ActivePointers: A Case for Software Address Translation on GPUs

如今，使用 GPU 对计算密集型的应用加速已将越来越常见。但是在 *GPU* 上面实施对大数据的运算还是很富有挑战的。主要原因还是 GPU 没有提供 I/O 抽象（如MMF，内存映射文件），导致程序员会陷入复杂的缓存和 I/O的管理。

为了提高并行运算的性能，GPU 在硬件上取消了一些原本在 *CPU* 上的东西，从而使得 GPU 不能从硬件上支持地址空间管理（address space management）、缺页处理机制（page fault handling mechanisms）。但是如 MMF 这样的 I/O 抽象需要缺页处理机制的支持。

作者希望能够通过软件来弥补 GPU 硬件上缺失的地址翻译和缺页处理机制，并且尽可能减小由此带来的开销，最终能够为 GPU 提供 I/O 抽象，从而能够实现如 MMF、GPU的加密文件系统、基于 GPU 集群的共享内存系统。而这种软件机制是由一种新型的内存指针 —— **ActivePointers** （apointers）所实现。

挑战

由软件带来的问题

降低软件实现翻译所带来的开销。访问任何一个 apointers 都需软件层面的地址翻译，需要尽可能减小此过程带来的开销，否则会严重降低系统的性能。

提供高效的线程级别的翻译。每一个线程都应该能够独立地访问任何虚拟地址，当没有出现缺页的时候，这种地址翻译应该是快速进行，以保证线程的快速访问。同时当出现缺页的时候，需要克服 GPU 硬件中所导致的锁步（lockstep）。

延展性。这个系统需要能够满足大量线程并行运算的需求。

由硬件带来的问题

系统一致性与硬件的不一致性。GPU 每一个流式多处理器（SM）中都有属于自己的共享内存（scratchpad or shared memory）和 L1 cache。每个 SM 内部共享内存和 L1 cache 的改变是独立的。这样会导致某些 SM 内部所储存的映射关系是过时的，再次访问这些映射的时候会导致缺页，从而降低性能。

页面映射改变的异步性。系统改变页面映射和应用的行为是异步的。比如，当系统改变了一个应用程序在页面的映射关系，但是应用程序并不知道，并且在继续访问此页，从而会导致缺页。

设计与实现

准则

固定正在被使用的页面（active page）的映射关系。当一个页面被使用的时候，它不能从页的缓存中被驱逐。这样的话每一个线程都能够在自己私有内存中去储存次页面的映射。否则线程在使用此页面的映射关系，但是此页已经从页面缓存中被驱逐，这样会导致缺页。

跟踪每一个 active page。为了区分页面是否是正在被使用的，系统需要维护一个包含每个页面引用次数的表，防止 active page 被驱逐。

细节

Active Pointers (apointers)

apointers 是一种新的内存指针，其使用逻辑和其它的指针大致相同（Figure 3）。在 apointers 的背后，可以出发缺页，监控内存保护以及帮助跟踪 active pages。

```
int foo(){
//ptr initalized unlinked
  APtr<float> ptr =
    gvmmap(size, O_RDONLY, fd, foffset)
  ptr += 10; // pointer arithmetics
  float f1 = *ptr;  // page fault on the first access
  *ptr=25;  // page fault free access via linked ptr
}
//ptr destroyed and unlinked
```

Figure 3: A simple example of using an *apointer* in GPU code. The *apointer* is initialized by calling the GPU version of `mmap()`, described in Section V.

状态。一个 apointer 有未初始化（uninitialized）、链接（linked）以及未链接（unlinked）三种状态。**未初始化的指针**可以通过其它函数的赋值或者调用虚拟内存管理函数来变成一个未链接的状态。在**未链接**的状态中，指针引用的数据可能在物理内存中，或者数据在页的缓存中，但是指针的映射还没有改变。解引用一个未链接的指针，会导致主要缺页（数据在物理内存中）或者次要缺页（数据在页的缓存中），从而使得指针变成链接状态。在**链接**状态中，指针保存了虚拟地址到物理地址的映射，并且将物理地保存到了硬件的寄存器中，解应用不会导致缺页和翻译查表。当指针被其它指针赋值或者在运算中超出引用的页的范围，这个指针就会变成为被链接的状态。

