2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture

ActivePointers: A Case for Software Address Translation on GPUs

Sagi Shahar, Shai Bergman, Mark Silberstein
Technion – Israel Institute of Technology
{*sagi, shaiberg1*}*@tx.technion.ac.il, mark@ee.technion.ac.il*

***Abstract*—Modern discrete GPUs have been the processors of choice for accelerating compute-intensive applications, but using them in large-scale data processing is extremely challenging. Unfortunately, they do not provide important I/O abstractions long established in the CPU context, such as memory mapped files, which shield programmers from the complexity of buffer and I/O device management. However, implementing these abstractions on GPUs poses a problem: the limited GPU virtual memory system provides no address space management and page fault handling mechanisms to GPU developers, and does not allow modifications to memory mappings for running GPU programs.**

We implement ActivePointers, a *software* address translation layer and paging system that introduces native support for page faults and virtual address space management to GPU programs, and enables the implementation of fully functional memory mapped files on commodity GPUs. Files mapped into GPU memory are accessed using *active pointers*, which behave like regular pointers but access the GPU page cache under the hood, and trigger page faults which are handled on the GPU. We design and evaluate a number of novel mechanisms, including a translation cache in hardware registers and translation aggregation for deadlock-free page fault handling of threads in a single warp.

We extensively evaluate ActivePointers on commodity NVIDIA GPUs using microbenchmarks, and also implement a complex image processing application that constructs a photo collage from a subset of 10 million images stored in a 40GB file. The GPU implementation maps the entire file into GPU memory and accesses it via active pointers. The use of active pointers adds only up to 1% to the application's runtime, while enabling speedups of up to $3.9\times$ over a combined CPU+GPU implementation and $2.6\times$ over a 12-core CPU-only implementation which uses AVX vector instructions.

***Keywords*-Operating systems; Parallel architectures; Memory management;**

I. Introduction

Discrete GPUs have become an integral part of high-performance computing systems, and they continue to be augmented with advanced hardware capabilities that improve their programmability and performance for general purpose parallel workloads. In line with these hardware trends, recent work demonstrates the benefits of providing operating system services, such as access to files and network sockets, directly to GPU programs [1], [2]. These services, coupled with hardware support for direct access to I/O devices from discrete GPUs [3], facilitate the development of GPU-accelerated I/O-intensive applications.

Despite these advances, GPU developers still lack important high-level I/O abstractions, especially useful for working with large data sets. Consider, for example, a database application which uses an index to randomly access parts of very large files. CPU developers commonly use the `mmap()` system call to map the file into a contiguous linear segment in a process's virtual address space. The file is then accessed via an intuitive pointer interface, while the operating system transparently loads the data on-demand and maintains a page cache to optimize application performance. Unfortunately, `mmap()` functionality is not available on GPUs. The recent GPUfs system adds the ability to access files from GPU programs, but does not support memory mapped files either [1].

Such unpredictable data-driven access patterns are common in many applications that deal with large datasets, for example, image similarity search [4], image classification [5], or tweet search [6]. Abundant parallelism and high compute intensity make such applications ideal candidates for GPU acceleration. Yet without the appropriate I/O abstractions, implementing them on GPUs is extremely hard.

The fundamental obstacle to building advanced I/O services on discrete GPUs lies in the severely constrained hardware virtual memory (VM), which lacks essential capabilities necessary for their implementation. For example, the traditional design of memory mapped files in CPUs relies on VM hardware to 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, and map it into the process's address space. In contrast, the majority of the available discrete GPUs have no support for page faults (with the notable exception of the recently announced NVIDIA Pascal), and there are none that provide public interfaces to manage GPU address space or modify memory mappings from running GPU kernels.

In this paper we explore the idea of overcoming the limitations of the GPU's hardware VM entirely *in software*. We describe the design of a lightweight software layer which adds low-overhead support for address translation and page faults to commodity discrete GPUs. The key element of our design is a new type of memory pointer we call *ActivePointer – apointer*. An *apointer* behaves like a traditional C pointer, but contrary to standard GPU pointers, access to an *apointer* may generate a page fault which is handled *on the GPU*.

1063-6897/16 $31.00 © 2016 IEEE
DOI 10.1109/ISCA.2016.58

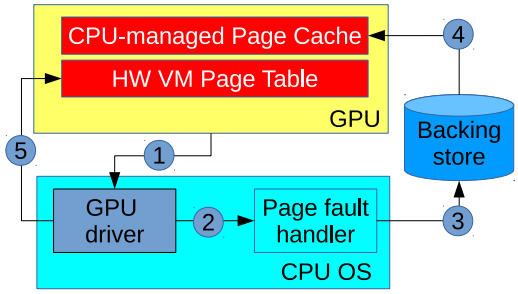


Figure 1: A likely implementation of GPU `mmap()` in a CPU-centric VM management design. (1) The page fault is passed to the GPU driver on the CPU. (2) The CPU executes the page fault handler. (3) The data is copied from the backing store and (4) written into the CPU-managed GPU page cache. (5) The CPU updates the GPU hardware VM page table.

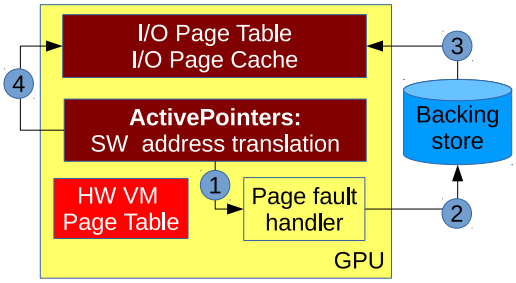


Figure 2: An implementation of GPU `mmap()` in the GPU-centric VM management design using ActivePointers. (1) The address translation layer triggers a page fault which is executed on the GPU (2) Data is copied from the backing store and (3) written into the GPU I/O page cache. (4) The GPU updates the I/O page table. No accesses to the hardware VM page table are necessary.

The primary goal of ActivePointers is to allow GPU developers to build compelling I/O services, which are well established in the CPU context, by bringing virtual address space management and page fault handling to GPUs. First, we focus on memory mapped files, because they significantly simplify application development: they eliminate buffer allocation, read/write system calls, and file pointer arithmetics, as well as enable seamless serialization/deserialization of in-memory data structures to/from files. At the same time, they offer multiple performance benefits that save programmer's optimization efforts, e.g., demand paging and zero-copy, which are not available with read/write calls alone. Second, one can build an encrypted file system for GPUs by installing custom page fault handlers for encrypting/decrypting file contents on-the-fly, like in CryptFS [7]. This design requires no changes to GPU application code and allows seamless offloading of encryption/decryption operations to the GPU, without storing plain-text data in CPU memory. ActivePointers can also provide a unified pointer-based interface for accessing a variety of low-latency storage devices directly from GPUs. Finally, page fault interposition has been useful for implementing software distributed shared memory in a CPU cluster. ActivePointers pave the way to building a distributed shared memory system in a cluster of GPUs.

