下文切换的线程块在全局内存的带宽上出现更多的冲突。

虽然Chimera的估计对于全部线程块都做排泄执行时更加精确，在某些特定情况仍然有非常不精确的地方。比如说，有的应用程序，如LBM和KS有多个阶段：它们的CPI在不同的时段不一样。在图4中，KS在开始的时候CPI非常低，但其CPI随着程序的执行而不断上升。因此，估计的时间和真实测量的时间相差很大。另外，Chimera估计上下文切换时间来选择抢占策略时，只考虑单个线程块。例如三个线程块在做上下文切换，则需要传输备份的总上下文大小是原上下文大小的三倍。因此真正的尚晓雯切换时间要比Chimera估计的每个线程块上下文时间长3倍。

从表1中可知，线程块的执行时间和线程块的上下文切换时间相差很大。对于大多数的应用程序，我们会选择排泄执行所有的线程块或者上下文切换所有的线程块。因此，排泄执行的时间和上下文切换的时间是可估计的。我们不需要担心上下文切换和排泄执行的互相干扰。为了避免CPI随时间变化的影响，我们在线收集线程块的执行时间。由于不同的线程块执行的指令在大多数情况都是相同的，所以线程块的执行时间是相对稳定的。因此，我们可以采用之前收集的平均线程块执行时间减去已经执行的时间来获取线程块的剩余执行时间。所以，如果在我们需要估计执行时间的时候还未收集到线程块的执行时间，我们将采用Chimera的时间估计方法。为了估计上下文切换的时间，我们采用了最坏情况来估计，即估计的时间为当前SM所有线程块被上下文切换出去的时间。由于上下文切换的时间范围小于线程块的执行时间，我们使用最坏情况来估计比较有保证。

1. 上下文缩减

传统的上下文切换将所有分配的上下文存储都全局内存中。但是，在某一个特定的时间点活跃的上下文总是小于分配的大小，这使得我们需要存储备份的上下文大小相对较少。我们采用脏位实时追踪活跃的上下文。但是，线程块主要有两种上下文，寄存器和共享内存。他们的声明周期是不同的。共享内存时每个线程块私有的。应为这是被程序员来管理的，我们将共享内存的生命周期当成线程块的生命周期。另一方面，寄存器是分配给每一个线程的，并且是在每一个warp中同时执行。因此，一个寄存器的生命周期与warp有关。当一个warp结束后，所有这个warp相关的寄存器将被完全释放。为了追踪寄存器的利用率，在协会的过程中，一旦一个寄存器被写入数据则需要置脏位为1。当warp运行完成了或者我们采用检查点技术后将重置相应脏位。我们可以用类似的方法追踪共享内存写。

图5展示了多个应用程序的脏寄存器的大小相比于被分配的大小。我们收集了不同执行阶段脏寄存器的百分比。我们的初始收集点是线程块执行的25%阶段。Dirty1和Dirty2是脏寄存器在50%和75%线程块执行时间相比于初始收集点的百分比。对于一些包含了非常多指令的kernel，MRI到PF，Dirty1和Dirty2相比于初始状态，平均下降了38.2%和48%。一般来说，Dirty2相比于Dirty1有较少的脏寄存器，因为在75%的时间点部分warp执行结束，并释放了一些寄存器。我们发现这样的脏位分析是足够的，我们不需要采用编译时的活性分析和寄存器值的压缩。

我们还在最后一次检查点备份中采用了本地上下文存储。因为新的kernel可以采用旧kernel没有使用的空间，本地上下文切换的方法得以采用。这将进一步降低实际的抢占延迟。

1. 主动抢占方法设计
2. 检查点备份：检查点备份只在大的被抢占的kernel中被使用，因为他们排泄执行的时间过长。当一个kernel在多个SMs上运行，如果一个cudaLaunch被调用，我们知道新的kernel将要在几十个微秒内被传输到GPU。在这个时候，GPU驱动间刚发送一个信号来激活微程序化陷阱程序。这是通过命令队列和存储映射寄存器。当前的GPUs通过一些可以被开发者直接访问的寄存器来激活抢占操作，但并不是被终端用户来激活。当检测到一个抢占kernel发射的请求，则一个初始检查点备份命令被写入命令队列，之后会修改存储映射的寄存器来开始每个SM的检查点备份。我们测量了NVIDIA GTX 1060 GPU的信号传输时间。这个延迟大约在1.3us，对于不同的应用程序这个值也相对稳定。这个信号将触发一次检查点备份。我们将暂停取新指令，然后开始完成流水线指令。否则，在检查点备份上下文的时候，上下文也在持续变化。如果当前的kernel是计算密集型的，完成流水线指令的过程需要几十个时钟周期。如果当前的kernel是访存密集型的，我们必须等待访存请求返回来完成流水线上的指令。因此每个SM中执行完所有流水线上的指令需要几百个时钟周期。第一个检查点需要备份的上下文是相对于初始状态的脏寄存器和共享内存。

当检查点备份完成后，所有的脏位将被重置。之后，GPU检查是否有新的kernel被传输到KMU。如果kernel在kernel等待池，则一旦获得SM资源则可以开始执行。则当前kernel可以被立刻抢占，因为当前的执行状态已经被存储。否则，当前kernel需要继续执行直到真正的抢占请求到来。当真正的抢占请求到来时，我们只需要存储变化的上下文。因为这一次检查点备份的上下文是相对于上一次检查点备份变化的部分，因此需要备份的上下文非常小，消耗的时间也短很多。因为只有变化的上下文需要被存储，没有重复的数据需要被存储。综上所述，当新的高优先级的kernel的cudaLaunch被调用时，基本检查点备份将开始。因此，新kernel很快被调用。所以，第二次检查点备份的上下文肯定会非常小。除此之外，由于本地上下文存储的利用，需要被存储的上下文的大小可以进一步减少。

被抢占的kernel的恢复和传统的检查点恢复类似。如果我们有两次检查点状态需要恢复，则必须一次接一次的恢复。但是，在这个时候，SM会被限制。所以。所有的带宽将被用来做上下文恢复。