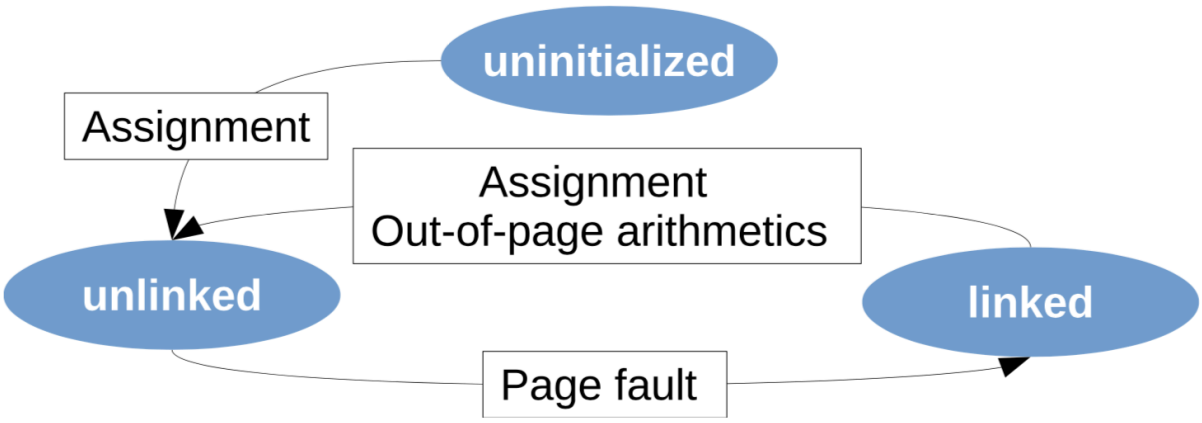

Figure 4: *Apointer* state transition diagram

引用计数。每一个页的缓存中记录者此页被处于连接状态的 apointers 所引用的次数，以此来跟踪 active pages。引用次数为整数的页面不会从缓存中被驱逐。当页面被链接的时候，页面引用次数增加。当指针和页面之间接触引用或者指针在程序外部被销毁的时候，页面引用次数减少

数据结构。一个 apointer 包含了翻译字段（translation field）和元数据字段（metadata field）。

- **翻译字段**被设计成 64 位，以便于编译器将其缓存到硬件寄存器中。它保存着虚拟地址到物理地址的映射。其包含一个有效位，来辨识此指针的状态，3 个访问许可位以及 48 位的地址，12 位的偏移量。当指针被链接的时候，其地址区域保存的是物理地址（aPhysical），当未链接的时候，地址区域保存的是数据在后端存储的位置（xAddress）。具体的翻译过程见 Figure 5.