We illustrate the utility of ActivePointers by implementing a fully functional memory-mapped file abstraction on discrete GPUs. Just as in CPUs, a programmer obtains an *apointer* by mapping a file region into GPU memory, uses it as a regular pointer to read and write file contents, and then unmaps it. *Apointer* page faults are passed to the GPU page cache layer, which manages the page cache and a page table in GPU memory, and performs data movements to and from the host file system.

ActivePointers are designed to *complement* rather than replace the VM hardware in GPUs, and serve as a convenient solution to bridge the functionality gap in discrete GPU systems today. Future generations of discrete GPUs, such as NVIDIA Pascal, are in fact adding page fault support in hardware [8], which naturally raises the question of the relevance of ActivePointers in that case.

One of the contributions of this work, is however, a novel *GPU-centric* application-level VM management design (Figure 2) that we believe to be conceptually different from the *CPU-centric* approach (Figure 1) likely to be implemented in future systems. Indeed, the common hardware VM design found in the recent research publications [9]–[12], and implemented in hybrid HSA-compliant GPU-CPU SoCs [13], [14], is the one in which the GPU page fault handling and VM management are delegated to the GPU driver running on the CPU.

In contrast, the GPU-centric approach we propose here offers certain advantages in the context of implementing advanced I/O abstractions. The GPU-centric approach does not suffer from the scalability bottlenecks of the CPU-centric design, in which the CPU needs to sustain potentially high load while handling multiple concurrent page faults triggered by massively parallel GPU code. On the other hand, running a page fault handler on the GPU may completely eliminate the CPU involvement, including the I/O operations. A key to bypassing the CPU for performing I/O operations is the ability to access peripheral I/O devices directly from the GPU via peer-to-peer Direct Memory Access (DMA), which is already broadly available today [3]. Together, GPU-native page fault handling and peer-to-peer DMA technology help reduce CPU load and power consumption, and improve latency of GPU-originated I/O, which is particularly important as the latency of access to I/O devices is constantly dropping.

There are two primary challenges that we address to make *apointers* practical.

Address translation overhead. The overhead of software address translation, invoked on every access to an *apointer*, may significantly affect the performance of page-fault free accesses. We introduce a novel address translation mechanism that *caches virtual-to-physical mappings in hardware registers*, thus eliminating the majority of page table

lookups. Additionally, we leverage the native latency hiding capabilities of GPUs to hide the remaining translation overheads behind the latency of memory accesses.

Per-thread *apointer* support. Enabling fine-grain accesses to *apointers*, and providing standard pointer semantics from each thread poses a challenge on the GPU hardware, which features lockstep execution of groups of threads (in NVIDIA GPUs 32 threads, which form a warp). We build a translation aggregation mechanism that performs coalesced parallel page translation, and guarantees deadlock-free page fault handling.

We implement ActivePointers for NVIDIA Kepler GPUs using CUDA, and integrate them with the GPUfs file system layer [1] to implement memory mapped files on GPUs.

We extensively evaluate the system performance on a range of microbenchmarks and real workloads. The overhead of the address translation layer alone without page faults is quite low, even when compared to direct memory access without *apointers*. For example, when executing memory copy between two GPU memory regions using *apointers*, we achieve 97% of the maximum GPU memory bandwidth, and an average slowdown of about 9% over eight workloads with different compute-to-memory ratios. While the *apointer* dereferencing and arithmetics operations add computations on each access, we show that their overheads become largely hidden as the GPU occupancy grows. This latency hiding ability of the GPU architecture is the key to providing efficient address translation in software.

We also implement a complex image processing application that builds a photo collage using a large dataset of 40GB worth of small images. The application accesses different parts of the dataset as it processes the input, using a data-driven algorithm to quickly find appropriate images in the dataset. In our implementation, we map the entire dataset into GPU memory and access it from the GPU through *apointers*. The application runs entirely in the GPU and requires no CPU code development. We find that the overhead of ActivePointers is less than 1% compared to the fastest GPU implementation using GPUfs without memory mapped files. Notably, this implementation is up to $2.6\times$ faster than the optimized CPU run on 12 cores which uses 256bit-wise AVX vector instructions, and up to $3.9\times$ faster than our CPU-GPU execution, even on the dataset that is 8 times larger than the GPU physical memory. Importantly, the use of *apointers* enables us to seamlessly support datasets where the images are not aligned at GPU page boundaries, demonstrating the true power of the memory-mapped file abstraction.

Contributions. Our contributions in this paper are as follows:

- We examine the GPU-centric management approach to managing GPU virtual memory in the context of advanced I/O abstractions on GPUs
- We design and implement an efficient software address translation layer for commodity discrete GPUs, and integrate it with a GPU file system layer to build a fully functional memory-mapped file abstraction
- We thoroughly study the performance of the translation layer and its design alternatives
- We present an end-to-end evaluation with a real GPU application that uses a 40GB dataset stored on the host, and features an unpredictable data-driven data access pattern.

We begin with a brief overview of the GPU architecture (§ II), describe the design (§ III) and implementation (§ IV) of the address translation layer, and its integration with GPUfs (§ V). We then evaluate the system (§ VI), and discuss its current limitations and the ways to eliminate them via extensions to hardware and compilers (§ VII). Finally, we review the related work (§ VIII) and conclude (§ IX).

II. Background

This section provides a brief overview of the GPU software/hardware model. We use NVIDIA CUDA terminology since we use NVIDIA GPUs in this work.

GPU architecture. GPUs are massively parallel processors comprised of multiple cores – *streaming multiprocessors* (SM). Each SM contains multiple SIMD vector units, and has several hardware contexts (64 in recent NVIDIA GPUs).

Execution model. An SM executes groups of threads (32 threads per group in NVIDIA GPUs) called *warps* in a lockstep manner. Threads in a *warp* may exercise divergent execution paths. These paths are executed sequentially, slowing down all the threads in the warp. Several warps (up to 32) are grouped in a threadblock and are guaranteed to run on the same SM. A program running on the GPU is called a *GPU kernel*.

GPU memory. All GPU threads share *global* GPU memory. This memory has high bandwidth and is separate from the host memory. Accesses to this memory are cached in a 2-level cache. In addition, all the threads in a threadblock share a small and high-bandwidth on-die scratchpad memory, managed explicitly by GPU code. Finally, each thread is allocated a set of private registers which are not directly accessible from other threads, but via `shuffle` intra-warp instructions.

GPU-CPU interaction. Discrete GPUs are connected to the CPU via a Peripheral Component Interconnect Express (PCIe) bus. GPU access to the CPU memory through the PCIe bus has about $20\times$ lower bandwidth than access to its own memory. Therefore the data is usually moved to GPU memory for faster access prior to kernel execution.

III. Design

Figure 2 shows the high level design of the system that implements memory-mapped file abstraction on GPUs. In this section, we focus on the address translation layer, and discuss the design of the page cache in Section V.

Terminology. The address translation layer provides the abstraction of a contiguous address space on top of scattered *GPU memory pages*, managed by a page cache. The data resides in a *backing store*, such as a disk or remote memory, and the page cache layer transfers the data between memory pages and the backing store upon page fault or swap out.

