ActivePointers: A Case for Software Address Translation on GPUs

problem

- no address space management and page fault handling mechanisms to GPU developers
- not allow modifications to memory mappings for running GPU programs and no support for page faults
- no I/O abstractions(especially useful for working with large data sets)
    - fundamental obstacle to building advanced I/O services on discrete GPUs lies in the severely constrained hardware virtual memory (VM):
        - trigger a major page fault on the first access to the mapped region
        - allowing the OS to initialize a physical page with the file contents
        - map it into the process's address space

movitation & goal

movitation

The author wanna overcoming the limitations of the limitation GPU's hardware VM by building it in software. Therefore we can build a system, which can support for the **address translation** and **page fault**.

> The key element of our design is a new type of memory pointer we call *ActivePointer* – *apointer*.

goal

- **primary goal: building advanced I/O services.** It needs virtual address sapcase management and page fault handling to GPUs.
    - **memory mapped files:** It can simplify the application development and optimize the appliction automatically.
    - **one can build an encrypted file system for GPUs by installing custom page fault handlers for encrypt- ing/decrypting file contents on-the-fly:** This design requires no changes to GPU application code and allows seamless offloading of encryption/decryption op- erations to the GPU.
    - **ActivePointers pave the way to building a distributed shared memory system in a cluster of GPUs.**

consideration & challenge

Main challenge: Low translation overhead. Because it is implemented by software, it will bring lantancy, especially for page fault-free access.

Efficient thread-level translation. Every threads can access any vitual address(page-fault free) fast. No deadlock when handling the page fault.

Scalability. The design must accommodate concurrent address translation requests from tens of thousands of con- currently active threads.

Coherence. Cached avitual-aphysical mapping must be kept coherent across all threads, which is because stale mappings will result in error. But unfortunately scratchpad and L1 are on-chip per SM memory, which are not coherent. Even more, there is no good ways to implement additional conheret protocols in software without inter-core interrupts.

Eliminating asynchronous changes of page mappings. The system and application are asynchronous on the page swapping. But building an infrasturcture to invalidate the mapping across all the translation caches is inefficient and hard. Therefore, *constraining the ability of the paging system to revoke application access to a page at will is necessary to prevent asynchronous page mapping changes, eliminating the need for translation cache coherence.*

design & implementation

principles

Active pages with fixed mapping. The page cannot be evicted from the page cache if it is being active used. Therefore each thread can cache the page mapping in its thread-private memory.

> A page whose page mapping is fixed and is not allowed to change is called an *active page*.

Keeping track of active pages. To determine whether a page is an active one, the system maintains a per-page reference count to prevent eviction of active page.

details

active pointers(apointers)

One apointer an be uninitialized, unlinked or linked. An apointer is created uninitialized. It can be initialized by assignment. Then it becomes unlincked. That means that the apointer holds a reference to data that might not be present in *aphysical* memory. Dereferencing an unlinked *apointer* may result in a page fault (minor or major page fault).

> Figure 4

Reference count. Each page in the page cache holds a reference count representing the number of linked *apointers* holding the reference to that page.

- The paging layer cannot evict a page with a positive reference count from the page cache. (*This condition guarantees page-fault free accesses via the linked apointer*.)
- The reference count of a page is automatically incremented and decremented.
    - **Incremente:** the page transitions to the linked state
    - **Decrement:** it becomes unlinked or it is destroyed outside the program scope

Data structure. It comprises two parts:

- **Translation field:** The translation field is specifically designed to fit into 64 bit. This allows the compiler to cache it in a hardware register. It stores the *avirtual*-to-*aphysical* mapping.
    - a *valid* bit: distinguish between linked and unlinked *apointers*
    - page access permission bits
    - address:
        - *aphysical* address
        - *x*Address: For unlinked *apointers*, the mapping data stores the location of the data in the backing store

> Figure 5

- **Metadata:** It can be stored in local memory (usually backed by L1 cache) and includes the page offset and the size of the mapped region, as well as auxiliary data used by the paging system, such as the file or device ID for which the mapping is performed. The metadata is only accessed on page faults and therefore does not affect the performance of fault-free accesses.

thread-level address translation

address translation

- When the page fault mechanism is triggered upon the first access, the *x*Address stored in the unlinked *apointer* is used to check:
    - **major page fault:** whether the relevant data needs to be transferred from the backing store
    - **minor page fault:** whether it is already cached
    - When the page is allocated and initialized, the *x*Address in an *apointer* is replaced with an *aphysical* address and the valid bit is set, marking the*apointer* as linked.
- **Design alternatives:** Long and short *apointers* provide two different ways to balance between the address space size (32/40 bits for *aphysical*/*x*Address vs 60 bits for each) and the cost in terms of the TLB size and runtime overhead.
    - a *short apointer*: the translation field always holds both an *aphysical* address and*x*Address.
    - a *long apointer*:
- **Optimizing performance via speculative prefetch.** Fetching the data from memory, while performing the valid bit voting across all the threads in parallel. The overhead of fault free accesses can be reduced by accessing memory in parallel with checking the *apointer*'s valid bit. These checks are performed by all the warp threads jointly, to decide whether page fault handling is necessary for any of them.

Translation aggregation

If no page fault is encountered by any of the threads, they quickly return the data without divergence. Otherwise, subgroups of threads accessing the same page select one thread from the group as a leader, which handles the page fault for that group.

All the threads in a warp converge to execute the page fault handler jointly. The warp leader is the only one that accesses concurrent data structures.