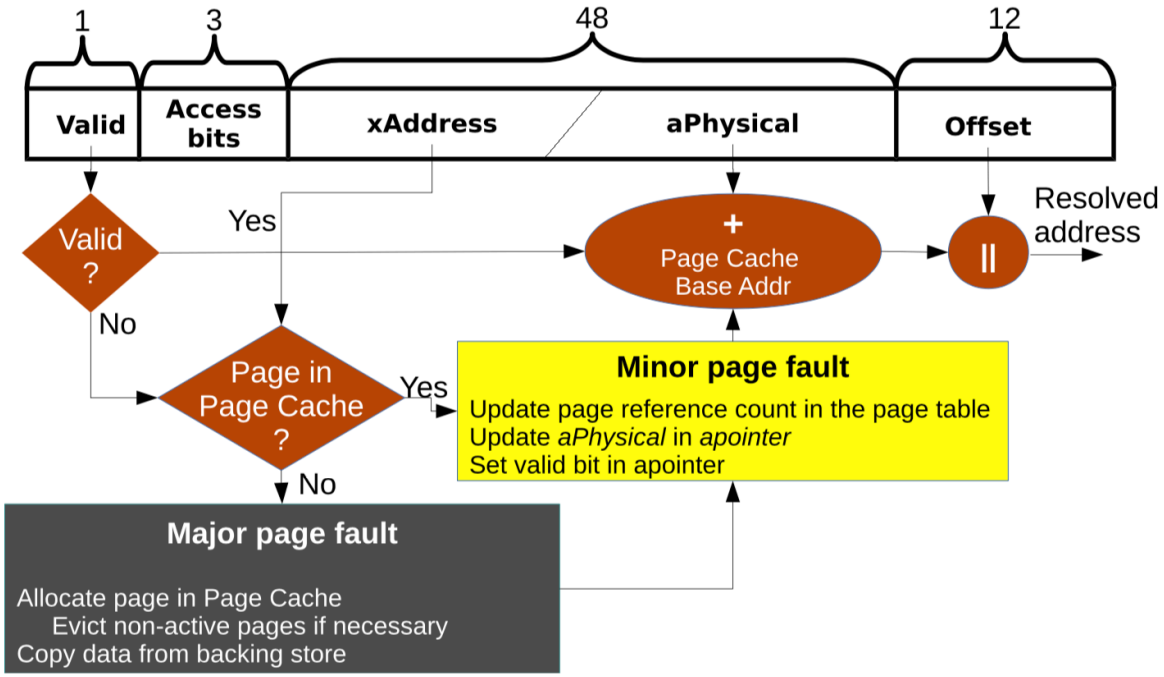

Figure 5: A functional diagram describing the use of *apointers*

- **元数据字段**一般储存在本地内存中，包含了页的偏移，映射区域的大小等信息，只在缺页的时候被使用，所以不影响无缺页的访问。

线程级别的地址翻译

具体的翻译过程可以见 Figure 5.

翻译字段的替代方案。可以在地址字段同时保存 aphysical 和 xAddress，这样能够减少查寻 TLB 的时间，但是可用的地址区段变小。

通过推测性预取（speculative prefetch）在进行优化。通过在所有线程中通过有效位进行投票来决定某个页面是否需要预取，从而减少缺页。同时由于是投票是通过现场并行执行的，所以开销不会很大。

翻译聚合（Translation aggregation）。在一个 warp 采用翻译聚合的方式解决之前提到的锁步的问题。一个 warp 中选取一个领导（leader）代表所有的线程访问共享的数据结构，但是 warp 中的所有线程会共同处理这也缺页，如共同从主机传输数据。翻译聚合能够防止缺页时导致的死锁，减少对共享数据结构访问带来的竞争。为代码可以见 Listing 1.

```
1  T dereference(aptr) {
2    // Test if there are page faults
3    isPagefault!=__all(aptr.valid);
4    if(isPagefault) pageFault(aptr);
5    return *(aptr.aPhys);
6  }
7  pageFault(aptr) {
8     // as long as there unhandled page faults
9    while(true) {
10     // Choose a leader in the group
11     warpLeader=__ffs(__ballot(!aptr.valid));
12     // No more pagefaults to handle
13     if(warpLeader == 0) break;
14     // Broadcast leader's backing store address to
15     // all threads
16     bAddr=__shfl(aptr.bAddr, warpLeader);
17
18     // Aggregate page reference count
19     isRequestHandling=(aptr.bAddr == bAddr);
20     pageRefCount=__popc(__ballot(requestHandling));
21      // Handle page fault using all threads in a warp
22     // Access locks only in warpLeader
23     // Update page reference count
24     aPhys =
25       HandlePageFault(warpLeader, bAddr, pageRefCount);
26     if(isRequestHandling) {
27       aptr.aPhys=aPhys;
28       aptr.valid=1;
29     }
30   }
31 }
```

Listing 1: Translation aggregation using CUDA warp primitives: `__all`: test if all warp threads satisfy a predicate. `__ballot`: fetch one bit across all warp threads, `__shfl`: send a word to all warp threads, `__ffs`: find the first set bit, and `__popc`: count the total number of set bits.

软件实现 TLB

因为 apointer 可以用硬件的寄存器来储存虚拟地址到物理地址的映射，所以 TLB 不是必须的。

每一个线程块维护一个属于自己的 TLB，从而减少之后地址翻译时带来的 PT 的查找。同时 TLB 可以对此线程块中引用的页面进程计数，但是这种计数是线程块私有的。

问题

- **计数丢失。**一个印有数不为 0 的页面不能被简单的驱逐，否则会丢失计数。
- **死锁。**如果一个 warp 中的线程都在对 TLB 中的条目进行计数，可能会导致死锁。
- **重复的 TLB。**全局的 apointers 能够被 GPU 中所有的线程访问，这些线程遍布多个线程块，这也就意味着会创建多个 TLB，并且所复制的引用计数值针对单个线程块可能是不正确的。
-