We refer to the GPU hardware virtual memory as *active physical*, or *aphysical*, memory, and to the address space exposed by the translation layer as *active virtual*, or *avirtual*, memory. This terminology reflects the fact that the translation layer we develop adds a level of indirection on top of hardware virtual memory.

A. Design considerations

We define the following design requirements:

1) **Low translation overhead.** Address translation is performed on every memory access to *avirtual* memory, and therefore the translation overhead must be low. Our main focus is on the performance of page-fault free accesses, which are particularly sensitive to the extra translation overhead.
2) **Efficient thread-level translation.** Each thread may access any virtual address independently of other threads. Page-fault free accesses must be fast even when accessing different *avirtual* addresses, and page fault handling must guarantee deadlock freedom.
3) **Scalability.** The design must accommodate concurrent address translation requests from tens of thousands of concurrently active threads.

Minimizing the address translation overhead is the main challenge. The system-wide page table is accessible to all threads and therefore is stored in global memory. To avoid costly page table lookups on every access, the *avirtual*-to-*aphysical* mappings of frequently used pages should be cached locally in per-SM memory, but implementing such caching in software poses a challenge.

The coherence challenge. Cached *avirtual*-to-*aphysical* mappings must be kept coherent across all threads, because using stale mappings would result in an error if the original page is swapped out. This coherence requirement makes it particularly hard to build an efficient software translation caching mechanism, because the two types of on-chip per-SM memory – a scratchpad and L1 cache – are incoherent across the SMs. As a result, additional coherence protocols for the translation cache must be implemented in software, but it is unclear how to implement them efficiently without inter-core interrupts, which are not supported on GPUs.

Eliminating asynchronous changes of page mappings. Asynchronous changes of the page mapping are the root cause of the coherence problem. For example, the paging system may decide to swap out a page, and the application, unaware of these changes, must experience a page fault when accessing the page. Here, the paging system is fully decoupled from the application. Consequently, a separate hardware/software infrastructure is needed to invalidate the mapping across all the translation caches exactly when it is changed via the TLB shootdown mechanisms. This infrastructure is hard and inefficient to implement as part of GPU kernel. 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*, thus trading some system flexibility for significant gains in performance and simplicity. We explore this design in our work.

B. Design principles

Active pages with fixed mappings. The system guarantees that an *avirtual*-to-*aphysical* mapping for a page may change only if no threads can access the page. In other words, the page cannot be evicted from the page cache if it is being actively used. This guarantee allows each thread to cache the page mapping in its thread-private memory, e.g., *hardware registers*, and safely access it without performing a TLB or page table lookup. A page whose page mapping is fixed and is not allowed to change is called an *active page*.

Keeping track of active pages. The system maintains a per-page reference count to prevent eviction of active pages. The challenge is to keep track of all the references to active pages, and do it transparently to the user. We design a mechanism we call ActivePointers, which controls all accesses to each page and increments the reference count when the page is first accessed, caching its *avirtual*-to-*aphysical* mapping. The mechanism then proactively decrements the count using a heuristic, discarding the cached mapping when the count reaches zero. Our heuristic strives to keep the number of non-evictable pages low, to avoid clogging the system memory, but also reduces unnecessary page table lookups if the reference count is decremented too quickly.

We now describe the main system components that implement these design principles.

C. Active pointers

The key building block of our design is a new type of memory pointer called an *active pointer* or apointer. *Apointers* are used to access *avirtual* memory and behave just like regular memory pointers. They support all the standard pointer operations, such as dereferencing and pointer arithmetics, and can be passed as function parameters or return values. A simple example of using an *apointer* is shown in Figure 3. Under the hood, *apointers* trigger page faults, monitor memory protection and help track active pages, as we discuss below.