1. Choose a leader: cheack all the warp threads' valid bits, choose the thread whose valid bit equals 0, and return its number.
2. The answer is zero means no more page fault to handle, otherwise go to the next.
3. Broadcast leader's backing store address to all threads.
4. Aggregate page reference count.
5. Handle page fault using all threads in a warp. Access locks only in warpLeader. Update page reference count

software TLB

The *apointer* design makes it possible to cache an *avir- tual*-to-*aphysical* mapping in hardware registers; therefore a traditional per-core TLB is no longer necessary.

- Each threadblock (up to 1024 threads) maintains its own TLB for its threads. (*when the same set of pages is accessed by multiple threads in a threadblock, adding a TLB may help further reduce page table lookups.*)
- The TLB keeps the *threadblock- private* reference count for each cached page, and serves as a reference count aggregator for the threads in a threadblock.

Problems:

- **Count lost:** a TLB entry with a non-zero reference count can no longer be simply evicted from the TLB upon conflict because the count will be lost.
- **Deadlock:** if the threads in the same warp contend for the TLB space, they may deadlock.
- **Multiple TLBs:** global*apointers* created on the heap or statically in a global array may not use the TLB. These *apointers* can be accessed by all the GPU threads, across multiple threadblocks, and therefore may end up being cached in multiple TLBs, leading to an erroneous reference count duplication.
- **Overhead:** the TLB data structure itself adds overheads to address translation, because the TLB updates are costly.

We therefore allow such threads to update the page count directly in the page table maintained by the paging layer. Adding the TLB is not necessarily advanta- geous in practice.

evaluation & analysis

The best balance between the GPU occupancy and register spillage overhead for our workloads is achieved with 64 registers, because lower occupancy with more registers/thread affects latency hiding.

- 2 × 6-core Intel i7-4960X CPUs at 3.6GHz, with 15MB L3/CPU, with power management and hyperthreading dis- abled for ensuring consistent results.
- a single GPU of the dual NVIDIA Tesla K80.
- Ubuntu Linux kernel 3.13.0-32 with CUDA SDK 7.0 and NVIDIA GPU driver 346.59.

All baseline implementations require fewer than 64 registers/thread and do not spill registers.

Focus

- What is the **overhead of the software address translation layer** on GPUs?
- What is the end-to- end performance of applications that use ActivePointers to map large datasets into GPU memory?

Apointer performance in page-fault free accesses

ways

- We run a test with 32 threads (one warp) where all the threads perform coalesced accesses to different offsets in one page.
- Each access involves a memory read and an increment operation.
- We read the internal GPU clock via clock() intrinsics. We deduct 16 cycles from each measurement to account for measurement overheads.

results

Lantancy overhead

> Table 1

The increment is relatively slow, since the most efficient *apointer*implementation uses 18 instructions vs. only 2 for a sim- ple pointer increment.

Throughput overhead

We run hundreds of warps to **saturate** all compute units in the mapped files and differs from the standard file system access pattern.

We implement a kernel that copies data between two memory regions. Each warp copies 1MB using 4-byte or 8-byte reads/writes per thread. We run the kernel on the GPU using 52 threadblocks, each with 32 warps (1024 threads/threadblock). We use memory tiling, which interleaves memory copy operations to fully utilize memory bandwidth.

> Table 2

analysis

Key of the high throughput: free- computation bubble.

- NVIDIA K80 GPU issues 2056 × 109 instructions per second per GPU
- 240 × 109 bytes/sec of mem- ory bandwidth
- The free-computation bubble is computed as their ratio and equals 8.6 instructions per byte of memory traffic.

The gpu uses the time of memory traffic to operate more instructions.

Compute-intensive workloads

ways

We run 32 warps/threadblock and vary the number of threadblocks from 1 to 52 to show the transition between the latency-sensitive (fewer threadblocks) and throughput- optimized (more threadblocks) execution mode. Each workload reads its data using *apointers* and accumulates the results in a register, written back to global memory at the end of the run. **Full GPU occupancy is attained with 26 threadblocks.**

results

4-byte reads

> Figure 6(a)

Workloads with low compute intensity. As we increase the number of threadblocks, the overheads due to *apointers* decreases significantly.

The most compute intensive workloads. xhibit relatively small overheads for few and many thread- blocks alike, because computations dominate the overall performance.

Rest show more modest improvement, because the size of the free-computation bubble shrinks.

Much better results can be achieved if we batch memory reads into 16-byte loads to amortize the access overheads and increase the size of the free-computation bubble. The results in Figure 6b show much lower overhead, with an average of 20% (7% when excluding FFT).

> Figure 6(b)

analysis

Anomalous performance of FFT. The differences are in the code regions unrelated to the global memory accesses where*apointers* are used. We observe that the order in which the FFT coefficients and the kernel inputs are loaded relative to each other is reversed in two versions. **Compiler artifacts are the main reason.**

Page cache and apointers

Page fault

The best performance, however, is achieved without the TLB with long *apointer* because it avoids the overheads of TLB updates.

Effects of TLB size.

To stress the TLB implementation, we vary the page reuse rate across warps in a threadblock.

TLB is effective when the number of unique pages accessed by each warp is low (high reuse). However, the more unique pages are accessed, the higher the TLB miss rate, and the more pronounced its overhead, leading to decreased performance.

Compute-intensive workloads with page cache

Minor page faults

In this experiment the data is prefetched into the page cache.

Major page faults enabled

We observe less than 1% overhead of *apointers* in all the workloads, including FFT.

End-to-end application performance

We observe that the use of *apointers does not introduce any measurable overheads* over the fastest GPUfs-only im- plementation, and therefore achieves both high end-to-end performance and programming simplicity in this complex I/O-intensive application.

disscusion and related work

conclusion

无CPU操作的GPU程序