***Apointer* states.** An *apointer* can be in one of three states: *uninitialized*, *unlinked* and *linked*.

An *apointer* is created uninitialized. The initialization is performed by either assignment from another *apointer*, or by calling a virtual memory management function such as `gvmmap`, discussed in Section V.

```
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.

A *linked apointer* holds a valid *avirtual*-to-*aphysical* mapping, i.e., it holds a reference to an active page. Dereferencing a linked *apointer* guarantees page fault-free data access, and requires no translation lookup. The system is designed to store the *aphysical* address of a page in the *apointer* itself, that is, in a hardware register that holds the value of the *apointer* when accessing memory.

An *unlinked apointer* holds a reference to data that might not be present in *aphysical* memory. In other words, dereferencing an unlinked *apointer* may result in a page fault. Just as in CPUs, there are *minor* and *major* page faults. A minor page fault occurs when the referenced page is already present in the page cache, but accessing it requires page table lookup. Minor page faults are handled internally by the translation layer. A *major page fault* occurs if the data is not present in the page cache and is passed to and dealt with by the paging system.

***Apointer* and the page reference count.** *Apointers* implement the reference counting for active pages. Specifically, 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 when the page transitions to the linked state, and decremented when it becomes unlinked, or when it is destroyed outside the program scope as in the example in Figure 3. This approach to reference counting works well for the common case of *apointers* allocated in local variables on the stack.

Transition between linked and unlinked states. The *apointer* state transition diagram is presented in Figure 4. The first access to an unlinked *apointer* generates a page fault, moving it into the linked state. An *apointer* transitions to the unlinked state when it is assigned from another *apointer*, for example when it is initialized. The intuition is that such an *apointer* might remain unused, unnecessarily keeping the page pinned in memory. Another transition to the unlinked state is performed when an *apointer* is modified to point outside of the current page, e.g., as a result of a pointer arithmetic operation.

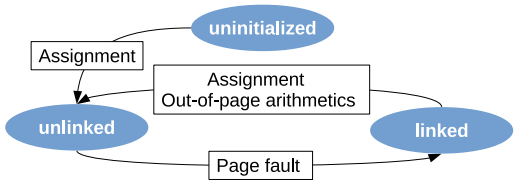


Figure 4: *Apointer* state transition diagram

D. Thread-level address translation

The *apointer* design allows efficient page-fault free memory access for threads in the same warp. This is because the page-fault free logic of the address translation mechanism does not exercise divergent control flow, even when accessing different pages. However, if some threads in the warp encounter a page fault, they execute the slow path, which involves access to the page table and other shared data structures. Naively making each thread in the warp handle its own page fault independently may result in a deadlock, for example when two threads try to acquire the same lock.

To solve this problem we design a warp-level *translation aggregation mechanism* that guarantees deadlock freedom when handling page faults. The mechanism identifies at runtime the subgroups of threads in a warp that follow the same control path and performs their address translation together. The faults of different pages are handled sequentially. The access to the shared data structures is performed by a single *leader* thread on behalf of all the others in the subgroup, eliminating the danger of deadlocks. This mechanism also helps reduce the contention on shared data structures. For example, a page reference count is incremented by the total number of threads in a warp that access that page, instead of incrementing by one for an *apointer* in each thread separately.

We also considered the idea of handling multiple page faults concurrently by different threads in the same warp, but decided against it. Allowing such fine grain access to globally shared data structures makes it much more difficult to guarantee deadlock freedom. However, the added complexity is unlikely to yield performance benefits due to control flow divergence in the page cache logic while accessing different pages.

We discuss the algorithm and its implementation in detail in Section IV-C.

E. Software TLB: pros and cons

The *apointer* design makes it possible to cache an *avirtual*-to-*aphysical* mapping in hardware registers; therefore a traditional per-core TLB is no longer necessary. Yet when the same set of pages is accessed by multiple threads in a threadblock, adding a TLB may help further reduce page table lookups. In our design, each threadblock (up to 1024 threads) maintains its own TLB for its threads. In addition to the page mappings, 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, similar to the optimization of sloppy counters in CPUs [15].

The case for a TLB-less address translation. The TLB design leads to unexpected complications. First, 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. Moreover, if the threads in the same warp contend for the TLB space, they may deadlock. We therefore allow such threads to update the page count directly in the page table maintained by the paging layer, noting that doing so does not affect the correctness of the counter. Second, 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. Finally, the TLB data structure itself adds overheads to address translation, because the TLB updates are costly.

Therefore, adding the TLB is not necessarily advantageous in practice. In fact, the best results in our experiments are achieved by using a TLB-less design (Section VI-C).

IV. Implementation

In this section we describe the implementation of the address translation layer on GPUs using NVIDIA CUDA.

A. Apointer

An *apointer* is implemented as a C++ class. It comprises two parts; a *translation* field, which stores the *avirtual*-to-*aphysical* mapping and is hence used in every access to an *apointer*, and *metadata* fields, used only in page faults. The translation field is specifically designed to fit into 64 bit. This allows the compiler to cache it in a hardware register, which is crucial for reducing the overhead of fault-free accesses. The metadata 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.

The translation field (Figure 5) contains a *valid* bit to distinguish between linked and unlinked *apointers*, page access permission bits, and the mapping data necessary for address translation which we discuss next.

B. Address translation

Figure 5 illustrates the use of *apointers* in address translation. For linked *apointers*, the mapping data holds an *aphysical* address, which, when combined with the page cache base address and the offset within the page, represents the location in the page cache with the requested data. For unlinked *apointers*, the mapping data stores the location of the data in the backing store, e.g., a file offset. We call

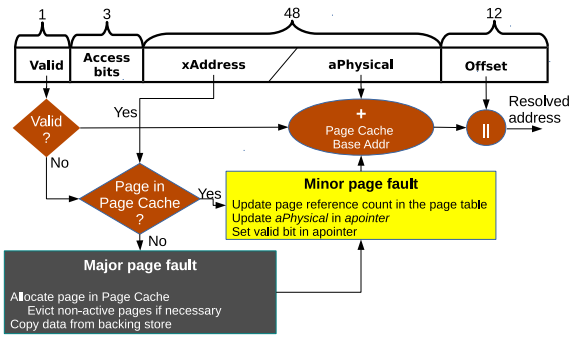


Figure 5: A functional diagram describing the use of *apointers*

this location an e*X*ternal address, or *x*Address. When the page fault mechanism is triggered upon the first access, the *x*Address stored in the unlinked *apointer* is used to check whether the relevant data needs to be transferred from the backing store (major page fault) or whether it is already cached (minor page fault). 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. We consider another design, which we call a *short apointer* (as opposed to a *long apointer*, which is described above and can hold longer addresses), where the translation field always holds both an *aphysical* address and *x*Address. 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. We evaluate both in Section VI-C.

Optimizing performance via speculative prefetch. 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. We take advantage of the instruction level parallelism of NVIDIA GPUs to speculatively fetch the data from memory, while performing the valid bit voting across all the threads in parallel. We evaluate this optimization in Section VI-A.

C. Translation aggregation

We implement a translation aggregation mechanism that enables efficient access to an *apointer* in page-fault free and page fault cases. The pseudocode is shown in Listing 1.

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. Note that all the threads in a warp converge to execute the page fault handler jointly, in order to enable internal page fault logic to be executed more efficiently. For example, the page fault handler may need to

```
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.

copy data from one page to another as a part of batching data transfers from the host, as we discuss in Section V. However, the warp leader is the only one that accesses concurrent data structures.

D. Software TLB

We implement a direct mapped TLB for simplicity. The TLB is implemented as a simple concurrent hash table in the per-threadblock scratchpad memory, enabling lock free search and locked modifications. Each TLB entry requires 12 bytes and 20 bytes for short and long *apointers* respectively, with an additional 4 bytes for entry lock. The total size of a TLB with 32 entries is 512 bytes and 768 bytes for short and long *apointers* respectively, which is less than 5% of the typical per-threadblock scratchpad memory size.

V. Integration with GPUfs

To build a complete system that provides support for memory mapped files on GPUs, we integrate ActivePointers with the GPUfs GPU file system layer [1], [16], [17]. GPUfs exposes a CPU-like file API, allowing GPU programs to read and write files on the host file system. GPUfs also implements a page cache in GPU memory, including support for page pinning and a swapping mechanism for accessing large files.

The main challenge is to enable fine-grain, random accesses to the file contents, which is typical for memory mapped files and differs from the standard file system access pattern. It imposes two requirements on the page cache design: small pages for allowing fine-grain access and frequent page table updates. As we show in Section VI-E, the original GPUfs does not satisfy these requirements since it has been optimized for large pages and rare page table updates. We briefly describe the necessary changes below.

A major page fault handling mechanism that accesses the host file system and transfer data has already been implemented in GPUfs and require only minor changes.

Highly concurrent page cache. We implement a new GPUfs page table using a single concurrent hash table to index pages for all files in the page cache. The total size of the hash table is set to be 16 times larger than the total number of pages in the page cache. This configuration results in a low collision rate (3%), with relatively small memory overhead (less than 5% of the page cache size). Our GPU concurrent hash table implementation uses fine-grain locking per bucket for insertion and lock-free reads.

Optimizing for small page size. We implement batching to reduce data transfer overheads over the PCI when dealing with small 4KB pages. Upon every request to read from a file, the system aggregates several host-to-GPU transfers on the host, and then issues a single call to copy data into the GPU staging area. The GPU threads that invoke the file read are responsible for moving the contents from the staging area into the respective GPU page cache page. The benefits of batching are particularly pronounced for small 4K pages.

A detailed analysis of the GPUfs performance and modifications is available elsewhere [17].

VI. Evaluation

In the evaluation we focus on two main questions: (1) What is the overhead of the software address translation layer on GPUs (§ VI-A and § VI-B) (2) What is the end-to-end performance of applications that use ActivePointers to map large datasets into GPU memory (§ VI-E).

We run our experiments on a SuperMicro server with $2 \times$ 6-core Intel i7-4960X CPUs at 3.6GHz, with 15MB L3/CPU, with power management and hyperthreading disabled for ensuring consistent results. We use a single GPU of the dual NVIDIA Tesla K80. We disable memory error correction. We run Ubuntu Linux kernel 3.13.0-32 with CUDA SDK 7.0 and NVIDIA GPU driver 346.59.

In all our benchmarks, we limit the compiler to allocate up to 64 hardware registers per thread, which enables full SM occupancy. 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. All baseline implementations require fewer than 64 registers/thread and do not spill registers. Therefore, all the reported results effectively include the register pressure overhead of ActivePointers.

Implementation	read	inc	read+ inc	read inc+rw
Raw access	225	32	257	257
Compiler	367 (+63%)	152 (×3.7)	519 (+101%)	585 (+127%)
Optimized PTX	282 (+25%)	–	434 (+69%)	544 (+111%)
Prefetching	271 (+20%)	–	423 (+65%)	435 (+75%)

Table I: GPU cycles when using *apointer* 4-byte read and increment (inc), separately and combined, and with page permission checks (rw), compared to the number of cycles when using a regular pointer (first row). Overhead is shown in parenthesis. Lower is better.

We run each benchmark ten times, clearing the GPU buffer cache between runs when applicable, and use the first 3 runs as a warm up. We report the arithmetic mean of the last 7 iterations. The measured standard deviation for GPU only benchmarks is less than 1%, while benchmarks that include major page faults exhibit standard deviation of up to 10%.

The source code of ActivePointers, all the benchmarks and GPUfs is available online [1].

A. Apointer *performance in page-fault free accesses*

We develop two versions of every benchmark. The baseline version accesses memory directly without *apointers*. The other uses exactly the same code but with *apointers* initialized to map a region in the GPU global memory. This experiment evaluates the overhead of the *apointer* logic for pointer dereferencing and arithmetics. The calls to the GPUfs layer are excluded.

Latency overhead. 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. To measure the latency we record the GPU tick count for each *apointer* operation. We read the internal GPU clock via `clock()` intrinsics. We deduct 16 cycles from each measurement to account for measurement overheads.

Table I shows the latency of memory read and increment operations in GPU clock cycles for different implementations. Low level CUDA assembly (PTX) optimizations and speculative prefetching (described in Section IV-B) reduce the read latency overhead from 63% to 20%. The increment is relatively slow, since the most efficient *apointer* implementation uses 18 instructions vs. only 2 for a simple pointer increment. We also measure the total latency overhead for both operations combined, because we found that memory reads are often followed by pointer increment. The performance for memory read combined with increment and permission checks (read+inc+rw) is compared against raw access with increment. Adding page permission checks increases the overhead, therefore we disable them in all future experiments unless specified otherwise.

Throughput overhead. Unlike the latency benchmark which uses only a single warp, in this experiment we run hundreds of warps to saturate all compute units in the GPU. 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 [18], which interleaves memory copy operations to fully utilize memory bandwidth.

As the baseline we use the bandwidth achieved by the highly optimized NVIDIA `cudaMemcpyDeviceToDevice` function (152GB/s on this GPU).

Table II shows that *apointers* achieve 65% of optimal bandwidth for 4-byte reads and 97% for 8-byte reads. Adding permission checks has no effect on 8-byte reads so it is not shown. The performance difference between the compiler-generated and hand-optimized *apointer* implementations is within 1%, so here and in the rest of the paper we report the results for the compiler-generated version only.

Latency hiding discussion. The significantly lower overheads in the throughput experiment compared to the memory latency benchmark stem from the ability of GPU architecture to perform latency hiding by overlapping memory and compute instructions between different warps. It therefore allows additional instructions to be executed without performance degradation, while the data is being fetched from memory. These additional instructions form a so called *free-computation bubble* [19]. Here we seek to obtain a rough estimate of the size of that bubble on our architecture to explain the encouraging performance results of our experiments.

NVIDIA K80 GPU issues 2056×10^9 instructions per second per GPU, and supports 240×10^9 bytes/sec of memory bandwidth. The free-computation bubble is computed as their ratio and equals 8.6 instructions per byte of memory traffic. In the memory copy experiment which uses four 4-byte memory accesses (two reads and two writes with tiling), the size of the free-computation bubble is about 124 instructions. By inspecting the compiler-generated device assembly code (SASS) we see that the *apointer* access involves 105 instructions. This is slightly smaller than the bubble and in theory should not lead to the observed slowdown. However, the theoretical values assume single cycle execution latency for every instruction, which is not the case in practice. Therefore, the actual size of the free-computation bubble is smaller, hence the overhead can be observed. Using 8-byte accesses, however, doubles the size of the bubble and indeed hides the *apointer* overheads almost completely.

The actual size of the free-computation bubble depends on the application instruction mix – smaller for compute intensive and larger for memory intensive workloads. However, compute intensive tasks perform fewer memory accesses so the address translation overheads are expected to be less pronounced. We next report the results of the experiments that demonstrate this point.

[1] https://github.com/gpufs/gpufs

Implementation	4-byte	4-byte+rw	8-byte
Compiler	99.7GB/s (65.4%)	97.7 (64.1%)	148.7 (97.6%)

Table II: Memory bandwidth in GB/s achieved by memory copy kernel, compared to the maximum achievable bandwidth of 152GB/s (in parentheses). Higher is better.

B. Compute-intensive workloads

We evaluate the *apointer* performance on a range of workloads, each with a different amount of computation per memory access, also called *compute intensity*.

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. Full GPU occupancy is attained with 26 threadblocks. 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. This behavior matches a common use case where data is being read from external memory, while the generated output is significantly smaller and can fit in internal memory. The data accesses in this experiment do not use memory tiling.

We evaluate the following workloads:

1) **Add** – Performs element-wise addition of two large vectors.
2) **Read** – Performs a simple read of a large vector.
3) **Random** – Each thread reads an element from global memory, and generates a pseudo-random number using it as a seed. For this workload, we perform different numbers of iterations for the pseudo-random generation in order to estimate different amounts of computation per memory read.
4) **Reduce** – Each warp reads a 32-element vector and performs reduction by summing up the values using warp-level shuffle instructions for intra-warp communication.
5) **FFT** – Each warp reads a 32-element vector and calculates the FFT transform of the vector using warp level shuffles. The FFT coefficients are stored in read-only constant memory.
6) **Bitonic sort** – Each warp reads a 32-element vector and sorts its elements using the bitonic sort algorithm, which is implemented using warp level shuffles.

The baseline implementations of these workloads are identical to the *apointer* versions, except that they use regular memory pointers instead.

Figure 6a shows the overheads of *apointers* over the baseline when performing 4-byte reads. We sort the workloads in the order of increasing compute intensity.

As we increase the number of threadblocks, the overheads due to *apointers* decreases significantly, in particular for the workloads with low compute intensity, such as Add and Read, which improve more than two-fold. The most compute intensive workloads, such as Random 50 and Bitonic sort, exhibit relatively small overheads for few and many threadblocks alike, because computations dominate the overall performance. The rest of the workloads (except for FFT which we discuss below) 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).

Anomalous performance of FFT. The overhead of the FFT workload is significantly higher than that of any other workloads with similar compute intensity, such as Reduce, and it remains high regardless of the number of threadblocks. We note that this workload exhibits no register spillage in both versions.

We find that such behavior is the result of significant differences in the generated SASS code between the *apointer* and the baseline. Interestingly, the differences are in the code regions unrelated to the global memory accesses where *apointers* are used. For example, 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. We believe that these compiler artifacts are the main reason for the extremely high overhead of this workload here and in the rest of the FFT experiments, which we include for completeness.

C. Page cache and apointers

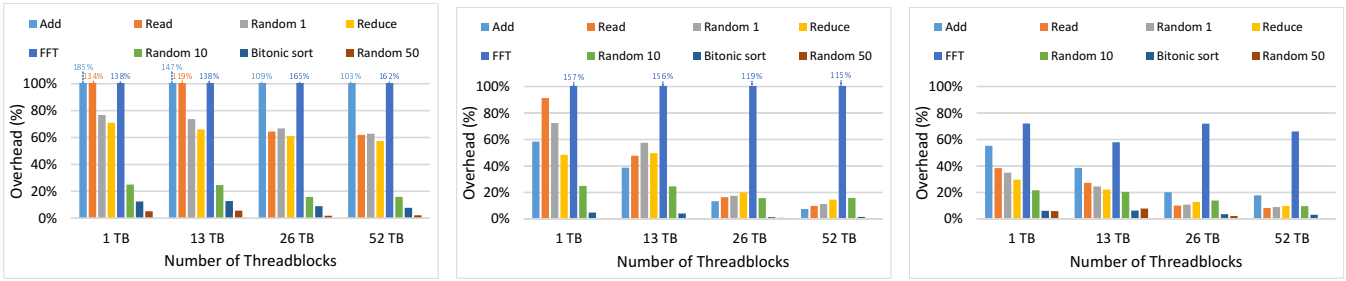
We next evaluate the overhead of adding the software address translation layer to the GPUfs page cache. Unlike the previous experiments, in these experiments accesses through the *apointer* might trigger major or minor page faults. Here, and in all other experiments in this section, we configure the page size to be 4KB.

Page faults. We evaluate the overhead imposed by the *apointer* page fault logic on top of the original `gmmap ()` call in GPUfs. `gmmap ()` locks the page up in the page table (minor page fault) and brings the data from the host (major page fault) if necessary.

We run a kernel with 52 threadblocks each with 32 warps. Each warp accesses 512 different pages in a loop, with all threads in a warp accessing the same page. The baseline uses `gmmap ()` to map a new page in each iteration. The *apointer* version calls `gvmmap()` once in each threadblock to initialize the *apointer*, and then uses pointer arithmetics to access other pages. We store the file in CPU RAM, using RAMfs in-memory file system to measure the worst-case overheads of *apointers*.

We use the same workload to evaluate both major and minor page faults. We run the kernel twice. We use the first execution to measure the cost of major page faults, and the second execution to measure minor page faults, such that the first one serves for the page cache warmup.

Table III shows the overheads over the baseline for different *apointer* implementations. For major page faults, all overheads are within the measured standard deviation



(a) 4-byte reads (b) 16-byte reads (c) 4-byte reads with GPUfs

Figure 6: *Apointer* overheads as a function of GPU occupancy (in number of threadblocks) (a) for 4-byte reads and (b) 16-byte reads excluding GPUfs (§ VI-B), and (c) for 4-byte reads with GPUfs (§ VI-C). Lower is better.

Implemetation	Minor Pagefault	Major Pagefault
***Apointer* Short**	20%	No observable overhead
***Apointer* Long**	24%	No observable overhead
no TLB	13%	No observable overhead

Table III: The overhead of short *apointer*, long *apointers*, and long *apointers* without TLB, with major and minor page faults. Lower is better.

because they are masked by the memory transfers from the host to the GPU. For minor page faults, both short and long *apointer* types which use TLB behave similarly. The best performance, however, is achieved without the TLB with long *apointer* because it avoids the overheads of TLB updates, as we analyze next.

Effects of TLB size. We run a kernel using a single threadblock comprised of 32 warps (1024 threads). Each thread reads one 4-byte element. All pages already reside in the page cache, so only minor page faults are triggered. To stress the TLB implementation, we vary the page reuse rate across warps in a threadblock, that is, the number of distinct pages accessed by a threadblock. Overall, each warp reads 4KB. The read offset inside the page is unique per warp, so that there is no data reuse across warps.

Figure 7 shows the access time per page. As expected, the 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. The performance without the TLB improves exactly when the TLB becomes inefficient, because it avoids TLB updates.

D. Compute-intensive workloads with page cache.

We use the same compute-intensive workloads as in Section VI-B, but now with GPUfs to access files. We use GPUfs without *apointers* as the baseline, and implement an *apointer*-based version with memory mapped files for input and output. Each thread performs 4-byte accesses and reads one page per warp, (one page fault per 32 accesses).

Minor page faults. In this experiment the data is prefetched into the page cache. Figure 6c shows the *apointer* overheads on top of GPUfs for the best performing *apointer* implementation without the TLB. We observe only 16% slowdown on average for fully utilized GPU (excluding FFT). We also repeat this experiment with 16-byte loads, which results in the overhead decreased to 9% for full utilization (not shown in the graph).

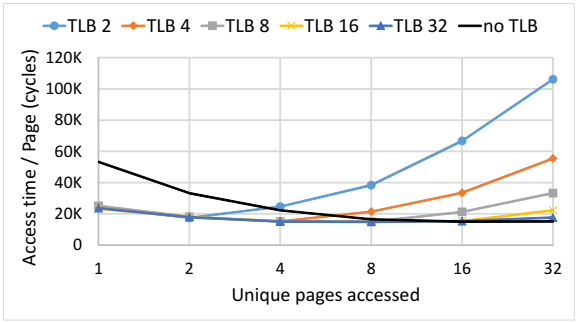


Figure 7: Read access times in cycles per page, as a function of unique pages accessed per threadblock, for different TLB sizes. Lower is better.

Major page faults enabled. We run the same experiment, but with major faults enabled. We observe less than 1% overhead of *apointers* in all the workloads, including FFT.

E. End-to-end application performance

We use an image collage workload to evaluate the end-to-end performance of the system.

Image collage. The image collage (see Figure 8 is created by replacing blocks in the input image with "similar" images from a large dataset, where the similarity is defined as the Euclidean distance between image color histograms [20]. To quickly find an image in a large dataset, the algorithm employs a Locality Sensitive Hashing (LSH) [21] heuristic, which narrows down the exhaustive search to only a few images in the same LSH *bucket*. The dataset images are placed in buckets indexed by the LSH keys derived from their histograms. For each block in the input, we search for the replacement image among the candidate images located in the buckets indexed by the block's LSH keys.

We use 10 million images from the tiny image dataset [22]. Their histograms are pre-computed and stored

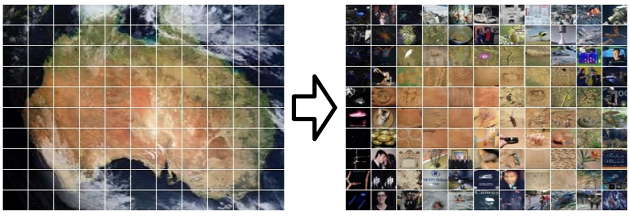


Figure 8: Collage example

in a file, with each histogram padded to 4KB. The size of the file with all the histograms is 38.14GB.

Our GPU implementation calculates the histogram of each 32×32 block, computes its LSH keys, reads the candidate histograms from the respective buckets stored in a file, exhaustively searches for the closest ones among the candidates, and in the end produces the indexes of the images to use in the collage. All these steps are performed in a single GPU kernel. The final stage of creating the output image is executed on the CPU.

GPUfs performance. We first evaluate the GPUfs performance without *apointers* by implementing several versions of the image collage algorithm:

1) CPU only (baseline) – This version uses Intel's Threading Building Blocks (TBB) and 256bit AVX instructions to execute on 12 CPU cores.
2) CPU+GPU – The GPU computes the LSH keys, and the CPU then groups them, eliminates duplicates, reads the candidates from the dataset, and invokes the GPU to search among candidates. GPUfs is not used.
3) GPUfs – All stages are executed on the GPU, using the warp-level GPUfs API with the new paging subsystem.
4) GPUfs + ActivePointers– As in GPUfs, but while mapping the whole file with the image dataset into GPU memory and accessing through *apointers*.

We use several high-resolution images of different contents and sizes. We store the entire dataset in CPU RAM using the RAMfs file system. The GPUfs page cache size is set to 2GB out of 12GB GPU RAM.

Figure 9 shows the relative speedup over the baseline CPU run for each implementation and input combinations. Different inputs exhibit different levels of data reuse (specified in labels on top of the curve), because in larger images more visually similar blocks are available. The GPUfs version outperforms both CPU and CPU+GPU for large image sizes, with an average speedup of 1.6× and 2.6× over the CPU and CPU-GPU versions respectively.

For the largest image, the size of the candidate images to be processed exceeds the size of the page cache, thus some data gets evicted from the cache. However we observe no significant slowdown.

Image collage using memory mapped files via *apointers*. The last bar in Figure 9 shows the end-to-end application performance when using *apointers* to access the dataset.

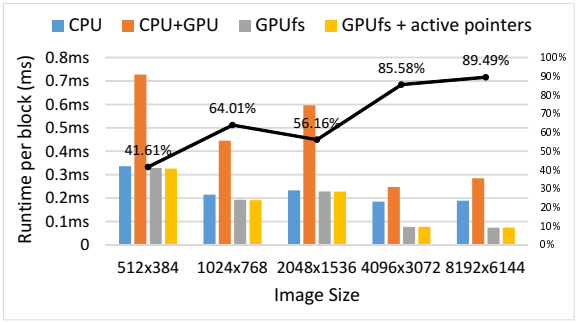


Figure 9: Runtime of the image collage implementations, normalized per image block. Lower is better. The right axis shows data reuse per input.

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

Unaligned access. We highlight the major usability benefit of memory mapped files: linear address space abstraction which is oblivious to page boundaries. We reimplement the collage workload while removing the padding, thereby storing the 3KB histograms in a file, so they are no longer aligned at the page boundary. The code using *apointers* works without any modifications, whereas the version using the original `gmmap` requires significant code changes.

VII. Discussion

The software-only address translation design presented in this paper shows promise as a building block for implementing convenient I/O services on commodity discrete GPUs. However, its performance and practical appeal can be further improved with only minor changes to existing compilers and GPU hardware.

Register pressure. Register pressure is a well-known problem in many GPU applications. While the *apointer* itself does not add new registers (it uses 64 bit, like standard pointers), the *apointer* operations add additional registers, which can lead to performance degradation. The NVIDIA compiler sometimes struggles with the register allocation, spilling registers in performance-critical loops. Fortunately, improved compiler support along with the increase in the number of registers – double in the recent hardware generation (Kepler K40 GPU vs. Kepler K80 GPU) – makes the register pressure less critical for current and future architectures.

Compiler support. Having access to the front end CUDA compiler would allow additional code optimizations, such as different implementations for memory reads and writes, as well as make possible static code analysis to reduce the number of *apointer* bound checks [23].

Instructions for boundary checking and pointer increment. Page boundary checking and pointer arithmetics are the main sources of *apointer* performance overheads, and the main reason for higher register pressure. Hardware extensions for these operations, similar to the ones proposed for CPUs [23], and special instructions which fuse shuffle and integer arithmetics, could help reduce or eliminate these overheads.

I/O preemption. ActivePointers raise the problem of I/O preemption for GPU threads. A major page fault incurs a long-latency access to the backing store, e.g., a disk. On today's GPUs, the stalled warp wastes the SM resources while waiting for data, calling for the addition of a hardware-assisted threadblock preemption mechanism [24].

VIII. Related work

Our work relates to a variety of topics in computer architecture, operating system design, compilers, concurrent algorithms, and GPU computing.

Hardware virtual memory in GPUs. Recent works propose hardware support for virtual memory on GPUs which supports page faults [9], [12]. Existing CPU-GPU SoCs, such as AMD A10 [14], also provide shared virtual memory and page fault support. These devices, however, exercise a different design point and their performance cannot compete with that of discrete GPUs, which are the focus of this paper. The applicability of our work to APUs/HSA is, however, an open research question, which we intend to address in the future.

Software memory management on GPUs. NVIDIA recently introduced a Unified Virtual Memory mechanism based on the Asymmetric Distributed Shared Memory design [25]. This design allows both CPU and GPU to share data using a unified address space. The memory is allocated on the GPU and can be moved transparently to the CPU when CPU processes access it. It does not support page faults on the device; the memory is thus moved back to the GPU upon kernel invocation. The Region-based Software Virtual Memory library [26] initially allocates memory on the CPU, and is able to transfer data between both systems upon request. This approach, however, still requires the entire dataset to be copied into a dedicated memory location on the host prior to GPU execution, and does not support direct access to the file system from the GPU.

System abstractions for GPUs. GPUfs and GPUnet [1], [2], [16] provide file access and networking directly to GPU programs. We enhance the GPUfs layer by redesigning the page cache to allow higher concurrency and smaller page size.

Large dataset support for GPUs. Several approaches have been proposed to handle datasets larger than the GPU physical memory. Some of them are application-specific, e.g., Shredder [27], while others [28] require a special programming model. Whether application specific or requiring a special programming model, they all use data chunking and multiple kernel invocations to achieve their goal. This makes them inapplicable when dealing with data dependent access patterns.

The authors of VAST [29] propose to predict application data accesses before kernel invocation to pre-load its data into the GPU. This mechanism, however, cannot handle data-driven accesses.

Pointer instrumentation. Our work uses techniques similar to the ones used for pointer safety and C-style arrays boundary checking [30]–[32]. The authors of those papers suggest the use of both compiler and hardware support to reduce the overhead of these checks, including static analysis to eliminate redundant checks, and special hardware support to speed them up [23]. We believe that our work can also benefit from these techniques.

Managed GPU languages. Programming languages for heterogeneous systems such as Harlan [33] avoid the need for shared virtual memory by providing higher level abstractions and data structures. Our work differs in that it aims to provide memory-mapped file abstractions for low-level C++ programs.

Software-managed address translation. A software mechanism has previously been proposed for virtual address translations on CPUs [34]. Our approach is orthogonal to this work: it targets different processor architecture and employs different optimization techniques.

IX. Conclusion

We describe ActivePointers, a lightweight software-only address translation mechanism for GPUs, which drives a GPU-centric virtual memory management design with page fault handling and address space modification from GPU programs. We demonstrate the benefits and performance of ActivePointers by implementing a fully functional memory-mapped files abstraction on NVIDIA GPUs. We achieve low overhead address translation thanks to (1) a co-design of a page cache and a translation mechanisms which enables safe caching of the virtual-to-physical mappings in per-thread hardware registers, and (2) GPU inherent latency hiding capabilities which hide the translation overheads.

The current software-only design is complementary to the hardware VM, and works well for accessing I/O devices. Broader applications of ActivePointers beyond the device I/O will likely require special hardware support such as the one discussed in § VII. These hardware changes, however, are quite different and potentially less intrusive than the traditional hardware VM mechanisms. We believe, therefore, that this work will encourage pursuing new research directions toward providing better hardware support for high-level operating system abstractions in future GPUs.

ACKNOWLEDGEMENTS

Mark Silberstein is supported by the Israel Science Foundation (grant No. 1138/14), the Israeli Ministry of Science, and the Israeli Ministry of Economics via HiPer consortium.

REFERENCES

[1] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, "GPUfs: integrating file systems with GPUs," in *ASPLOS'13*. ACM, 2013.

[2] S. Kim, S. Huh, X. Z. Yige Hu, A. Wated, E. Witchel, and M. Silberstein, "GPUnet: Networking Abstractions for GPU Programs," in *OSDI'14*. USENIX, 2014, pp. 6–8.

[3] GPUDirect, "GPUDirect RDMA," http://docs.nvidia.com/cuda/gpudirect-rdma/index.html, 2015.

[4] J. Pan and D. Manocha, "Fast GPU-based Locality Sensitive Hashing For K-nearest Neighbor Computation," in *SIGSPATIAL'11*. ACM, 2011, pp. 211–220.

[5] A. S. S. Michel and R. Schenkel, "RankReduce–Processing K-Nearest Neighbor Queries on Top of MapReduce," in *Workshop on Large-Scale Distributed Systems for Information Retrieval*, 2010, pp. 13–18.

[6] S. Petrović, M. Osborne, and V. Lavrenko, "Streaming First Story Detection with Application to Twitter," in *Annual Conference of the Association for Computational Linguistics*, 2010, pp. 181–189.

[7] E. Zadok, I. Badulescu, and A. Shender, "CryptFS: A Stackable Vnode Level Encryption File System," Tech. Rep., 1998.

[8] HMM, "Heterogeneous Memory Management (mirror process address space on a device MMU)," https://lwn.net/Articles/597289/, 2014.

[9] J. Power, M. D. Hill, and D. A. Wood, "Supporting x86-64 Address Translation for 100s of GPU Lanes," in *HPCA'14*. IEEE, 2014, pp. 568–578.

[10] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, "Page Placement Strategies for GPUs within Heterogeneous Memory Systems," in *ASPLOS'15*. ACM, 2015, pp. 607–618.

[11] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch, "Unlocking Bandwidth for GPUs in CC-NUMA Systems," in *HPCA'15*. IEEE, 2015, pp. 354–365.

[12] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural Support for Address Translation on GPUs," in *ASPLOS'14*. ACM, 2014.

[13] P. Rogers, "Heterogeneous System Architecture Overview," in *Hot Chips'13*, vol. 25, 2013.

[14] D. Bouvier and B. Sander, "Applying AMD's Kaveri APU for Heterogeneous Computing," in *Hot Chips'14*, 2014.

[15] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, F. M. Kaashoek, R. Morris, and N. Zeldovich, "An Analysis of Linux Scalability to Many Cores," in *OSDI'10*, vol. 10, no. 13. USENIX, 2010, pp. 86–93.

[16] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, "GPUfs: Integrating a File System with GPUs," *TOCS*, vol. 32, no. 1, p. 1, 2014.

[17] S. Shahar and M. Silberstein, "Supporting data-driven I/O on GPUs using GPUfs," in *SYSTOR'16*. ACM, 2016.

[18] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. mei W Hwu, "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU using CUDA," in *PPoPP'08*. ACM, 2008, pp. 73–82.

[19] D. Merrill and A. Grimshaw, "High Performance and Scalable Radix Sorting: A Case Study of Implementing Dynamic Parallelism for GPU Computing," *Parallel Processing Letters*, vol. 21, no. 02, pp. 245–272, 2011.

[20] S.-H. Cha and S. N. Srihari, "On Measuring the Distance Between Histograms," *Pattern Recognition*, vol. 35, no. 6, pp. 1355–1370, 2002.

[21] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-Sensitive Hashing Scheme Based on P-Stable Distributions," in *Annual Symp. on Comp. Geometry*. ACM, 2004, pp. 253–262.

[22] A. Torralba, R. Fergus, and W. T. Freeman, "80 Million Tiny Images: A Large Data Set for Nonparametric Object and Scene Recognition," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 30, no. 11, pp. 1958–1970, 2008.

[23] S. Nagarakatte, M. M. Martin, and S. Zdancewic, "WatchdogLite: Hardware-Accelerated Compiler-Based Pointer Checking," in *HPCA'14*. IEEE, 2014, p. 175.

[24] L. Zeno, A. Mendelson, and M. Silberstein, "GPUpIO: The case for I/O-driven preemption on GPUs," in *Workshop on General Purpose GPU (GPGPU'16)*. ACM, 2016.

[25] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W. mei W Hwu, "An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1. ACM, 2010, pp. 347–358.

[26] F. Ji, H. Lin, and X. Ma, "RSVM: A Region-Based Software Virtual Memory for GPU," in *PACT'13*. IEEE, 2013, pp. 269–278.

[27] P. Bhatotia, R. Rodrigues, and A. Verma, "Shredder: GPU-Accelerated Incremental Storage and Computation." in *FAST'14*. USENIX, 2012, p. 14.

[28] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "PTask: Operating System Abstractions to Manage GPUs as Compute Devices," in *SOSP'11*. ACM, 2011, pp. 233–248.

[29] J. Lee, M. Samadi, and S. Mahlke, "VAST: The Illusion of a Large Memory Space for GPUs," in *PACT'14*. ACM, 2014, pp. 443–454.

[30] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: Type-Safe Retrofitting of Legacy Software," *TOPLAS'05*, vol. 27, no. 3, pp. 477–526, 2005.

[31] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C," in *ACM Sigplan Notices*, vol. 44, no. 6. ACM, 2009, pp. 245–258.

[32] J. Devietti, C. Blundell, M. M. Martin, and S. Zdancewic, "Hardbound: Architectural Support for Spatial Safety of the C Programming Language," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 2, pp. 103–114, 2008.

[33] E. Holk, R. Newton, J. Siek, and A. Lumsdaine, "Region-Based Memory Management for GPU Programming Languages: Enabling Rich Data Structures on a Spartan Host," in *OOPSLA'14*. ACM, 2014, pp. 141–155.

[34] B. Jacob and T. Mudge, "Software-Managed Address Translation," in *HPCA'97*. IEEE, 1997, pp. 156–167.